



**POLITECHNIKA LUBELSKA
WYDZIAŁ ELEKTROTECHNIKI
I INFORMATYKI**

**KIERUNEK STUDIÓW
INFORMATYKA**

*MATERIAŁY DO ZAJĘĆ
LABORATORYJNYCH*

Programowanie obiektowe w Java

Autor:
mgr inż. Piotr Wójcicki

Lublin 2020



Fundusze Europejskie
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny



INFORMACJA O PRZEDMIOCIE

Cele przedmiotu:

- Cel 1. Zapoznanie studentów ze składnią języka Java i strukturą programu.
- Cel 2. Zapoznanie studentów z zasadami programowania obiektowego w języku Java.
- Cel 3. Przedstawienie metod tworzenia aplikacji konsolowych w języku Java.
- Cel 4. Przedstawienie metod tworzenia aplikacji z graficznym interfejsem użytkownika.

Efekty kształcenia w zakresie umiejętności:

- Efekt 1. Student potrafi stworzyć aplikację w języku Java zgodnie z zasadami programowania obiektowego posługując się narzędziem IDE.
- Efekt 2. Student potrafi wybrać i zastosować w praktyce zasady programowania obiektowego do realizacji zadania programistycznego.

Literatura do zajęć:

- Literatura podstawowa
 1. Eckel B.: Thinking in Java, Wydanie IV, Wydawnictwo Helion, 2006
 2. Horstmann C. S.: Java 9. Przewodnik doświadczonego programisty. Wydanie II, Helion, 2018
 3. Bloch J.: Java. Efektywne programowanie. Wydanie III, Helion, 2018
 4. Horstmann C. S. , Java 2. Podstawy, Wydanie X, Helion, 2016
 5. Schildt H.: Java. Przewodnik dla początkujących. Wydanie VI
 6. Schildt H.: Java. Kompendium programisty. Wydanie X (Java 9), Oracle Press, Helion
 7. Sierra K., Bates B.: Headfirst Java, Wydanie II, Wydawnictwo Helion, 2011
 8. Barteczko K.: Java : programowanie praktyczne od podstaw, Wydawnictwo PWN 2014
 9. Krawiec J.: Java : programowanie obiektowe w praktyce, Oficyna Wydawnicza Politechniki Warszawskiej, 2017
 10. Kubiak M.: Java: zadania z programowania z przykładowymi rozwiązaniami, Wydawnictwo Helion, 2011.
- Literatura uzupełniająca
 1. Horstmann C. S.: Java 2. Techniki zaawansowane, Wydanie X, Helion, 2016
 2. Czerniak J.: Podstawy programowania Pascal/Java/C# : skrypt do wykładu i laboratoriów, Wydawnictwo Uniwersytetu Kazimierza Wielkiego, 2016
 3. Kamińska A.: Java. Kurs podstawowy. Najnowsza wersja JAVA SE 6, Wydawnictwo NAKOM , 2008

Metody i kryteria oceny:

- Oceny cząstkowe:
 - Ocena 1 Przygotowanie merytoryczne do zajęć laboratoryjnych na podstawie: wykładów, literatury, pytań kontrolnych do zajęć.
 - Ocena 2 Zaliczenie pisemne z laboratorium (dwa kolokwia)



- Ocena końcowa - zaliczenie przedmiotu:
 - Pozytywne oceny cząstkowe.
 - Ewentualne dodatkowe wymagania prowadzącego zajęcia.

Informacje wstępne

- Głównym celem przedmiotu Programowanie obiektowe w Java jest zapoznanie studentów z podstawami języka Java oraz wybranymi aspektami programowania obiektowego. Zakłada się, że student dobrze opanował dotychczasowy materiał toku studiów.
- Przewidziane przykłady i zadania laboratoryjne nie wyczerpują wszystkich poprawnych rozwiązań danego problemu.
- Sugeruje się, aby podczas wykonywania zadań laboratoryjnych każdy student rozwiązywał zadania samodzielnie i tworzył na potrzeby dalszych zajęć kopię wykonanych programów (do wybranych zadań będą nawiązywały dalsze laboratoria).
- W wybranych miejscach umieszczono odnośnik do dokumentacji lub innych źródeł.

Plan zajęć laboratoryjnych:

Lab1.	Wprowadzenie – zapoznanie z IDE, tworzenie prostych programów.
Lab2.	Składnia języka, instrukcje sterujące, typy danych, klasy, metody, obiekty.
Lab3.	Tworzenie aplikacji obiektowych do przetwarzania tekstu.
Lab4.	Tworzenie aplikacji obiektowych wykorzystujących funkcje matematyczne.
Lab5.	Programowanie obiektowe z wykorzystaniem dziedziczenia i interfejsów.
Lab6.	Obsługa wyjątków.
Lab7.	Wykorzystanie kolekcji.
Lab8.	Wyrażenia lambda w języku Java.
Lab9.	Operacje wejścia – wyjścia, obsługa plików.
Lab10.	Współbieżność w języku Java.
Lab11.	Tworzenie aplikacji z graficznym interfejsem użytkownika z obsługą zdarzeń.
Lab12.	Tworzenie aplikacji do komunikacji sieciowej.
Lab13.	Tworzenie aplikacji do obsługi bazy danych.



LABORATORIUM 1. WPROWADZENIE – ZAPOZNANIE Z IDE, TWORZENIE PROSTYCH PROGRAMÓW.

Cel laboratorium:

Przedstawienie podstawowych informacji niezbędnych do instalacji i sprawnego korzystania z IDE. Nauka tworzenia prostych programów.

Zakres tematyczny zajęć:

- Instalacja zintegrowanego środowiska programistycznego (IDE) na przykładzie Apache NetBeans IDE.
- Tworzenie nowego projektu programistycznego.
- Uruchomienie prostej aplikacji.

Pytania kontrolne:

1. Wyjaśnij co oznacza skrót IDE.
2. Opisz co wchodzi w skład JDK.
3. Co musi zawierać dowolny program napisany w języku Java?

Instalacja zintegrowanego środowiska programistycznego

Tworzenie kodu może odbywać się w dowolnym edytorze tekstowym, jednak takie podejście jest nieprofesjonalne, utrudnia proces powstawania aplikacji i wyszukiwania błędów. Z tego względu zdecydowano by podczas laboratorium wykorzystywać jedno z darmowych środowisk programistycznych (ang. Integrated Development Environment, IDE) Apache NetBeans w wersji 12.0. Dodatkowo niezbędne będzie pobranie i instalacja pakietu JDK (ang. Java Development Kit), czyli zestawu narzędzi niezbędnych do tworzenia aplikacji. W skład JDK wchodzi m.in.:

- Kompilator (javac),
- Archiwizator (jar),
- Debugger (jdb).

Poza powyższym zestawem zostanie zainstalowane środowisko uruchomieniowe Java (ang. Java Runtime Environment, JRE) w skład którego wchodzi wirtualna maszyna Javy (ang. Java Virtual Machine, JVM) wraz z podstawowymi klasami.

W tej części zostanie przedstawiony proces instalacyjny dla systemu Windows (środowisko posiada odpowiednie wersje również na systemy Linux oraz macOS).

W celu przeprowadzenia instalacji należy najpierw pobrać pliki dla systemu Windows:

1. OpenJDK 14 (aktualnie najnowsza wersja) - <https://jdk.java.net/14/>

Ciekawostka: Od stycznia 2019 roku firma Oracle wydaje JDK na licencji, która nie zezwala na jakiekolwiek wykorzystywanie komercyjne, bez wykupienia odpowiedniej subskrypcji (od wersji JDK 11). Rozwiązaniem może być wykorzystanie darmowego zestawu narzędzi Open JDK (również od firmy Oracle).

2. Apache NetBeans IDE 12.0 - <https://netbeans.apache.org/download/index.html>



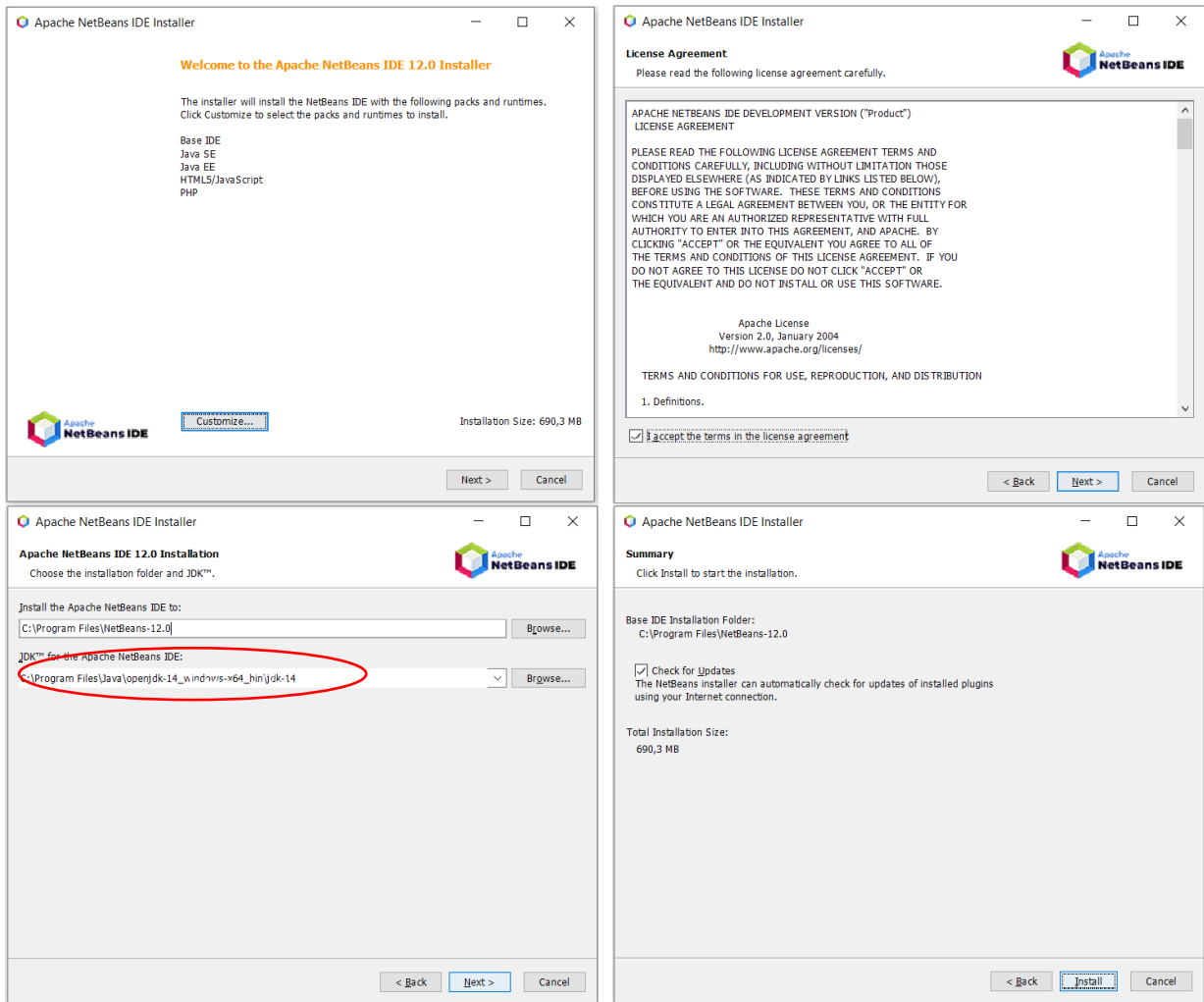
Fundusze Europejskie
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny



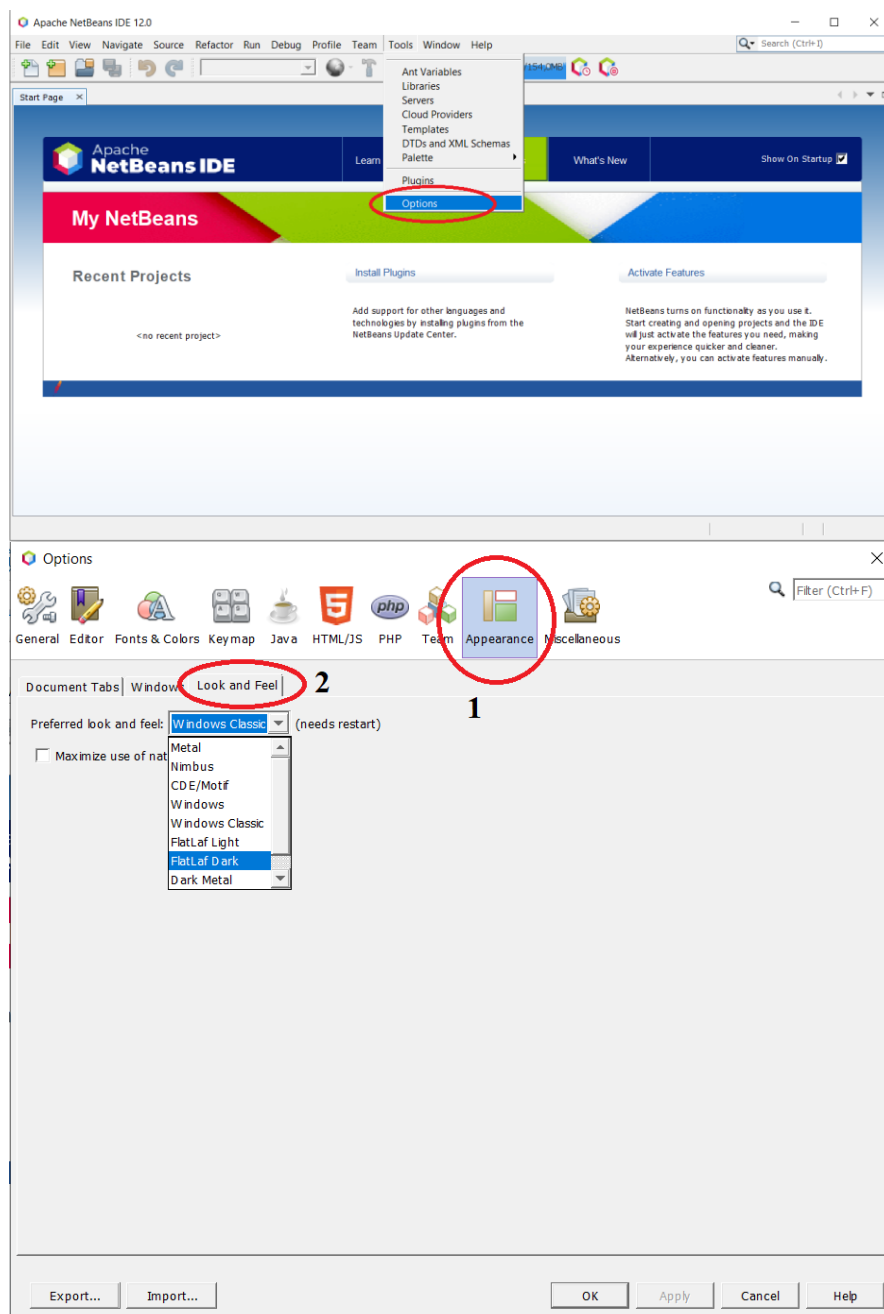


Rys. 1.1. Proces instalacji IDE.

Na rysunku 1.1 zaznaczono ścieżkę do zainstalowanego zestawu JDK, którą należy podać podczas instalacji środowiska.

Konfiguracja środowiska

Po wykonaniu instalacji możliwe jest już tworzenie własnych projektów. Dobrą praktyką jest przeprowadzenie konfiguracji środowiska w celu lepszego dostosowania go do swoich potrzeb. Przykładowo użytkownik może zmienić wygląd IDE zgodnie z rysunkiem 1.2. Wybieramy: Tools → Options → Apperance → Look and Feel. Operacja wymaga przeprowadzenia restartu środowiska.



Rys. 1.2. Konfiguracja wyglądu środowiska

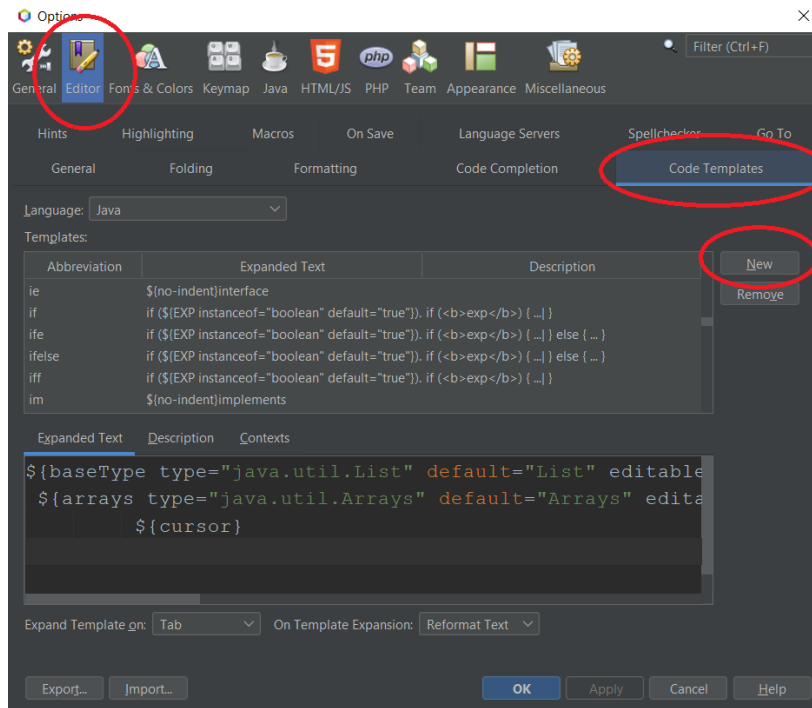
Poza wyglądem wykorzystując środowisko programistyczne warto poznać podstawowe skróty klawiaturowe (tabela 1.1) oraz szablony kodu (tabela 1.2) ułatwiające tworzenie kodu źródłowego.

Tabela 1.1. Wykaz podstawowych skrótów klawiaturowych.

Skrót klawiaturowy	Opis
<i>Alt + Insert</i>	Wstawianie kodu
<i>Alt + Shift + F</i>	Formatowanie kodu
<i>Ctrl + Space</i>	Lista podpowiedzi
<i>Alt + Enter</i>	Wskazówki

<i>Ctrl + Shift + Space</i>	Dokumentacja
<i>Ctrl + F</i>	Wyszukiwanie
F6	Uruchomienie projektu

Po wpisaniu konkretnego szablonu należy użyć klawisza **Tab**, aby kod został wygenerowany. Miejsce wpisania szablonu ma znaczenie.



Rys. 1.3. Dodanie własnego szablonu kodu.

Istnieje możliwość dodania własnych szablonów: Tools → Options → Editor → Code Templates, jak pokazano na rysunku 1.3.

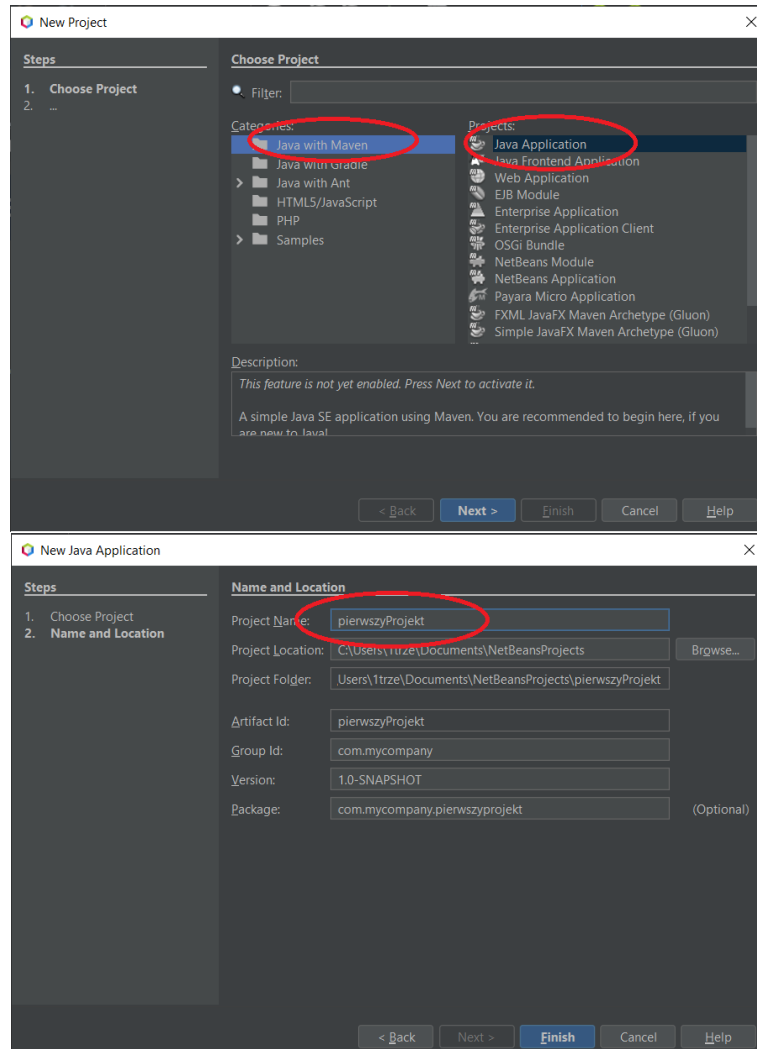
Tabela 1.2. Wykaz podstawowych szablonów kodu.

Szablon kodu	Opis
dowhile	Tworzy pętlę do – while
wh	Tworzy pętlę while
fore	Tworzy pętlę for – each
fori	Tworzy pętlę for
ifelse	Tworzy instrukcję warunkową if – else
trycatch	Tworzy konstrukcję try – catch
psvm	Tworzy metodę main
pu	public
pe	protected
pr	private
psf	private static final
re	return
sout	System.out.println (" ");
soutv	System.out.println("Object = " + Object);



Pierwszy projekt

Po poprawnym zainstalowaniu środowiska i ewentualnej konfiguracji można utworzyć pierwszy projekt. W tym celu należy wybrać: File → New Project → Java with Maven → Java Application. Następnie należy wybrać nazwę dla projektu i kliknąć przycisk Finish, by nowy projekt został utworzony. Na rysunku 1.4 przedstawiono proces tworzenia nowego projektu.



Rys. 1.4. Tworzenie nowego projektu.

Nowy projekt nazwany jako *pierwszyProjekt* został przedstawiony na rysunku 1.5.



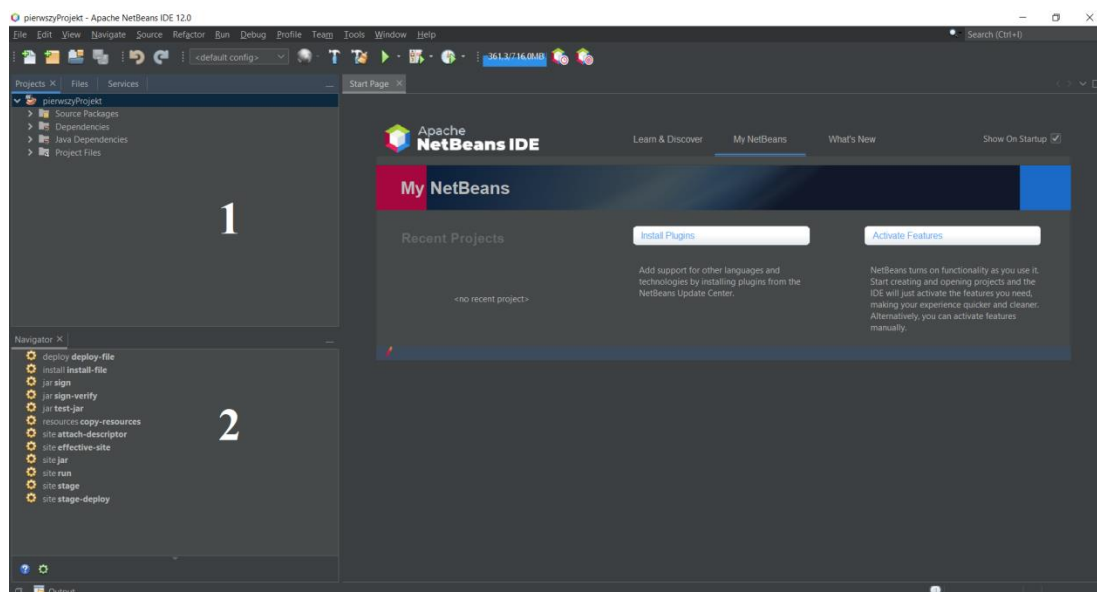
Fundusze Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny



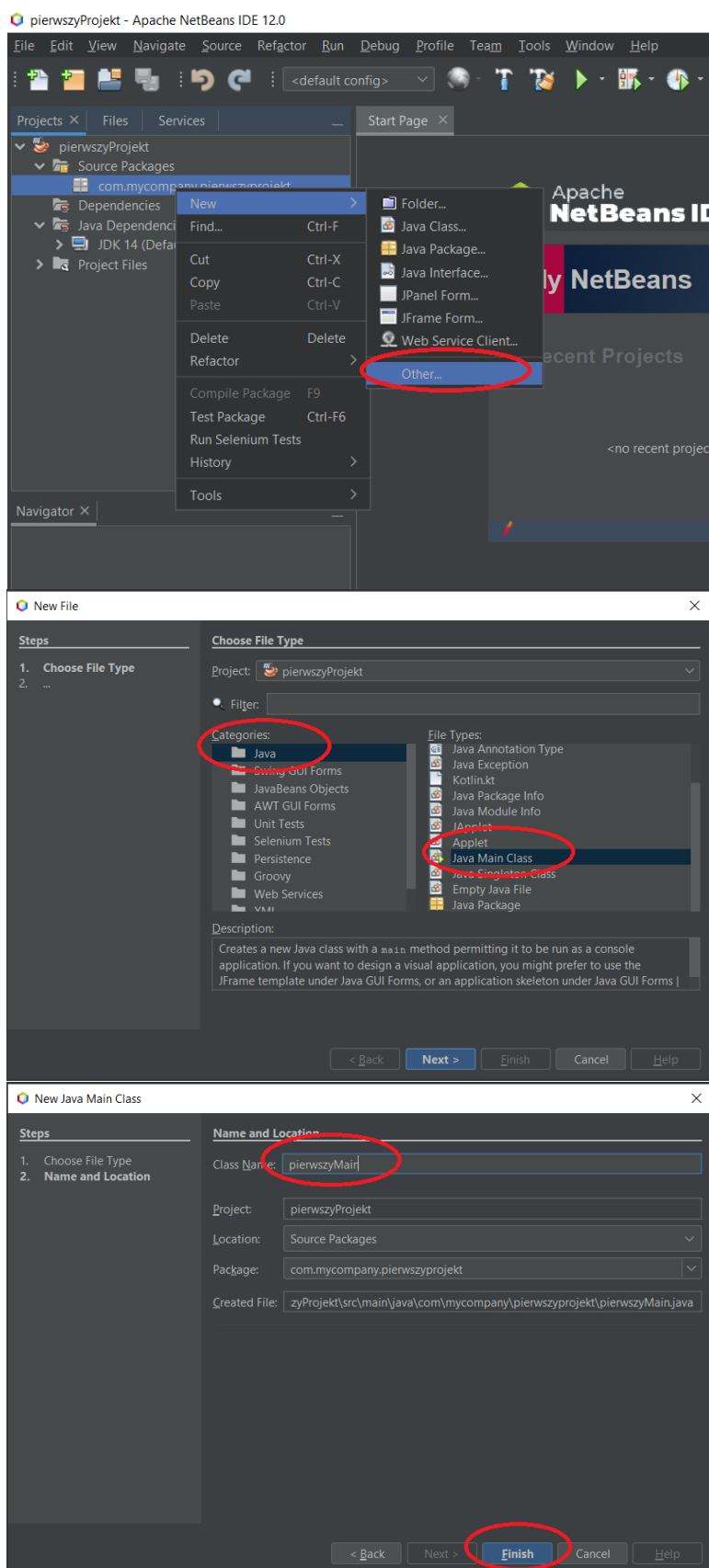


Rys.1.5. Efekt utworzenia nowego projektu.

Po lewej stronie ekranu (1) znajduje się lista projektów wraz z tzw. drzewem projektowym – jest to lista katalogów wraz z zawartością (po rozwinięciu), ułatwia zarządzanie projektem. Poniżej (2) dostępny jest nawigator, który ułatwia orientację w danym pliku wyświetlając listę pól i metod w danej klasie, oraz ewentualną ścieżkę dziedziczenia. Główną częścią IDE jest edytor kodu – poza tworzeniem kodu umożliwia m.in.:

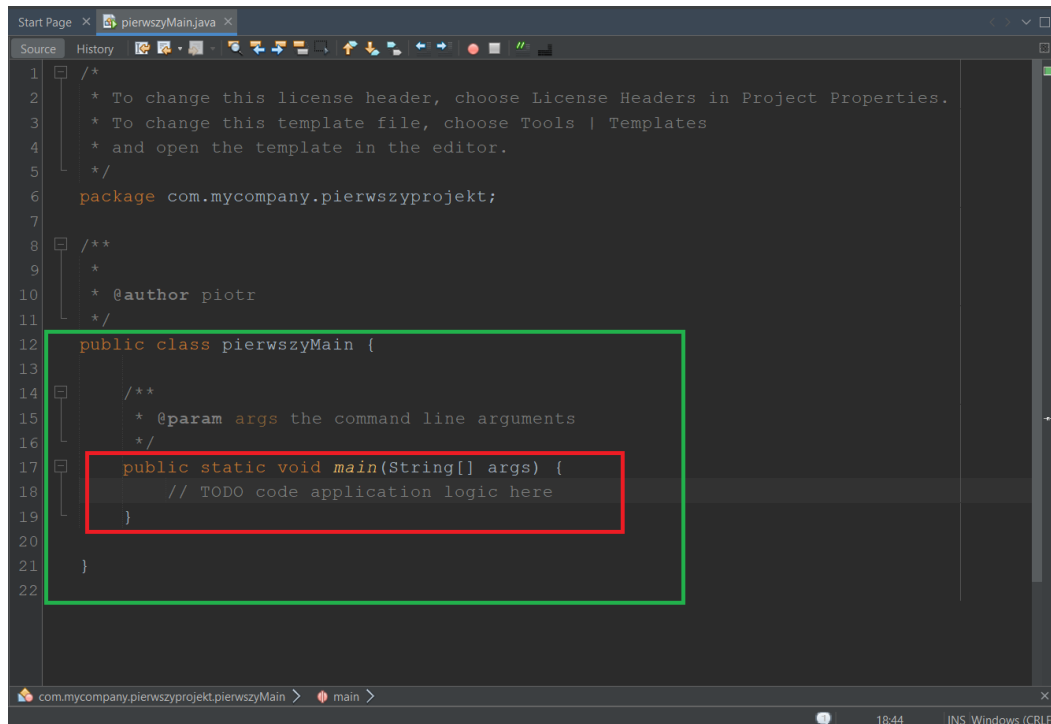
- szybkie formatowanie,
- lokalizację błędów,
- autouzupełnianie kodu,
- automatyczne generowanie kodu,
- wiele innych funkcjonalności.

Kolejnym krokiem jest stworzenie pierwszej klasy zawierającej metodę `main`. Terminy klasa i metoda zostaną rozwinięte w laboratorium 2. W celu utworzenia pierwszej klasy na drzewie projektowym należy rozwinąć katalog *Source Packages*, a następnie kliknąć prawym przyciskiem myszy (PPM) na pakiet *com.mycompany.pierwszyprojekt*, wybrać *New* → *Java Main Class*. Przy pierwszym tworzeniu może nie być tej klasy na liście, wówczas należy wykonać kroki przedstawione na rysunku 1.6. Przy kolejnym tworzeniu nowej klasy ten rodzaj zostanie dodany do listy. W dalszej kolejności wybieramy nazwę dla klasy głównej i przyciskiem *Finish* dodajemy klasę do pakietu. Utworzony zostaje plik o zadanej nazwie z rozszerzeniem **.java*, który można zmieniać przy użyciu edytora kodu.



Rys. 1.6. Tworzenie klasy Main.

Na rysunku 1.7 przedstawiono utworzoną klasę (zaznaczoną na zielono) wraz z metodą main (zaznaczoną na czerwono). Dowolny program w języku Java musi posiadać przynajmniej jedną klasę wraz z metodą main(). Tradycyjnie pierwszy program napisany w danym języku będzie wyświetlał łańcuch znaków np. Hello world!



Rys.1.7. Nowa klasa z metodą main.

Do wyświetlania tekstu można wykorzystać klasę **System** oraz jej pole **out**, które jest standardowym strumieniem wyjściowym. Metod wyświetlających w konsoli tekst jest wiele (zostaną wprowadzone w późniejszych laboratoriach) ale najbardziej podstawową jest **println()**, która wypisuje tekst i przenosi kursor do kolejnego wiersza. Uzupełnij metodę main o poniższy kod:

```
System.out.println("Witaj w świecie Javy!");
```

Ostatnim krokiem jest proces zbudowanie i uruchomienie projektu (można użyć przycisku F6 lub zielonego trójkąta – Run Project). Nastąpi proces kompilacji i z pliku *pierwszyMain.java* powstanie plik kodu bajtowego z instrukcjami dla JVM z rozszerzeniem *.class. Efekt działania programu jest widoczny w dolnej części ekranu (Output – Run (pierwszyProjekt)). Pierwszy projekt został utworzony w oparciu o narzędzie do zarządzania oprogramowaniem (Maven) i poza samym napisem „Witaj w świecie Javy!” w konsoli pojawiają się dodatkowe informacje, które zostaną opisane w dalszej części. Pora wykonać kilka zadań samodzielnie.

Zadanie 1.1. Wyświetlenie informacji

Uzupełnij pierwszy projekt o polecenia wypisania kilku informacji:

- Ile lat uczysz się programowania,

- Ile języków (programowania) znasz na poziomie co najmniej podstawowym – wymień każdy w oddzielnym wierszu,
 - Ulubiony język programowania poznany do tej pory.
- Do wypisania tych informacji możesz użyć metody `println()` lub `print()`.

Zadanie 1.2. Deklaracja zmiennych

Przeanalizuj przykład kodu umieszczonego w metodzie `main`:

```
int x, y;
int z = 12;
double i = 2.45;
char znak = 'z';
double j;
x = 0;
y = 444;
j = 0.002;
System.out.print("x = "+ x +", y = "+ y +", z = "+ z );
```

- Wypisz wszystkie wartości zmiennych nie wyświetlonych w danym fragmencie kodu.
- Dopisz kilka własnych deklaracji.

Zadanie 1.3. Formatowanie wyjścia

Do wyświetlenia wartości w odpowiednim formacie można zastosować metodę ***printf***, w której należy podać łańcuch formatujący dane ze specyfikacją przekształceń. Można wskazać np. na ilu pozycjach ma być wyświetlona wartość zmiennej i z iloma miejscami po kropce (przecinek nie jest separatorem!). Do rozróżnienia typów stosuje się tzw. specyfikację przekształceń rozpoczynającą się znakiem %, po którym umieszcza się odpowiednie formaty:

- %d dla liczb całkowitych,
- %f lub %e dla liczb rzeczywistych,
- %c dla znaku,
- %s dla łańcucha znaków.

```
System.out.printf("\n i=%4.1f, j=%8.2e, x=%6d znak:%c", i, j,
x, znak);
//i będzie wyświetlone na 4 pozycjach z jedną cyfrą po kropce
//j będzie wyświetlone na 8 pozycjach z dwoma cyframi po
kropce w formacie
//zmiennopozycyjnym
//x będzie wyświetlone na 6 pozycjach (spacje z przodu)
//znak będzie wyświetlony jako pojedynczy znak
```

Sprawdź efekt działania metody. Zmodyfikuj sposób wyświetlania danych tak, aby wszystkie wartości całkowite wyświetlane były co najmniej na 5 pozycjach, a rzeczywiste w postaci wykładniczej z 5 miejscami po separatorze.

DODATEK

Apache Maven

Apache Maven jest narzędziem służącym do budowania i zarządzania dowolnym projektem opartym na języku Java. Występuje jako wbudowana część wielu środowisk (np. NetBeans lub IntelliJ). Najważniejsze cechy Maven'a to:

- prosta konfiguracja projektu,
- spójne wykorzystanie w we wszystkich projektach,
- doskonałe zarządzanie zależnościami (w tym automatyczne aktualizowanie i domykanie zależności),
- duże i ciągle rosnące repozytorium bibliotek,
- szybki dostęp do nowych funkcji przy niewielkiej dodatkowej konfiguracji,
- kompilacje oparte na modelach,
- zarządzanie wydaniem i publikacja dystrybucji.

System kontroli wersji

Git to darmowy, rozproszony system kontroli wersji o otwartym kodzie źródłowym, zaprojektowany do obsługi zarówno małych jak i bardzo dużych projektów. Innymi słowy jest to system zapisujący zmiany zachodzące w plikach (tzw. wersje). Programista może przywrócić dowolną poprzednią wersję w razie potrzeby (np. gdy po najnowszych zmianach aplikacja działa błędnie). Równie ważną cechą jest rozgałęzianie, czyli sytuacja kiedy programista wprowadza zmiany w kodzie i jednocześnie chce zachować stabilną wersję projektu – tworzona jest nowa gałąź (ang. branch). Zmiany wprowadzone na nowej gałęzi mogą zostać połączone z główną wersją aplikacji lub też usunięte bez ingerencji w główną gałąź projektu (ang. master branch). Przełączane się pomiędzy gałęziami jest natychmiastowe.

Główne cechy to m.in.:

- śledzenie zmian (przegląd historii kodu źródłowego),
- rozgałęzianie kodu,
- dostęp do poprzednich wersji plików lub całego projektu,
- łączenie modyfikacji kodu wykonanych przez wiele osób w różnym czasie (praca zespołowa poprzez wykorzystanie zdalnych repozytoriów).

W skrócie jest to bardzo rozbudowane narzędzie pozwalające na zarządzanie historią oraz współdzielenie kodu źródłowego. Aby skorzystać z dobrodziejstw Git'a należy pobrać odpowiednią wersję plików instalacyjnych (w zależności od systemu) ze strony <https://git-scm.com> i przeprowadzić proces instalacji. Korzystanie z surowej wersji tego systemu może wydawać się uciążliwe, zwłaszcza jeśli kod tworzony jest przy użyciu IDE. Należy wówczas sprawdzić czy środowisko jest wyposażone w możliwość podłączenia tego narzędzia.

System kontroli wersji jest rozproszony, ponieważ nie istnieje jedno główne repozytorium. Podczas tworzenia wieloosobowego projektu każdy programista posiada lokalne repozytorium, które po wprowadzeniu zmian może zostać połączone ze zdalnym. Istotne informacje takie jak czas wprowadzenia zmian czy autor są zapisywane. Niedopuszczalna jest sytuacja, kiedy dowolna osoba może mieć nieograniczony dostęp do plików projektu, anonimowo wprowadzać zmiany przy braku jakiegokolwiek kontroli ze strony twórców projektu. Zdalne repozytoria są udostępniane przez platformy zajmujące się hostingiem. Najbardziej popularne to:

- GitHub (<https://github.com>),



Fundusze Europejskie
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny



- GitLab (<https://about.gitlab.com>),
- BitBucket (<https://bitbucket.org>).

Oficjalna dokumentacja (częściowo w języku polskim) dostępna jest pod adresem:

<https://git-scm.com/book/pl/v2>

W dalszej części zostanie przedstawiona platforma GitHub oraz sposób połączenia się z nią poprzez Apache Netbeans IDE.

GitHub

GitHub jest platformą do udostępniania projektów, które można kontrolować przy użyciu narzędzia Git. Projekty mogą być udostępniane innym programistom (np. jeśli jest to projekt prowadzony w zespole). Można wykorzystać tę platformę jako hosting repozytoriów i przechowywać swoje projekty w chmurze. Ostatnią popularną funkcją jaką pełni GitHub (i inne podobne platformy) to portfolio – możliwość prezentacji swojej pracy przyszłym pracodawcom lub rekruterom.

Pierwszym krokiem jest założenie konta na: <https://github.com>.

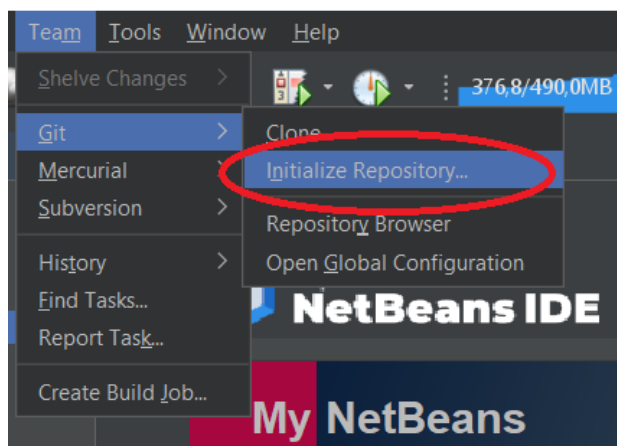
Następnie należy utworzyć nowe repozytorium klikając przycisk oznaczony znakiem „plus” i wybrać **New repository**. Pozostaje wybranie nazwy dla repozytorium, opcjonalnie można umieścić krótki opis oraz określić poziom dostępu:

- public (każdy może zobaczyć zawartość, właściciel decyduje kto może dodawać zawartość),
- private (właściciel określa kto może zobaczyć zawartość lub ją dodawać).

Poniżej przedstawiono krótką instrukcję wykorzystania Git’a i GitHub’a w środowisku Apache NetBeans IDE:

Inicjalizacja repozytorium

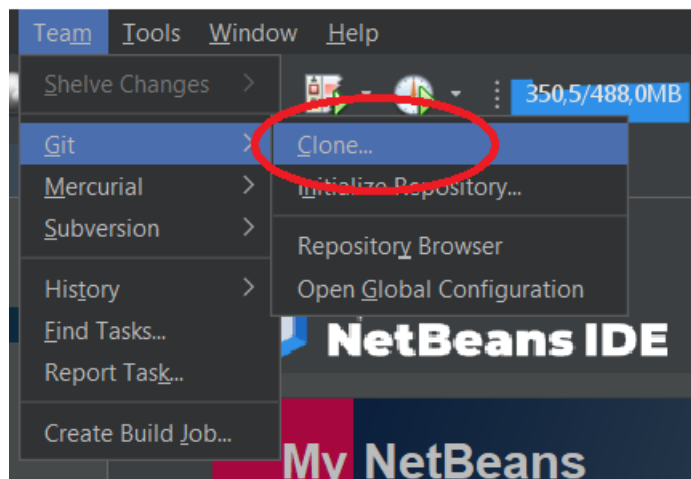
W celu przeprowadzenia inicjalizacji repozytorium należy na drzewie projektowym zaznaczyć wybrany projekt a następnie wybrać zakładkę Team → Git → Initialize Repository (rys.1.8). W kolejnym kroku należy wybrać adres lokalnie przechowywanego repozytorium (można wykorzystać domyślnie nadany).



Rys.1.8. Inicjalizacja repozytorium.

Klonowanie projektu ze zdalnego repozytorium

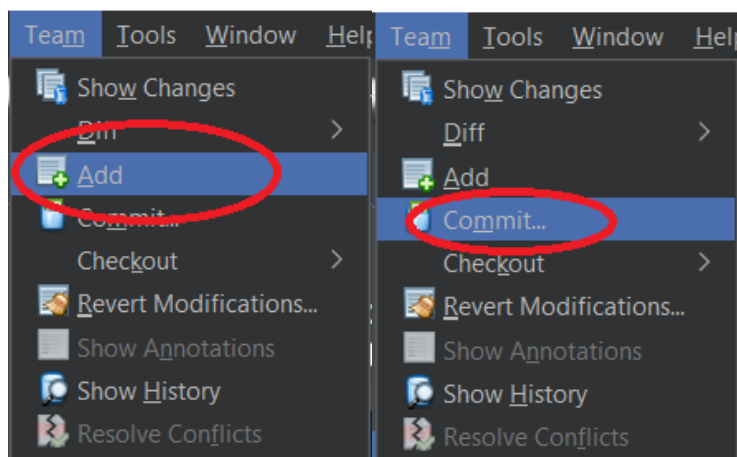
W celu sklonowania projektu ze zdalnego repozytorium, którego nie mamy lokalnie należy wybrać zakładkę Team → Git → Clone (rys.1.9), a następnie podać adres zdalnego repozytorium, dane logowania oraz określić docelowy folder projektu. W następnym kroku należy wybrać gałąź projektu, nadać nazwę klonowanego projektu i zakończyć.



Rys.1.9. Klonowanie zdalnego repozytorium.

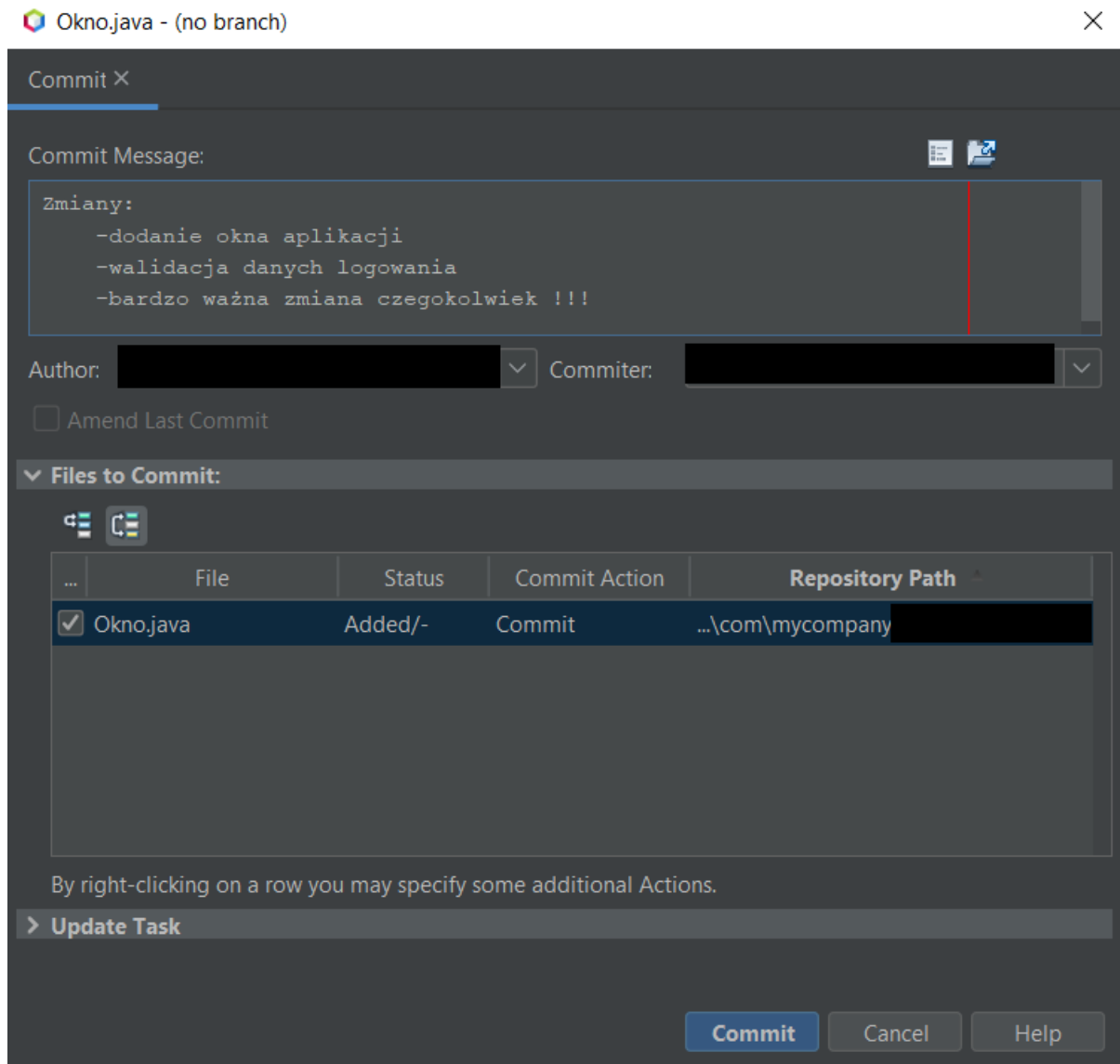
Dodawanie i zatwierdzanie zmian

W celu dodania zmian w repozytorium należy wybrać plik lub projekt w którym nastąpiły zmiany i wybrać Team → Add. Działa tylko dla projektu z zainicjalizowanym repozytorium (rys.1.10).



Rys.1.10. Dodanie/zatwierdzanie zmian do repozytorium.

W celu zatwierdzenia zmian w repozytorium należy wybrać plik lub projekt w którym nastąpiły zmiany i wybrać Team → Commit. Działa tylko dla projektu z zainicjalizowanym repozytorium, o ile nastąpiło dodanie zmian (rys.1.10). W przeciwnym wypadku pojawi się komunikat o braku plików do zatwierdzenia. Przy zatwierdzaniu zmian warto postarać się o merytoryczny komentarz co zostało zrobione/zmienione – ułatwi to późniejszą pracę (rys.1.11).



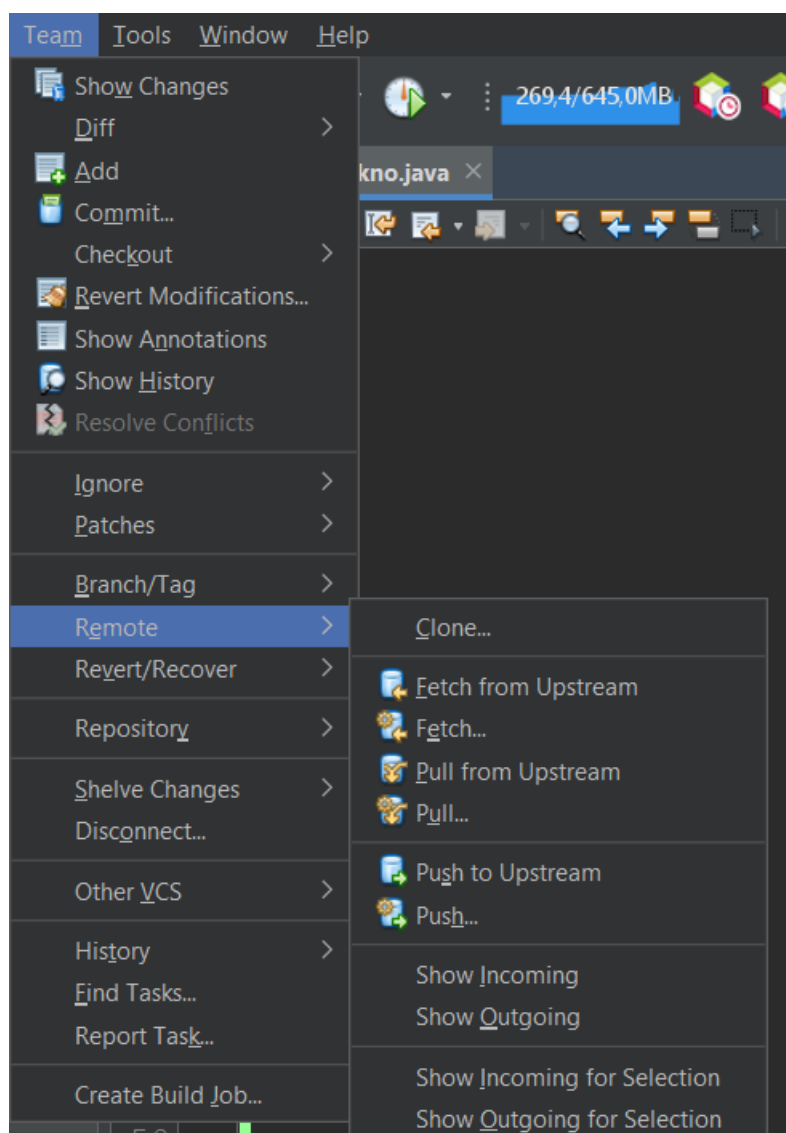
Rys.1.11. Zatwierdzanie zmian.

Wysyłanie zmian do zdalnego repozytorium

Po dodaniu i zatwierdzeniu zmian lokalnie można wysłać je do zdalnego repozytorium. W celu wysłania należy wykorzystać polecenie push dostępne w zakładce Team → Remote → Push (rys.1.12), następnie określić repozytorium i wybrać gałąź. Jeżeli w zdalnym repozytorium nastąpiły zmiany wówczas należy najpierw zaktualizować lokalne repozytorium do ostatniej zatwierdzonej zmiany (commit).

Aktualizowanie zmian lokalnie

W tym celu należy wykonać polecenie pull dostępne w zakładce Team → Remote → Pull (rys.1.12), następnie określić repozytorium i wybrać gałąź. Do pobierania i scalania zdalnych zmian w katalogu roboczym należy wykorzystać polecenia fetch i merge.



Rys.1.12. Polecenia do wysyłania zmian i aktualizowania lokalnego.

Poruszone zagadnienia w żadnym razie nie wyczerpują tematyki pracy ze zdalnym repozytorium ale są zachętą do wykorzystywania tego narzędzia i pogłębiania wiedzy z zakresu współpracy przy tworzeniu kodu.

LABORATORIUM 2. SKŁADNIA JĘZYKA, INSTRUKCJE STERUJĄCE, TYPY DANYCH, KLASY, METODY, OBIEKTY.

Cel laboratorium:

Zapoznanie się ze składnią języka Java, poznanie instrukcji sterujących. Przedstawienie podstawowych typów danych. Definicja klasy, metody i obiektu.

Zakres tematyczny zajęć:

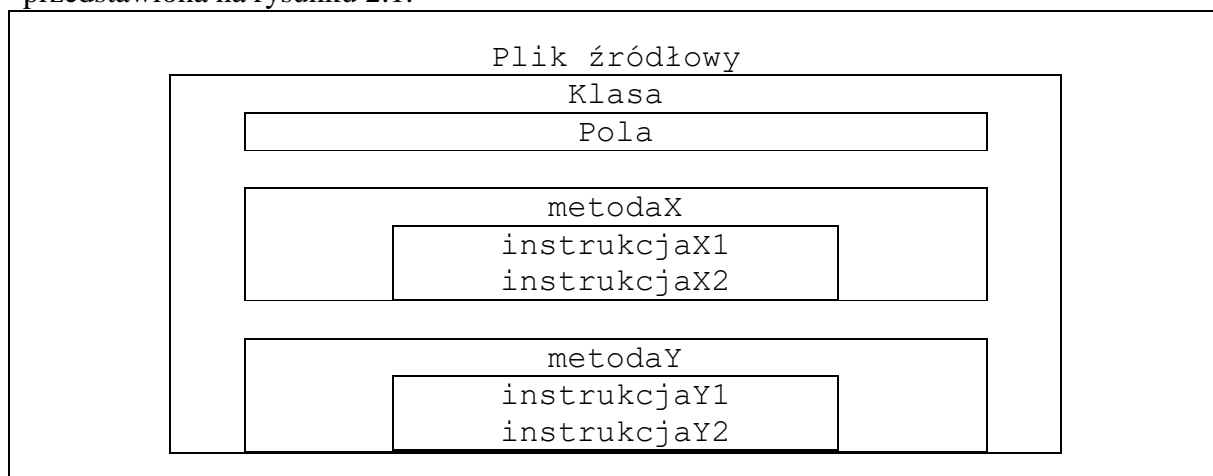
- Składnia języka Java.
- Instrukcje sterujące.
- Typy danych.
- Klasy i składowe klasy.

Pytania kontrolne:

1. Z czego składa się plik źródłowy *.java?
2. Wymień znane instrukcje sterujące. Podaj przykład zastosowania dowolnej instrukcji.
3. Wymień znane typy proste.
4. Zdefiniuj klasę i składowe klasy
5. W jaki sposób tworzy się obiekt?

Składnia języka Java

W języku Java (i nie tylko) kod powstaje w tzw. plikach źródłowych (z rozszerzeniem *.java). Dobra praktyka sugeruje, żeby jeden plik źródłowy składał się z jednej klasy (są uzasadnione przypadki występowania wielu klas w jednym pliku). Cały kod powstaje wewnątrz klas – klasa reprezentuje pewien element programu. Małe aplikacje mogą składać się z jednej klasy. Składowymi klasy są pola (właściwości) oraz metody. Metoda to nazwa dla procedur lub funkcji napisanych w języku Java. W skład każdej metody wchodzi instrukcje określające sposób jej działania. Struktura pojedynczego pliku została przedstawiona na rysunku 2.1.



Rys.2.1. Struktura kodu w języku Java



Po uruchomieniu aplikacji JVM rozpoczyna działanie od odnalezienia danego pliku. Wewnątrz pliku odnajduje specjalną metodę **main** i wykonuje wszystkie instrukcje znajdujące się pomiędzy nawiasami klamrowymi tej metody. Każda aplikacja w języku Java musi zawierać co najmniej jedną klasę i przynajmniej jedną metodę main (w całej aplikacji). Poniżej przedstawiono kod z Laboratorium 1 – przypisy wyjaśniają znaczenie poszczególnych słów kluczowych. Należy zwrócić szczególną uwagę na liczbę nawiasów klamrowych w pliku źródłowym – nawiasów otwierających musi być zawsze tyle samo co nawiasów zamykających (jedno z podstawowych źródeł błędów). Wspomniano już, że cały kod powstaje wewnątrz klasy. Tworzony kod składa się na plik źródłowy, a następnie po poprawnym przejściu procesu kompilacji z pliku źródłowego z rozszerzeniem *.java powstaje plik z rozszerzeniem *.class – uruchamiając program przy pomocy JVM uruchamiamy tak naprawdę klasę.

```
public1 class2 pierwszyMain3 {4

    public5 static6 void7 main8(String[] args)9 {10

        System.out.println("Witaj w świecie Javy!");11

    }12

}13
```

Prawie każda instrukcja w języku Java musi kończyć się średnikiem – „;”. Brak średnika w odpowiednim miejscu generuje błąd podczas kompilacji. W Javie kilka instrukcji może znaleźć się w jednym wierszu i jest to zrozumiałe dla kompilatora (o ile są oddzielone średnikami) – jest to bardzo zła praktyka i nie zaleca się pisać aplikacji jednolinijkowych (patrz skrót kl. Alt + Shift + F). Środowisko w niektórych przypadkach przełamuje wiersz automatycznie.

-
- ¹ modyfikator dostępu klasy – określa poziom widoczności klasy
 - ² słowo kluczowe **class** definiuje klasę
 - ³ nazwa klasy – musi być unikalna (w obszarze swojej „widoczności”)
 - ⁴ klamra otwierająca klasę
 - ⁵ modyfikator dostępu metody – określa poziom widoczności metody
 - ⁶ słowo kluczowe **static** – określa niezależność metody od obiektu danej klasy
 - ⁷ słowo kluczowe void – typ wyniku zwracanego przez metodę (void oznacza, że metoda nie zwraca żadnej wartości)
 - ⁸ nazwa metody – main, musi być unikalna (w obszarze swojej „widoczności”)
 - ⁹ argumenty przekazywane do metody – w tym przypadku jest to tablica łańcuchów znaków o nazwie args
 - ¹⁰ klamra otwierająca metodę
 - ¹¹ instrukcja wewnątrz metody main – domyślne wyświetlenie danych (zadany łańcuch znaków) na standardowym wyjściu (konsola), prawie każdy wiersz w języku Java kończy się średnikiem (;)
 - ¹² klamra zamykająca metodę
 - ¹³ klamra zamykająca klasę



Instrukcje sterujące

W języku Java wszystkie instrukcje sterujące wyglądają bardzo podobnie jak w języku C/C++. Ogólny podział wyróżnia instrukcje wyboru (if – else, switch – case) oraz pętle (for, for – each, while, do – while). Poniżej przedstawiono składnię wszystkich instrukcji wraz z wyjaśnionym przykładem.

Instrukcja warunkowa if – else

Instrukcja if – else jest podstawową metodą wyboru. Występuje w kilku wariantach. Najbardziej ogólna wersja instrukcji if – else wygląda następująco:

```
if (wyrażenie_logiczne) {  
    instrukcjaX;  
    instrukcjaZ;  
}else{  
    instrukcjaY;  
    instrukcjaV;  
}
```

Wyrażenie logiczne (warunek) musi zwracać wartość logiczną (prawda lub fałsz). Jeżeli warunek jest prawdziwy wykonają się instrukcje X oraz Z, jeżeli jest fałszywy analogicznie instrukcje Y oraz V. W przypadku gdy po instrukcja jest krótka (jedna linijka) można zrezygnować z nawiasów klamrowych. Dodatkowo część else nie jest niezbędna (można ją pominąć). Wówczas instrukcja ma postać:

```
if(wyrażenie_logiczne)  
    instrukcjaX;
```

Instrukcja warunkowa może być zagnieżdżona:

```
if(wyrażenie_logiczne){  
    if(inne_wyrażenie_logiczne){  
        instrukcjaABC;  
    }else{  
        instrukcjaBCA;  
    }  
}else{  
    instrukcjaXYZ;  
}
```

lub może występować kilka instrukcji if po sobie – jeżeli jest kilka warunków do sprawdzenia:

```
if(wyrażenie_logiczne){  
    instrukcjaX;  
    instrukcjaY;  
}else if(inny_warunek_logiczny){
```



```

        instrukcjaZ;
        instrukcjaV;
    }else{
        instrukcjaABC;
    }

```

W przypadku, gdy warunek logiczny zostanie spełniony wykonają się instrukcje X oraz Y, jeśli nie będzie spełniony i inny warunek logiczny będzie spełniony wykonają się instrukcje Z oraz V. Jeśli żaden warunek nie będzie spełniony wykona się instrukcja ABC. Zwróć uwagę na zastosowanie średników – każda instrukcja powinna kończyć się średnikiem o ile nie jest instrukcją sterującą.

Do opisu wyrażeń logicznych można stosować operatory relacji oraz logiczne. Operatory relacji dostępne w języku Java zostały przedstawione w tabeli 2.1., a operatory logiczne w tabeli 2.2.

Tab.2.1. Operatory relacji.

Operator relacji	Opis	Przykład
==	Sprawdzenie równości	a == b (a równe b)
!=	Sprawdzenie różności	a != b (a różne od b)
>=	Większe równe	a >= b (a większe równe b)
<=	Mniejsze równe	a <= b (a mniejsze równe b)
>	Większe	a > b (a większe od b)
<	Mniejsze	a < b (a mniejsze od b)

Tab.2.2. Operatory logiczne.

Operator logiczny	Opis	Przykład
&&	Koniunkcja („i”)	(a > 5) && (b < 3) (a większe od 5 i b mniejsze od 3)
	Alternatywa („lub”)	(a < 0) (b > 10) (a mniejsze od 0 lub b większe od 10)

Operator trójargumentowy (lub operator warunkowy) zwraca wartość w przeciwieństwie do tradycyjnej instrukcji if – else. Wygląda następująco:

```
wyrażenie_logiczne ? wartośćA : wartość B
```

Jeżeli wyrażenie logiczne jest prawdziwe zostanie zwrócona wartość A, jeżeli fałszywe wartość B. Może być stosowane np. po słowie kluczowym return w metodzie (znaczenie skraca kod).

Instrukcja switch – case

Instrukcja switch – case jest instrukcją wielokrotnego wyboru i działa następująco: na podstawie wyrażenia całkowitego wybierany jest odpowiedni fragment kodu oznaczony słowem case. Ogólna wersja instrukcji switch – case została przedstawiona poniżej:

```
switch (wyrażenie_całkowite) {
```



Fundusze Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny



```
case wartość_całkowita1: instrukcja1; break;
case wartość_całkowita2: instrukcja2; break;
case wartość_całkowita3: instrukcja3; break;
//...
default: instrukcja_domyślna;
}
```

Wyrażenie całkowite jest porównywane kolejno z wartościami całkowitymi 1, 2, 3, ..., jeżeli w którymś zachodzi równość zostaje wykonana odpowiednia instrukcja, jeżeli żadna wartość nie jest równa wyrażeniu całkowitemu to zostanie wykonana instrukcja domyślna. Break jest opcjonalny i po każdej instrukcji powoduje zakończenie działania switch – case. Program wykonuje dalsze instrukcje zgodnie z kodem. Nieobecność słowa break powoduje dalsze wykonywanie switch – case. Wartością całkowitą może być liczba (int) lub znak (char – każdy znak ma przypisaną całkowitą wartość).

Pętla for

Pętla for jest pętlą licznikową, a to oznacza, że wykona się określoną ilość razy. Z każdym wykonaniem instrukcji wewnątrz pętli zmienia się stan licznika. Instrukcje w pętli for będą wykonywane dopóki wyrażenie logiczne będzie prawdziwe. Należy zawsze wybrać takie wyrażenie, które może zmienić wartość na false, w przeciwnym razie zostanie stworzona pętla nieskończona (istnieją słowa kluczowe pomagające wyjść z pętli). Pętla for wygląda następująco:

```
for(int i = 0; i < 10; i++){

    instrukcja_wewnątrz_pętli;

}
```

Pierwszą wartością w nawiasie jest lokalna zmienna – licznik pętli, po średniku występuje warunek działania pętli (pętla działa dopóki jest prawdziwy), ostatnią częścią jest inkrementacja licznika (w tym przypadku). Warto zauważyć, że licznik nie musi się zwiększać – może np. odliczać od wartości 100 do 0. Może również zmieniać się skokowo np. co 2 – zamiast i++ należy wpisać i = i+2.

Instrukcje wewnątrz pętli będą wykonywane tak długo jak pętla będzie działać. Pętla for wykona się określoną ilość razy (są wyjątki).

Pętla for – each

Pętla for – each jest przeznaczona do przeglądania elementów tablic i kontenerów. W przeciwieństwie do pętli for nie jest wymagana zmienna licznikowa. Przedstawiony przykład wykracza poza tematykę Laboratorium 2, więc wróć tu podczas Laboratorium 4. W przykładzie zadeklarowano tablicę 20 – elementową, gdzie każdy element tablic ma wartość równą indeksowi. Tablica została wypełniona tradycyjną pętlą for (przy nadawaniu wartości należy jawnie podać indeks) ale wyświetlona dzięki pętli for – each.

```
int tablica[] = new int[20];
```



```
for (int i = 0; i < tablica.length; i++) {  
    tablica[i] = i;  
}  
for (int t : tablica) {  
    System.out.println(t);  
}
```

Pętla for each różni się składnią – należy odczytać to następująco dla każdego (całkowitego) t z kontenera tablica wykonaj instrukcje wewnątrz nawiasów klamrowych (wypisz wartość pobraną z tablicy). Typ elementu t zależy od typu tablicy/kontenera.

Pętla while

Pętla while jest pętlą warunkową, gdzie najpierw sprawdzany jest warunek i o ile jest prawdziwy następuje wykonanie instrukcji z ciała pętli. Po wykonaniu instrukcji warunek jest sprawdzany ponownie itd. aż do fałszywego warunku. Pętla while może się nigdy nie wykonać (jeśli początkowo warunek jest fałszywy).

```
while (warunek) {  
    instrukcja_wewnatrz_pętli;  
}
```

Pętla do – while

Pętla do – while działa podobnie do pętli while z jedną różnicą. Warunek w pętli jest sprawdzany po wykonaniu instrukcji więc pętla do – while zawsze wykona się co najmniej raz. Zwróć uwagę na średnik po warunku pętli.

```
do{  
    instrukcja_wewnatrz_pętli;  
}while (warunek);
```

Instrukcja break i continue

Wewnątrz każdej pętli może zdarzyć się sytuacja wymagająca przerwania działania pętli. Instrukcja break przerywa działanie pętli (wyjście z pętli), natomiast instrukcja continue przerywa działanie aktualnie wykonywanej iteracji i rozpoczyna kolejną iterację. Przeanalizuj przykład:

```
for (int i = 0; i < 10; i++) {  
    System.out.println(i);  
    if (i == 3) {  
        break;  
    }  
}
```

Ile wartości zostanie wyświetlonych? Zwróć uwagę na wartość początkową i.



Dostęp do obiektów – referencje, tworzenie, przechowywanie danych

Języki Java i C++ są pod wieloma względami bardzo podobne. Ze względu na to, że C++ jest nadzbiorem języka C, to dozwolone jest używanie podejścia zarówno strukturalnego jak i obiektowego. Programy pisane w języku Java są zorientowane ściśle obiektowo. W każdym języku istnieje określony sposób dostępu do danych np. odwołanie do obiektu bezpośrednio lub pośrednio (wskaźnik w C/C++). Język Java jest prosty i wszystko jest traktowane jako obiekt – dzięki temu występuje jeden spójny zapis używany wszędzie. Dostęp do obiektu uzyskujemy poprzez odwołanie (referencję) do obiektu. Można to wyjaśnić na przykładzie telewizora (obiekt) sterowanego pilotem (referencja). Tak długo jak posiadamy referencję mamy połączenie z telewizorem. Wykonanie czynności tj. przyciszenie czy zmiana programu odbywa się przy użyciu pilota (referencji) – wówczas następuje zmiana stanu telewizora (obiektu). Do kontroli telewizora jest potrzebny wyłącznie pilot – referencja. Referencja może również istnieć bez przypisanego do niej obiektu (pilot bez telewizora). Do przechowania zdania można posłużyć się referencją typu String:

```
String zdanie;
```

Tak stworzona referencja nie jest przypisana do żadnego obiektu – przesłanie wiadomości zakończyłoby się komunikatem o błędzie (brak telewizora) dlatego dla bezpieczeństwa lepiej zainicjalizować referencję przy tworzeniu:

```
String zdanie = "Ala ma kota na punkcie Javy";
```

Podany przykład jest bardzo specyficzny (czyt. wyjątkowy) – łańcuch znaków można zainicjować podając tekst w cudzysłowie – tworząc „tradycyjny” obiekt należy użyć ogólnego sposobu inicjalizacji.

Tradycyjne tworzenie obiektów

Tworząc referencję należy powiązać ją z obiektem – normalna droga to użycie słowa kluczowego **new**. Mówi ono, że należy stworzyć nowy obiekt danego typu – rozbudowując poprzedni przykład:

```
String zdanie = new String("Ala ma kota na punkcie Javy");
```

Oznacza to, że poza poleceniem utworzenia nowego łańcucha znaków podawany jest sposób utworzenia (na podstawie przekazanego łańcucha znaków). Typ String nie jest jedynym istniejącym w języku Java – dostępnych jest wiele gotowych typów, jednak najważniejsza jest możliwość tworzenia własnych typów – klas.

Przechowywanie danych

Przydatne informacje dotyczące miejsca umieszczania danych są kluczowe dla lepszego zrozumienia podstaw obiektowości. Istnieje pięć miejsc do przechowywania danych:

1. Rejestry – najszybciej dostępna pamięć (wewnątrz procesora). Liczba rejestrów jest bardzo ograniczona. Programista nie ma bezpośredniej kontroli nad tym obszarem i co najważniejsze z poziomu programu nie jest w stanie udowodnić, że rejestry w ogóle istnieją.
2. Stos – umieszczany w obszarze pamięci RAM oraz bezpośrednio obsługiwany przez procesor. Wskaźnik stosu przesuwany w dół w celu zajęcia nowego obszaru pamięci lub w górę w celu zwolnienia pamięci. Szybki i efektywny sposób przydzielania pamięci, zaraz po rejestrach. Kompilator Javy musi znać dokładny rozmiar i czas życia wszystkich danych przechowywanych na stosie podczas tworzenia programu (musi wygenerować kod odpowiedzialny za przesuwanie wskaźnika stosu) i z tego powodu część danych jest przydzielana do pamięci stosu – w szczególności są to referencje do obiektów – obiekty nie są tam umieszczane.
3. Sterta – fragment pamięci ogólnego zastosowania. Tu przechowywane są wszystkie obiekty Javy – kompilator nie potrzebuje informacji o niezbędnym rozmiarze ani czasie zajmowania pamięci. Przy użyciu słowa kluczowego **new** tworzony jest obiekt i przydzielane jest miejsce w pamięci sterty, jednak przydzielenie pamięci sterty zajmuje więcej czasu niż w przypadku stosu.
4. Obszar stałych – stałe umieszczane są w kodzie programu, co chroni przed zmianami.
5. Obszar spoza RAM – dane umieszczone poza programem mogą istnieć również jeśli program nie jest uruchomiony. Przykładem może być strumieniowanie obiektów (obiekt przekształcany jest na strumień bajtów w celu przesłania np. na inną maszynę) lub obiekt trwały (umieszczany na dysku, zachowuje swój stan nawet po zamknięciu programu).

Typy proste

W języku Java występuje specjalna grupa typów – typy proste (podstawowe). Stworzenie obiektu będącego prostą zmienną nie jest wydajne, ponieważ operator **new** przydziela pamięć na stercie. Wykorzystano rozwiązanie zaczerpnięte z języków C/C++ i zamiast tworzyć nowy obiekt tworzona jest zmienna „automatyczna”, która nie jest referencją – przechowuje wartość i jest umieszczana na stosie co jest wydajniejsze.

W Javie istnieje 8 typów prostych – każdy z nich ma określony rozmiar i jest on niezmienny oraz niezależny od architektury sprzętowej. W tabeli 2.3 przedstawiono typy proste. Każdemu z nich odpowiada typ obiektowy – tzw. klasa opakowująca. Rozmiar typu boolean nie jest określony – dopuszcza dwa literały: true (prawda) i false (fałsz).

Tabela 2.3. Typy proste w języku Java.

Nazwa typu	Rozmiar	Wartość min.	Wartość max.	Typ obiektowy
boolean	–	–	–	Boolean
char	16 bitów	Unicode 0	Unicode $2^{16} - 1$	Character
byte	8 bitów	-128	+127	Byte
short	16 bitów	-2^{15}	$+2^{15} - 1$	Short
int	32 bity	-2^{31}	$+2^{31} - 1$	Integer
long	64 bity	-2^{63}	$+2^{63} - 1$	Long



float	32 bity	IEEE754	IEEE754	Float
double	64 bity	IEEE754	IEEE754	Double

Typ obiektowy umożliwia tworzenie na stercie obiektów reprezentujących typ prosty np.:

```
char c = 'a';  
Character ch = new Character(c);
```

ewentualnie jawnie pomijając typ prosty (char):

```
Character ch = new Character('a');
```

Własne typy danych – klasy

Klasa jest szablonem do tworzenia obiektów – przekazuje informacje JVM jak należy utworzyć obiekt konkretnego typu. Każdy obiekt utworzony na podstawie klasy może mieć unikalne wartości składowe. Programowanie obiektowe opiera się w głównej mierze na tworzeniu własnych typów danych. Przykładem może być klasa Owoc przy użyciu której można stworzyć dowolną liczbę obiektów – każdy o innej nazwie, kolorze, masie, kształcie, kraju pochodzenia itd.

```
public class Owoc {  
  
    String nazwa;  
    Color kolor;  
    double masa;  
    String krajPochodzenia;  
  
    public Owoc() {  
    }  
  
    public Owoc(String nazwa, String krajPochodzenia) {  
        this.nazwa = nazwa;  
        this.krajPochodzenia = krajPochodzenia;  
    }  
  
    public void podajMase(double masaZmierzona) {  
        masa = masaZmierzona;  
    }  
}
```

Klasa definiowana jest przy użyciu słowa kluczowego class. Wewnątrz klasy mogą znajdować się jej składowe tj. pola i metody. Pola (właściwości) opisują dany obiekt. Metody (inaczej sposób na wykonanie czegoś, w języku C – funkcja) to wydzielone fragmenty kodu spełniające zadania np. wyświetlenie wartości, wykonanie obliczeń itp.



Pola i metody

W podanym przykładzie klasa Owoc posiada cztery pola tj. *nazwę*, *kolor*, *masę* i *krajPochodzenia*. Każda pojedyncza reprezentacja tej klasy (obiekt) będzie opisywana przy pomocy tak zdefiniowanych pól. Mogą być obiektem dowolnego typu (również typu podstawowego). Dostęp do pól (o ile jest możliwy) jest możliwy przy użyciu operatora kropki:

```
nazwaObiektu.nazwaPola;
```

np.

```
Owoc banan = new Owoc();  
System.out.println(banan.krajPochodzenia);
```

W klasie Owoc występuje tylko jedna metoda – *podajMase()*. Metoda wykorzystuje podany parametr i ustawia wartość pola. Do wywołania metody stosuje się operator kropki:

```
nazwaObiektu.nazwaMetody(parametr1,parametr2,...);
```

np.

```
banan.podajMase(192.168);
```

Należy zauważyć, że przy podawaniu listy parametrów są one oddzielone przecinkami. Separatorem w przypadku liczby rzeczywistej jest więc kropka, a nie przecinek.

Metody w wyniku swojego działania mogą zwracać wartości (nie muszą). Jeżeli metoda ma zwrócić wartość to należy określić typ jaki ma zwracać w deklaracji metody. Wówczas na końcu metody występuje słowo kluczowe return po którym należy wypisać zwracaną wartość. Jeśli metoda nic nie zwraca jest typu void i słowo return nie występuje. Przykłady przedstawiono poniżej:

```
public void metodaNieZwraca(int a){  
    System.out.println(a);  
}
```

```
public int metodaZwraca(int a){  
    return a*a;  
}
```

Metoda, która zwraca wartość może być parametrem dla innej metody. Lista parametrów nie jest konieczna, jeżeli metoda nie przyjmuje żadnych parametrów należy zostawić pusty nawias po nazwie:

```
public void nazwaMetody(){  
    //...  
}
```



Konstruktor

Tworząc obiekt klasy (instancję, pojedynczy egzemplarz) niezbędny jest konstruktor. Konstruktor ma zawsze taką samą nazwę jak nazwa klasy. W klasie *Owoc* zdefiniowano dwa konstruktory (jeden bez parametrów – *Owoc()*, a drugi z parametrami – *Owoc(String nazwa, String krajPochodzenia)*). Nie ma konieczności tworzenia konstruktora, wówczas do tworzenia obiektu zostanie użyty konstruktor domyślny (nie można wykorzystać konstruktora domyślnego jeżeli w klasie zdefiniowano konstruktor z parametrami). Konstruktor z parametrami umożliwia inicjalizację pól obiektu już na etapie tworzenia.

Wartości domyślne

Jeżeli pole jest typu podstawowego i nie ma zainicjalizowanej wartości w definicji klasy ani w konstruktorze wówczas zostanie zainicjowane z wartością domyślną – tabela 2.4. Nie dotyczy to zmiennych lokalnych (np. w metodach) – wówczas niezainicjalizowana zmienna ma wartość nieokreśloną.

Tab.2.4. Wartości domyślne pól typów prostych.

Nazwa typu	Wartość domyślna
boolean	false
char	'\u0000'
byte	(byte)0
short	(short)0
int	0
long	0L
float	0.0f
double	0.0d

Zadanie 2.1. Inicjalizacja

Stwórz klasę składającą się z niezainicjalizowanych pól typu int, char, String. Stwórz obiekt klasy, a następnie wypisz wartości pól. Jakie wartości zostały nadane?

Zadanie 2.2. Metoda statyczna

Standardowe metody są powiązane z obiektami (aby wywołać metodę niezbędny jest obiekt). Wykorzystując słowo kluczowe static napisz metodę wypisującą wartości 0 – 54 (każda wartość w nowym wierszu). Wywołaj metodę nie używając obiektu.

Zadanie 2.3 Klasa Uczeń

Stwórz nową klasę – Uczeń, zadeklaruj niezainicjalizowane pola:

- klasy String – imię i nazwisko
- typu int – wiek
- typu double – średnia

Dodaj konstruktor bezparametrowy i parametrowy ustawiający wszystkie pola klasy. W metodzie main stwórz kilka obiektów używając obu konstruktorów. Wyświetl dane obiektów (nazwisko, imię, wiek, średnia) każdy obiekt w jednym wierszu.



Zadanie 2.4 Klasa Owoc

Bazując na klasie przedstawionej w Laboratorium 2 napisz własną klasę Owoc. Dodaj kilka metod zwracających wartość (np. nazwę i masę) i wykorzystaj je do wypisania wartości pól konkretnych obiektów.

Zadanie 2.5 Odliczanie w pętli

Napisz kilka wariantów metody odliczającej w pętli od 5 do 80 co 15 wypisując te wartości w jednym wierszu (oddziel wartości spacją). Wykorzystaj wszystkie rodzaje pętli.

Zadanie 2.6 Trójkąt

Napisz metodę przyjmującą trzy parametry całkowite i sprawdzającą czy możliwe jest zbudowanie trójkąta o danych bokach. Sprawdź dodatkowo czy będzie to trójkąt równoboczny, równoramienny czy różnoboczny.

Zadanie 2.7. Liczby całkowite

Napisz metodę która wyświetli wszystkie liczby całkowite z zakresu $<11,111>$ podzielne bez reszty przez 13.

Zadanie 2.8. Ciąg Fibonacciego

Napisz metodę przyjmującą jeden parametr całkowity dodatni – n. Metoda ma wyświetlać n – elementów z ciągu Fibonacciego (w jednym wierszu, każdy element oddzielony spacją).

LABORATORIUM 3. TWORZENIE APLIKACJI OBIEKTOWYCH DO PRZETWARZANIA TEKSTU.

Cel laboratorium:

Zapoznanie z klasą String oraz z operacjami przeprowadzanymi na tekście. Napisanie aplikacji przetwarzających tekst.

Zakres tematyczny zajęć:

- Klasa String.
- Konwersja łańcucha znaków.
- Odczyt z konsoli.

Pytania kontrolne:

1. W jaki sposób można stworzyć łańcuch znaków?
2. W jaki sposób zamienić łańcuch znaków na typ liczbowy?
3. W jaki sposób zamienić typ liczbowy na łańcuch znaków?
4. W jaki sposób odczytać dane wprowadzone przez użytkownika?

Klasa String – jako tablica znaków

Klasa *String* reprezentuje ciąg znaków – wszystkie literały łańcuchowe w języku Java np. „ABC” są implementowane jako instancja tej klasy. Ciągi znaków są stałe tzn. po utworzeniu ich wartości nie mogą się zmienić. Obiekty klasy String są niezmiennicze (ang. immutable) – każda z metod tej klasy zwraca zupełnie nowy obiekt (pierwotny egzemplarz pozostaje nietknięty). Mogą więc być współużytkowane, np.:

```
String ciag = "abc";
```

jest równoważny:

```
char dane[] = {'a','b','c'};  
String ciag = new String(dane);
```

Klasa String zawiera metody do badania poszczególnych sekwencji znaków, porównywania łańcuchów, wyszukiwania łańcuchów, wyodrębniania podciągów oraz tworzenia kopii łańcucha z translacją (np. na wielkie lub małe litery). Mapowanie wielkości liter jest oparte na wersji standardu Unicode określonej przez klasę Character. W języku Java możliwa jest konkatenacja (łączenie) łańcuchów znaków (przy użyciu przeciążonego operatora +) oraz konwersja innych obiektów na łańcuch znaków.

Zamiana liczby na łańcuch znaków

Konwersja liczby na łańcuch znaków może odbywać się na kilka sposobów, w zależności od typu liczby. Przykładowo dla liczby całkowitej przechowywanej w obiekcie klasy Integer można:



1. wykorzystać metodę statyczną: ***Integer.toString***,
2. wykorzystać metodę statyczną: ***String.valueOf***,
3. połączyć liczbę z pustym ciągiem znaków z użyciem operatora konkatencji (+).

Odpowiednio dla innych typów obiektowych lub prostych można wykorzystać analogiczne metody znajdujące się w odpowiednich klasach.

Zamiana łańcucha znaków na liczbę

Zamiana łańcucha znaków na liczbę niesie ze sobą ryzyko, że poza cyframi (i ewentualnym separatorem) w danym ciągu mogą znaleźć się inne znaki. Wówczas zamiana na typ liczbowy zakończy się wyrzuceniem wyjątku *NumberFormatException* – obsłudze wyjątków poświęcone jest Laboratorium 6. W przypadku takiej konwersji jest również kilka sposobów przedstawionych poniżej na przykładzie zamiany na liczbę całkowitą:

1. wykorzystanie metody statycznej ***Integer.valueOf*** – metoda zwraca obiekt klasy *Integer*,
2. wykorzystanie metody statycznej ***Integer.decode*** – metoda zwraca obiekt klasy *Integer*,
3. wykorzystanie metody statycznej ***Integer.parseInt*** – metoda zwraca typ prosty *int*.

Szczegółowa obsługa wyjątków zostanie omówiona w dalszej części – w tym momencie należy zapamiętać, z jakiego powodu program zamieniający łańcuch znaków na liczbę może nie działać.

Metody klasy *String*

W klasie *String* występują metody bardzo pomocne przy operowaniu na łańcuchach znaków. Poniżej przedstawiono podstawowe zestawienie – więcej szczegółów na temat podanych metod można odnaleźć w dokumentacji.

Tab.3.1. Wybrane metody klasy *String*

Modyfikator i typ	Metoda	Opis
char	<i>charAt(int index)</i>	Zwraca znak o określonym indeksie
int	<i>codePointAt(int index)</i>	Zwraca kod znaku o określonym indeksie
int	<i>compareTo(String s)</i>	Porównuje dwa ciągi leksykograficznie
boolean	<i>contains(CharSequence s)</i>	Zwraca prawdę wtedy i tylko wtedy, gdy ciąg zawiera określoną sekwencję znaków
boolean	<i>equals(Object o)</i>	Porównuje ciąg z określonym obiektem
static String	<i>format(String format, Object ...args)</i>	Zwraca sformatowany ciąg przy użyciu określonego ciągu formatu i argumentów
int	<i>indexOf(int ch)</i>	Zwraca indeks pierwszego wystąpienia określonego znaku w danym ciągu
boolean	<i>isBlank()</i>	Zwraca prawdę jeżeli ciąg jest pusty lub zawiera białe znaki, w przeciwnym wypadku zwraca fałsz
int	<i>length()</i>	Zwraca długość danego ciągu
boolean	<i>matches(String regex)</i>	Informuje, czy ciąg pasuje do podanego wyrażenia regularnego



String	<i>replace(char old, char new)</i>	Zwraca ciąg będący wynikiem zastąpienia wszystkich znaków „old” na „new”
String	<i>replaceAll(String regex, String replacement)</i>	Zastępuje każdy podciąg pasujący do podanego wyrażenia regularnego na podany zamiennik
String []	<i>split(String regex)</i>	Dzieli ciąg wykorzystując podane wyrażenie regularne
String	<i>strip()</i>	Zwraca ciąg z usuniętymi wszystkimi początkowymi i końcowymi białymi znakami
String	<i>substring(int begin, int end)</i>	Zwraca podciąg z danego ciągu
char []	<i>toCharArray()</i>	Konwertuje ciąg na nową tablicę znaków
String	<i>toLowerCase()</i>	Konwertuje wszystkie znaki w ciągu na małe litery
String	<i>toUpperCase()</i>	Konwertuje wszystkie znaki w ciągu na wielkie litery
static String	<i>valueOf(int i)</i>	Zwraca ciąg znaków reprezentujących argument

Odczyt wprowadzonego ciągu znaków – klasa Scanner

Klasa *Scanner* służy do odczytu i analizy prostych tekstów z podziałem na typy podstawowe i ciągi znaków przy użyciu wyrażeń regularnych. Dane wejściowe są dzielone na tokeny przy zastosowaniu wzorca ogranicznika (domyślnie białe znaki). Otrzymane tokeny można przekonwertować na wartości różnych typów wykorzystując dostępne w klasie metody. Przykładowe odczytanie liczby całkowitej wygląda następująco:

```
Scanner sc = new Scanner(System.in);
int i = sc.nextInt();
```

Znajdujące się w klasie *Scanner* metody są bardzo pomocne w odczycie konkretnych typów danych np. ***nextInt()*** lub ***nextDouble()*** zwraca kolejny token jako liczbę całkowitą lub zmiennoprzecinkową. Należy zachować szczególną ostrożność – jeśli token nie może zostać dopasowany do wyrażenia regularnego zdefiniowanego jako liczba całkowita (lub odpowiednio zmiennoprzecinkowa) to metoda wyrzuci wyjątek *InputMismatchException*. Najprostszym rozwiązaniem poza klasyczną obsługą wyjątków przedstawioną w Laboratorium 6 jest wykorzystanie metod ***hasNextInt*** lub ***hasNextDouble*** które zwracają prawdę jeśli następnym tokenem jest odpowiednio liczba całkowita lub zmiennoprzecinkowa.

W klasie *Scanner* wartą uwagi metodą jest ***hasNext*** zwracająca prawdę jeśli kolejny token oczekuje na wejściu. W połączeniu np. z pętlą *while* można wykorzystać tę metodę do ciągłego odczytywania wejścia (np. konsoli) dopóki użytkownik nie wprowadzi określonego wyrażenia.

Zadanie 3.1. Obecność w łańcuchu

Napisz metodę przyjmującą jako parametr znak (*char*) i łańcuch znaków. Metoda ma sprawdzać czy i ile razy w podanym łańcuchu występuje dany znak.



Zadanie 3.2. Suma ASCII

Napisz metodę przyjmującą jako parametr łańcuch znaków i sumującą wartości kodów ASCII wyłącznie z małych liter i cyfr w podanym parametrze.

Zadanie 3.3. Liczba na znak

Napisz metodę odczytującą z konsoli (od użytkownika) liczbę całkowitą z zakresu <33,126>. Metoda ma zwracać znak przypisany do danej liczby. Załóż, że użytkownik może wprowadzić wyłącznie cyfry.

Zadanie 3.4. Szyfr Cezara

Napisz aplikację zawierającą metody:

- odczytującą tekst wprowadzony przez użytkownika i zwracającą go jako łańcuch znaków,
- pobierającą argument klasy String, a następnie szyfrujący go przy użyciu szyfru Cezara. Metoda ma zwracać zaszyfrowany łańcuch znaków. Zadbaj o możliwość szyfrowania wyłącznie liter,
- pobierającą argument klasy String, a następnie deszyfrujący go zgodnie z szyfrem Cezara. Metoda ma zwracać odszyfrowany łańcuch znaków. Deszyfruj wyłącznie litery,
- pobierającą argument klasy String, a następnie sprawdzającą czy jest on palindromem. Metoda ma zwracać prawdę lub fałsz.

Znajomość dokładnych pozycji z tabeli kodów ASCII nie jest konieczna. Wykorzystaj napisane metody i sprawdź działanie programu.

Zadanie 3.5. Binarna zerówka

Napisz aplikację zawierającą metody:

- pobierającą argument wejściowy typu int i zwracającą ciąg znaków – binarną reprezentację podanej liczby,
- pobierającą argument wejściowy klasy String – liczbę w reprezentacji binarnej oraz zwracającą całkowitą liczbę sekwencji zer w postaci łańcucha znaków. Należy założyć, że dany łańcuch znaków składa się wyłącznie cyfry 0 i 1. W przypadku braku sekwencji zer należy zwrócić komunikat. Sekwencję zer należy rozumieć jako ciąg zer oddzielony obustronnie jedynekami. Przykładowo liczba 556 -> 1000101100 posiada dwie sekwencje zer z czego najdłuższa posiada trzy zera i metoda powinna zwrócić wartość „3”. Ostatnie dwa zera nie są brane pod uwagę, ponieważ nie są obustronnie oddzielone jedynekami.

Wykorzystując powyższe metody wyświetl zadaną liczbę, jej reprezentację binarną oraz największą sekwencję zer (o ile istnieje). W klasie **Integer** znajdziesz interesującą metodę.



LABORATORIUM 4. TWORZENIE APLIKACJI OBIEKTOWYCH WYKORZYSTUJĄCYCH FUNKCJE MATEMATYCZNE.

Cel laboratorium:

Wprowadzenie do tablic. Zapoznanie z klasą Math i wykorzystanie podstawowych funkcji matematycznych w programie.

Zakres tematyczny zajęć:

- Tablice jedno i wielowymiarowe.
- Klasa Math i operacje matematyczne.
- Klasa Random.

Pytania kontrolne:

1. W jakim celu stosuje się tablice?
2. Jak wygląda deklaracja tablicy jednowymiarowej, a jak wielowymiarowej?
3. Wymień i opisz podstawowe metody z klasy Math.
4. W jaki sposób wylosować liczbę z zakresu $<-12,24>$?
5. Czy można wylosować dwa razy taką samą liczbę?

Teoria o tablicach

Tablica to zbiór uporządkowanych danych tego samego typu, w którym do poszczególnych elementów (komórek tablicy) można odwołać się przy użyciu indeksu. Indeksy w języku Java (jak w wielu innych językach) są liczbami całkowitymi – pierwszym indeksem jest zawsze 0. Rozmiar (ilość elementów) tablicy musi być określony. W tablicach można przechowywać dane dowolnego typu, również obiekty dowolnej klasy – tworząc tablicę obiektów tworzona jest tak naprawdę tablica referencji do obiektów i każda z nich automatycznie ustawiana jest na wartość oznaczoną słowem kluczowym **null**. Gdy kompilator widzi taką referencję (null) rozpoznaje, że nie wskazuje ona obiektu – przed odwołaniem należy przypisać do referencji obiekt, w przeciwnym razie pojawi się komunikat – zabezpieczenie przed typowym błędem odwołania do tablicy. Kolejnym zabezpieczeniem jest brak możliwości użycia tablicy bez wcześniejszej inicjalizacji – początkowy błąd programistów.

W języku Java tablica jest najbardziej wydajnym sposobem zapisu i swobodnego dostępu do sekwencji obiektów (dokładniej referencji do obiektów). Jak wspomniano wyżej rozmiar tablicy musi być ustalony podczas tworzenia i nie może być zmieniony w czasie życia tablicy. Bez względu na to jakiego typu tablicę zastosujemy, identyfikator tablicy jest w rzeczywistości odwołaniem do prawdziwego obiektu (stworzonego na stercie). Ten obiekt przechowuje odwołania do innych obiektów i może być stworzony albo pośrednio jako skutek inicjalizacji tablicy albo bezpośrednio z użyciem operatora **new**. Wyjątkiem może być tutaj tablica typów podstawowych, która przechowuje bezpośrednio wartości.

Podobnie jak inne obiekty, tablica posiada pola i metody – jedynym publicznym polem jest **length** i przechowuje informację o rozmiarze tablicy. Odwołanie się do konkretnego obiektu tablicowego jest użycie zapisu z nawiasem kwadratowym []. Poniżej przedstawiono kolejno sposoby deklaracji i inicjalizacji tablicy.



```
//DEKLARACJA - OBA ZAPISY SĄ POPRAWNE
typTablicy[] nazwaTablcy;
typTablicy nazwaTablicy[];

//INICJALIZACJA
nazwaTablicy = {element0,element1,element2,...,element_nty};

//DEKLARACJA I INICJALIZACJA
typTablicy [] nazwaTablcy =
{element0,element1,element2,...,element_nty};

//DEKLARACJA I INICJALIZACJA BEZ PRZYPISANIA ELEMENTOM
WARTOŚCI
typTablicy [] nazwaTablcy = new typTablicy[rozmiar];
```

Poniżej przedstawiono przykładowy zapis i odczyt z tablicy przechowującej liczby całkowite:

```
//DEKLARACJA I INICJALIZACJA 100-ELEMENTOWEJ TABLICY LICZB
CAŁKOWITYCH
int mojaTablica[] = new int[100];

//ZAPIS DO TABLICY WARTOŚCI RÓWNYCH INDEKSOM
for(int i = 0; i < mojaTablica.length; i++){
    mojaTablica[i] = i;
}

//ODCZYT WARTOŚCI Z TABLICY I WYPISANIE ICH NA KONSOLI
for(int i = 0; i < mojaTablica.length; i++){
    System.out.println(mojaTablica[i]);
}
```

W języku Java jest możliwość zwrócenia całej tablicy (np. w wyniku działania metody), co nie występuje w językach C i C++. Typ zwracany metody jest deklarowany jako tablica (użycie []), a po słowie return należy wpisać nazwę konkretnej tablicy zwracanej przez metodę (sama nazwa, bez nawiasów kwadratowych). Typ zwracanej i zadeklarowanej w metodzie tablicy muszą być zgodne. Poniżej przedstawiono przykładowy kod:

```
public int[] metodaCalkowita(){
    int tabliczka[] = new int[28];

    //kolejne instrukcje...

    return tabliczka;
}
```



Tablice wielowymiarowe deklaruje się analogicznie do jednowymiarowych – każdy kolejny wymiar dodajemy nawiasem kwadratowym. Określenie rozmiaru jest wówczas konieczne dla co najmniej jednego wymiaru (a nie dla wszystkich). Przykładowa deklaracja tablicy wielowymiarowej:

```
int tablicaWielowymiarowa[][] = new int[20][20];
```

W przypadku takiej deklaracji zostanie utworzona tablica dwuwymiarowa z dwudziestoma kolumnami i wierszami.

Każdy wiersz tablicy tworzącej macierz może mieć dowolny rozmiar. Przykład wielowymiarowej tablicy obiektów przedstawiono poniżej:

```
Integer tablica[][] = new Integer [10][];
for(int i = 0; i < tablica.length; i++){
    tablica[i] = new Integer[3];
    for(int j = 0; j < tablica[i].length; j++){
        tablica[i][j] = i + j;
    }
}
```

W klasie **Arrays** można znaleźć wiele metod przydatnych do wykonywania określonych operacji na tablicach. Podstawowe metody przedstawiono w tabeli 4.1.

Tab.4.1. Podstawowe metody klasy Arrays.

Modyfikator i typ	Metoda	Opis
static int	binarySearch (Object[] a, Object key)	Wyszukuje obiekt w tablicy przy użyciu algorytmu wyszukiwania binarnego
static boolean	deepEquals (Object[] a1, Object[] a2)	Zwraca prawdę, jeśli dwie tablice obiektów są sobie równe – dla tablic wielowymiarowych
static boolean	equals (Object[] a, Object[] a2)	Zwraca prawdę, jeśli dwie tablice obiektów są sobie równe
static void	fill (Object[] a, Object val)	Przypisuje określoną referencje do obiektu każdemu elementowi z tablicy obiektów
static void	sort (Object[] a)	Sortuje tablicę obiektów w kolejności rosnącej zgodnie z naturalnym uporządkowaniem elementów

Dodatkowo standardowa biblioteka Javy dostarcza metodę **System.arraycopy()** służącą do kopiowania tablic (znacznie szybciej, niż przepisywanie z tablicy do tablicy przy użyciu pętli for).



Klasa Math

Klasa **Math** zawiera metody do przeprowadzania podstawowych operacji numerycznych, takich jak elementarne funkcje wykładnicze, logarytmiczne, potęgowe i trygonometryczne. Równolegle w pakiecie java.lang występuje klasa **StrictMath**, która jest prawie identyczna z klasą Math. Jedyne różnice jakie można zauważyć po zapoznaniu się z dokumentacją to w przypadku klasy StrictMath metody zwracają takie same wyniki bez względu na platformę sprzętową – kosztem wydajności. W przeciwieństwie do niektórych metod numerycznych klasy StrictMath, wszystkie implementacje równoważnych funkcji klasy Math nie są zdefiniowane tak, aby zwracały te same wyniki bit po bicie. To rozluźnienie umożliwia bardziej wydajne implementacje, w których ścisła powtarzalność nie jest wymagana. Domyślnie wiele metod Math po prostu wywołuje równoważną metodę w StrictMath w celu ich implementacji. Podsumowując szybsze i mniej dokładne wyniki (inne w zależności od platformy) otrzymamy wykorzystując metody z klasy Math. W tabeli 4.2 przedstawiono wybrane metody klasy Math.

Tab.4.2. Wybrane metody klasy Math.

Modyfikator i typ	Metoda	Opis
static int	abs(int a)	Zwraca wartość bezwzględną danej liczby całkowitej
static double	exp(double a)	Zwraca liczbę Eulera podniesioną do zadanej potęgi (e^a)
static double	random()	Zwraca pseudolosową wartość z zakresu $<0.0,1.0$)
static double	sqr(double a)	Zwraca zaokrąglony dodatni pierwiastek kwadratowy
static double	toDegrees(double angdeg)	Konwertuje kąt mierzony w radianach na w przybliżeniu równoważny kąt mierzony w stopniach
static double	toRadians(double angdeg)	Konwertuje kąt mierzony w stopniach na w przybliżeniu równoważny kąt mierzony w radianach

Wszystkie metody klasy Math mają modyfikator static co oznacza, że są one ściśle związane z klasą, a nie z obiektem klasy – wywołanie bez obiektu.

Precyzyjne obliczenia

W przypadku operowania na liczbach przekraczających zakres typów podstawowych pomocne mogą okazać się dwie klasy – **BigInteger** i **BigDecimal**. Klasa BigInteger reprezentuje typ całkowity dowolnej precyzji. Oznacza to, że można wiernie reprezentować zmienne typu całkowitego dowolnych rozmiarów bez utraty informacji. Klasa BigDecimal jest przeznaczona dla liczb stałoprzecinkowych dowolnej precyzji – jest to klasa polecana w przypadku wykonywania dokładnych obliczeń finansowych.



Klasa Random

Obiekt tej klasy służy do generowania strumienia liczb pseudolosowych. Klasa używa 48-bitowego ziarna – jeśli dwa obiekty klasy **Random** zostaną utworzone z tym samym ziarnem i dla każdego zostanie utworzona ta sama sekwencja wywołań metod, wygenerują i zwrócą identyczne sekwencje liczb. Instancje klasy Random są bezpieczne dla wątków. Jednak równoczesne użycie tej samej instancji java.util.Random w różnych wątkach może skutkować słabą wydajnością. Zamiast tego należy rozważyć użycie klasy **ThreadLocalRandom** w projektach wielowątkowych. Jednocześnie obiekty klasy Random nie są zabezpieczone kryptograficznie. Zamiast tego należy wziąć pod uwagę użycie klasy **SecureRandom**, aby uzyskać bezpieczny kryptograficznie generator liczb pseudolosowych do wykorzystania w aplikacjach wrażliwych na bezpieczeństwo. W tabeli 4.3 przedstawiono wybrane metody klasy Random.

Tab.4.3. Wybrane metody klasy Random

Modyfikator i typ	Metoda	Opis
void	setSeed(long seed)	Ustawia ziarno generatora liczb pseudolosowych na liczbę całkowitą (long)
int	nextInt(int a)	Zwraca pseudolosową liczbę całkowitą z zakresu <0,a)
double	nextDouble()	Zwraca pseudolosową liczbę rzeczywistą z zakresu <0.0,1.0>

Przykład użycia klasy Random przedstawia poniższy kod:

```
Random rnd = new Random();
for(int i = 0; i < 50; i++)
    System.out.println(rnd.nextInt(21)-10);
```

Wykorzystując pętlę for 50 liczb z zakresu <-10,10> zostanie wylosowanych i wypisanych, każda w oddzielnym wierszu.

Zadanie 4.1. Tablica jednowymiarowa

- Napisz metodę, która utworzy jednowymiarową, 100–elementową tablicę liczb całkowitych, wypełni ją wartościami 0 – 99 i zwróci wypełnioną tablicę.
- Napisz metodę, która przyjmie jako argument jednowymiarową tablicę liczb całkowitych, a następnie wyświetli zawartość w sposób następujący:

```
00, 01, 02, 03, 04, 05, 06, 07, 08, 09,
10, 11, 12, 13, 14, 15, 16, 17, 18, 19,
20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
30, 31, 32, 33, 34, 35, 36, 37, 38, 39,
40, 41, 42, 43, 44, 45, 46, 47, 48, 49,
50, 51, 52, 53, 54, 55, 56, 57, 58, 59,
60, 61, 62, 63, 64, 65, 66, 67, 68, 69,
70, 71, 72, 73, 74, 75, 76, 77, 78, 79,
80, 81, 82, 83, 84, 85, 86, 87, 88, 89,
90, 91, 92, 93, 94, 95, 96, 97, 98, 99,
```



Liczby 0 – 9 wyświetlaj jako dwucyfrowe (dodaj 0 na początku). Dodatkowo metoda ma obliczać średnią wartość w każdej kolumnie i wyświetlać ją w dodatkowym wierszu – odpowiednio pod odpowiadającą jej kolumną.

Zadanie 4.2. Tablica dwuwymiarowa

- Napisz metodę, która utworzy kwadratową tablicę dwuwymiarową liczb całkowitych o losowej wielkości (wylosowanej z zakresu [10,20]). Liczba kolumn i wierszy ma być taka sama. Tablica ma zostać wypełniona wartościami wylosowanymi z przedziału [-20,20], z wyjątkiem elementów znajdujących się na przekątnej tablicy, które losowo otrzymają wartość -1 lub 1.
- Napisz metodę, która obliczy stosunek sumy liczb leżących w komórkach tablicy o parzystych indeksach wierszy do sumy liczb leżących w komórkach tablicy o nieparzystych indeksach kolumn. Wykorzystaj klasę Random.

Zadanie 4.3. Klasa Okrag

- Utwórz klasę Okrag:
 - Zadeklaruj 3 pola odpowiednio dla współrzędnych x, y oraz promień okręgu,
 - Dodaj konstruktor ustawiający odpowiednio wszystkie pola klasy:
 - Współrzędne jako losowe liczby z zakresu <5,95>,
 - Promień jako losową liczbę z zakresu <1,5>.
- W klasie głównej dodaj trzy metody:
 - Pierwsza ma utworzyć tablicę obiektów klasy Okrag.
 - Druga ma sprawdzić wzajemne położenie dwóch dowolnych okręgów na płaszczyźnie i zwrócić stosowny komunikat (styczne, rozłączne, przecinające się, pokrywające się). Metoda ma przyjmować jako parametry dwa obiekty klasy Okrag.
 - Trzecia ma wyświetlić w jednym wierszu wszystkie pola danego obiektu (każdy obiekt w nowej linii). Wykorzystaj pętlę for – each.



LABORATORIUM 5. PROGRAMOWANIE OBIEKTOWE Z WYKORZYSTANIEM DZIEDZICZENIA I INTERFEJSÓW.

Cel laboratorium:

Zapoznanie z mechanizmami dziedziczenia i modyfikatorami dostępu. Tworzenie klas abstrakcyjnych i interfejsów. Wykorzystanie klas nadrzędnych.

Zakres tematyczny zajęć:

- Dziedziczenie.
- Modyfikatory dostępu.
- Mechanizm refleksji.
- Klasa abstrakcyjna.
- Interfejs.

Pytania kontrolne:

1. Na czym polega dziedziczenie?
2. Jakie znasz modyfikatory dostępu?
3. Do czego służy refleksja?
4. Czym różni się klasa abstrakcyjna od interfejsu?

Dziedziczenie

Dziedziczenie jest jednym z podstawowych mechanizmów występujących w programowaniu obiektowym. Pozwala na zmniejszenie ilości generowanego kodu poprzez utworzenie jednoznacznej hierarchii wykorzystywanych klas. Przykładem przywoływanym w literaturze jest klasa *Figura* oraz dziedziczące po niej klasy: *Koło*, *Prostokąt*, *Trójkąt*. W klasie nadrzędnej (*Figura*) definiowane są odpowiednie pola oraz metody np.: *współrzedneŚrodka*, *obliczPole()*, *obliczObwód()*, *rysuj()*, *ustawKolor()* itp.. Klasy podrzędne (*Koło*, *Prostokąt*, *Trójkąt*) mają dostęp do tych pól i metod. Mogą je dowolnie wykorzystać lub nadpisać (ang. *override*) czyli stworzyć nową definicję metody. **W Javie każda klasa może dziedziczyć wyłącznie po jednej klasie.** Dopuszczalne jest dziedziczenie wielopoziomowe np.:

Figura ← Prostokąt ← Kwadrat

gdzie strzałka zawsze wskazuje klasę nadrzędną(bazową) która jest uogólnieniem klasy podrzędnej np. kwadrat jest szczególnym przypadkiem prostokąta. Do zdefiniowania klasy dziedziczącej używane jest słowo kluczowe ***extends*** w następujący sposób:

```
modyfikatory class KlasaPodrzedna extends KlasaNadrzedna {  
  
}
```



Modyfikatory dostępu

Modyfikator dostępu to słowo kluczowe poprzedzające element (klasę, pole, metodę) określając jego widoczność/dostępność zgodnie z dokumentacją. Modyfikatory występujące w języku Java:

- public,
- protected,
- brak modyfikatora (package-private),
- private.

Poziom dostępu (widoczność) dla konkretnych modyfikatorów przedstawia tabela 5.1.

Tab.5.1. Poziom dostępu w zależności od modyfikatora

Modyfikator	Klasa	Pakiet	Podklasa	Wszędzie
public	Tak	Tak	Tak	Tak
protected	Tak	Tak	Tak	Nie
brak modyfikatora	Tak	Tak	Nie	Nie
private	Tak	Nie	Nie	Nie

Hermetyzacja (lub inaczej enkapsulacja) jest jedną z cech obiektowych języków programowania. Polega na ukrywaniu szczegółów implementacji klasy – dzięki takiemu działaniu nie ma ryzyka zmiany stanu obiektu w sposób nieuprawniony lub przypadkowy. W zależności od użytych modyfikatorów dostępu można ograniczyć dostęp „z zewnątrz” klasy do konkretnych obszarów. Dobrą praktyką jest stosowanie najbardziej restrykcyjnych ograniczeń – modyfikator private – dla wszystkich elementów klasy używanych wewnątrz. Pozostałe elementy odpowiadające za komunikację z innymi klasami można oznaczyć modyfikatorem mniej restrykcyjnym np. public. Za niedopuszczalną należy uznać sytuację w której można zmodyfikować wartości pól danego obiektu jeśli taka możliwość nie była planowana.

Mechanizm dziedziczenia pozwala na zmianę modyfikatora dostępu w nadpisanych metodach. Finalnie do dyspozycji są dwie metody – jedna z klasy bazowej i druga z klasy potomnej. Każda z nich może mieć inny modyfikator dostępu. Aby zapobiec zmianie definicji metody należy wykorzystać słowo kluczowe **final**.

Mechanizm refleksji jest obejściem zasady hermetyzacji – pozwala bowiem na dostęp do konkretnego pola klasy omijając modyfikator dostępu. **Nie jest zalecane** używanie takiego rozwiązania, a jego opis został podany jedynie w celach edukacyjnych. W mechanizmie refleksji wykorzystywana jest klasa **Field**. Klasa ta dostarcza informacje i dostęp dynamiczny do pojedynczego pola klasy lub interfejsu. Przeanalizuj poniższy przykład:

```
public class Konto {

    private int stanKonta = 250;

    public int podajStan(){
        return stanKonta;
    }

}
```

W metodzie main zastosowano mechanizm refleksji z wykorzystaniem klasy Field:



Fundusze Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny



```
public static void main(String[] args) throws
IllegalAccessException, NoSuchFieldException {

    Konto mojeKonto = new Konto();
    System.out.println("Stan mojego konta: "+
mojeKonto.podajStan());

    Field zlodziej =
Konto.class.getDeclaredField("stanKonta");
    zlodziej.setAccessible(true);
    zlodziej.set(mojeKonto, -2500);

    System.out.println("Stan mojego konta: "+
mojeKonto.podajStan());
}
}
```

Wynik wyświetlony w konsoli po uruchomieniu powyższego kodu:

```
Stan mojego konta: 100
Stan mojego konta: -2500
```

Pomimo braku metody ustawiającej pole stanKonta w klasie Konto jest możliwy (ale niewskazany) dostęp do pola prywatnego. Analogicznie do klasy Field w pakiecie java.lang.reflect istnieją również klasy zapewniające dostęp do metod i konstruktorów – odpowiednio klasy **Method** i **Constructor**.

Klasa abstrakcyjna

Klasę abstrakcyjną można rozumieć jako szkielet bazowy dla innej klasy. Posiada metody abstrakcyjne i nie ma możliwości utworzenia obiektu. Może być dziedziczona, wówczas klasa potomna musi przesłonić wszystkie metody abstrakcyjne z klasy nadrzędnej, w przeciwnym razie będzie klasą abstrakcyjną. Do utworzenia stosuje się słowo kluczowe abstract. Metody abstrakcyjne nie posiadają ciał – występuje jedynie nazwa.

```
public abstract class KlasaOgolna {

    abstract void metodaAbstrakcyjna();

    public void metodaNieabstrakcyjna() {

    }

}
```

Implementacja metod abstrakcyjnych zachodzi dopiero w klasie potomnej, która jest zmuszona do dodania wszystkich metod abstrakcyjnych z klasy nadrzędnej. Metody nieabstrakcyjne nie muszą być dodawane (ale mogą) do klasy potomnej.

Interfejs

Interfejs podobnie jak klasa abstrakcyjna jest szkieletem – określa jakie metody i pola muszą znaleźć się w klasie. Jest definiowany przy użyciu słowa kluczowego **interface**. W celu utworzenie interfejsu na drzewie projektowym należy rozwinąć katalog Source Packages, a następnie kliknąć prawym przyciskiem myszy (PPM) na pakiet np.: com.mycompany.dziedziczenie, wybrać New → Java Interface. W dalszej kolejności wybieramy nazwę dla interfejsu i przyciskiem Finish dodajemy go do pakietu. Utworzony zostaje plik o zadanej nazwie z rozszerzeniem *.java, który można zmieniać przy użyciu edytora kodu. Głównym składnikiem interfejsu są publiczne metody abstrakcyjne – metody bez implementacji. Nie jest konieczne używanie modyfikatorów w tym przypadku (domyślnie metody są public abstract). Poza metodami abstrakcyjnymi interfejs może zawierać pola public static final (stałe) – nie ma konieczności jawnego stosowania modyfikatorów. Pierwotnie interfejsy mogły zawierać wyłącznie metody abstrakcyjne ale od ukazania się JDK w wersji 8 mogą zawierać metody domyślne – posiadające implementację (ciało) już wewnątrz interfejsu. W celu utworzenia metody domyślnej należy użyć słowa kluczowego **default** przed typem metody.

```
public interface InterfejsPodstawowy {  
  
    void pustaMetoda();  
  
    int calkowitaMetoda();  
  
    default void domyslnaMetoda() {  
        System.out.println("Jesteś w metodzie domyślnej");  
    }  
}
```

Implementowanie interfejsu odbywa się poprzez użycie słowa kluczowego **implements** w sposób następujący:

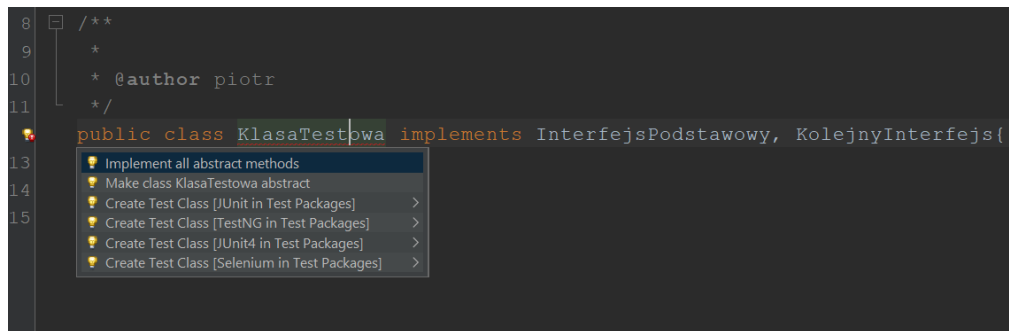
```
public class KlasaTestowa implements InterfejsPodstawowy{  
  
}
```

W Javie możliwe jest implementowanie wielu interfejsów przez jedną klasę – w przeciwieństwie do rozszerzania klasy (dziedziczenia), gdzie możliwe jest dziedziczenie wyłącznie po jednej klasie nadrzędnej. Jest to zdecydowana zaleta interfejsów. Możliwa jest również implementacja interfejsu w klasie która już dziedziczy po innej klasie. Poniżej przedstawiono opisaną sytuację:

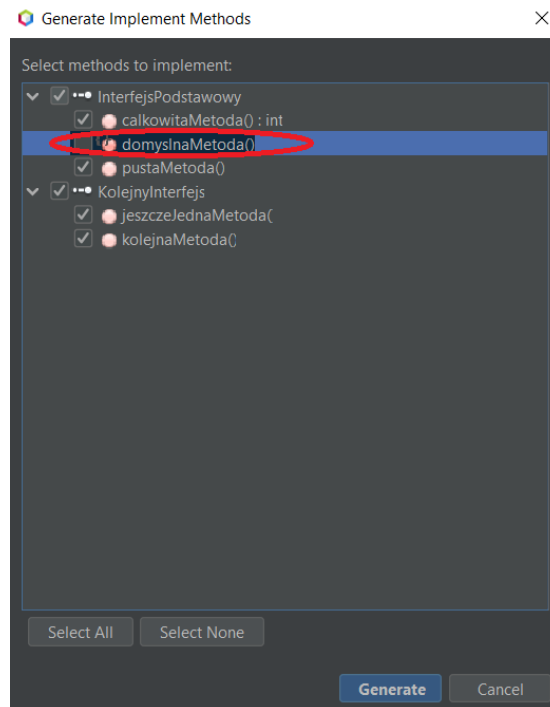
```
public class KlasaTestowa extends KlasaOgolna implements  
InterfejsPodstawowy, KolejnyInterfejs{  
  
}
```



W klasie implementującej interfejs muszą znaleźć się wszystkie metody abstrakcyjne. Najprościej po implementacji interfejsu wykorzystać pomocne IDE i dodać je automatycznie przy użyciu skrótów klawiaturowych. W NetBeans IDE po dodaniu nazwy interfejsu wiersz z nazwą klasy zostanie podświetlony na czerwono – należy wysunąć listę wskazówek i wybrać **Implement all abstract methods** – służy do tego skrót Alt + Enter (rys.5.1). Nie wszystkie metody muszą być dodane – wyłącznie abstrakcyjne. Po dodaniu metod należy zwrócić uwagę na wartości domyślne zwracane przez metody. W zależności od ustawień środowiska można dla generowanych metod ustawić odpowiednie „wypełnienie” poprzez Tools → Templates → Java → Code Snippets.



Rys.5.1. Implementacja interfejsu.



Rys.5.2. Dodanie metod.

Zadanie 5.1. Dziedziczenie

Utwórz nową klasę – Figura. Zdefiniuj w klasie pola:

- protected:
 - int pole,
 - int obwód,
 - private:
 - Color kolor (klasa importowana) .
- Zdefiniuj w klasie metody chronione (protected):
- void rysuj(),
 - void usuń() ,
 - void przesun(),
 - String podajParametry(),

Dodaj konstruktor parametrowy oraz metody typu get i set (ALT+Insert → getter and setter dla pól klasy). Utwórz nowe klasy dziedziczące po klasie Figura ← Elipsa, Figura ← Wielokąt. Wróć do klasy figura i zdefiniuj tam konstruktor bezparametrowy. Usuń modyfikator dostępu klasy Elipsa i Wielokąt. Zdefiniuj w klasach odpowiednio pola prywatne:

- W klasie Elipsa:
 - półoś wielka (a), półoś mała (b), pozycje x,y dla dwóch ognisk (F1x, F1y, F2x, F2y) oraz odległość ognisk od środka elipsy (c).
- W klasie Wielokąt:
 - liczba wierzchołków, liczba boków, suma kątów wewnętrznych.

Dodaj konstruktory, gettery i settery dla każdej z klas – zauważ, że klasa dziedzicząca może wykorzystać konstruktor klasy nadrzędnej. Zwróć uwagę na słowo super – do czego się odnosi? Sprawdź czy masz dostęp do pól klasy bazowej. Dodaj/nadpisz dla każdej klasy metody z klasy bazowej (Alt+Insert → Override Methods). Zwróć uwagę na adnotację oznaczoną symbolem @ oraz ponownie słowo super. Metody mogą zawierać wypisanie informacji np.:

```
System.out.println("Trwa rysowanie wielokąta o " + wierzcholki  
+ "wierzchołkach");
```

Dodaj metody obliczające:

- W klasie Elipsa: odległość ognisk od środka elipsy (c):

$$c = \sqrt{a^2 - b^2}$$

- W klasie Wielokąt: sumę kątów wewnętrznych:

$$sumaKatow = (n - 2) \cdot 180^\circ$$

gdzie n jest liczbą boków wielokąta.

Utwórz nowe klasy dziedziczące:

Elipsa ← Okrąg

Wielokąt ← Trójkąt

Wielokąt ← Prostokąt



Fundusze Europejskie
Wiedza Edukacja Rozwój



Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny



Prostokąt ← Kwadrat

Dodaj odpowiednie pola np. w klasie Trójkąt wysokość (h) oraz podstawa (a). Dla pozostałych klas analogicznie. W każdej z klas utwórz odpowiednie konstruktory ustawiające pola klas nadrzędnych. Nadpisz również metody – mogą zawierać sam komunikat. Dodaj również metody obliczające pole i obwód i ustawiające odpowiednie pola np.: po wywołaniu metody obliczObwód () dla obiektu klasy Okrąg ma zostać ustawione pole klasy (obwód) z klasy Figura.

W jaki sposób wywołać na obiekcie klasy podrzędnej oryginalną metodę klasy nadrzędnej? Utwórz nowy pakiet (drzewo projektowe → katalog src → New → package). Stwórz nową klasę (nazwa dowolna). Sprawdź widoczność/dostępność wszystkich stworzonych wcześniej klas. Co trzeba zmienić?

Zadanie 5.2. Klasa abstrakcyjna i interfejs

Utwórz klasę abstrakcyjną Czlowiek (słowo kluczowe abstract – przed słowem class). Dodaj do klasy metody abstrakcyjne:

- jedz(),
- pij().

Dodaj do klasy metody z implementacją:

- ileLat(),
- cechy().

Utwórz dwie kolejne klasy:

- Dziecko i Dorosły, dziedziczące po klasie Czlowiek.

Dodaj do tych klas metody abstrakcyjne:

- zabawa(),
- obowiazki().
- dodaj 2 autorskie metody z implementacją

Czy klasy Dziecko i Dorosły należy zmodyfikować w jakiś sposób? Dodaj kolejne klasy: Uczeń, Student, Emeryt, dziedziczące odpowiednio po klasach Dziecko oraz Dorosły. Co należy dodać do każdej z klas?

- Stwórz interfejsy z metodami:
 - Podstawowy
 - jedz(),
 - pij(),
 - spij(),
 - wstan().
 - Szkoła
 - uczSie(),
 - odrobLekcje(),
 - dodaj 2 autorskie metody.
 - Studia
 - studiuj(),
 - nieIdzNaZajecia(),
 - dodaj 2 autorskie metody.
 - Praca
 - pracuj(),
 - placPodatki(),
 - dodaj 2 autorskie metody.



- Emerytura
 - odbierzEmeryture(),
 - idzDoLekarza(),
 - dodaj 2 autorskie metody.

Zaimplementuj każdy interfejs (dla klasy Uczeń – Podstawowy i Szkoła, dla klasy Student – Podstawowy, Studia (i ewentualnie Praca), dla klasy Emeryt – Podstawowy, Emerytura (i ewentualnie Praca)). Dodaj prostą implementację metod w klasach dziedziczących. Utwórz obiekty każdej klasy (o ile to możliwe) i wywołaj dla nich odpowiednie metody.

LABORATORIUM 6. OBSŁUGA WYJĄTKÓW.

Cel laboratorium:

Zapoznanie z obsługą wyjątków, hierarchia wyjątków.

Zakres tematyczny zajęć:

- Wyjątek i błąd.
- Klauzula try – catch – finally.
- Klauzula throws.

Pytania kontrolne:

1. Co to jest wyjątek i czym różni się od błędu?
2. W jaki sposób należy obsłużyć błąd?
3. W jaki sposób należy obsłużyć wyjątek?
4. Czy można zadeklarować własny wyjątek?

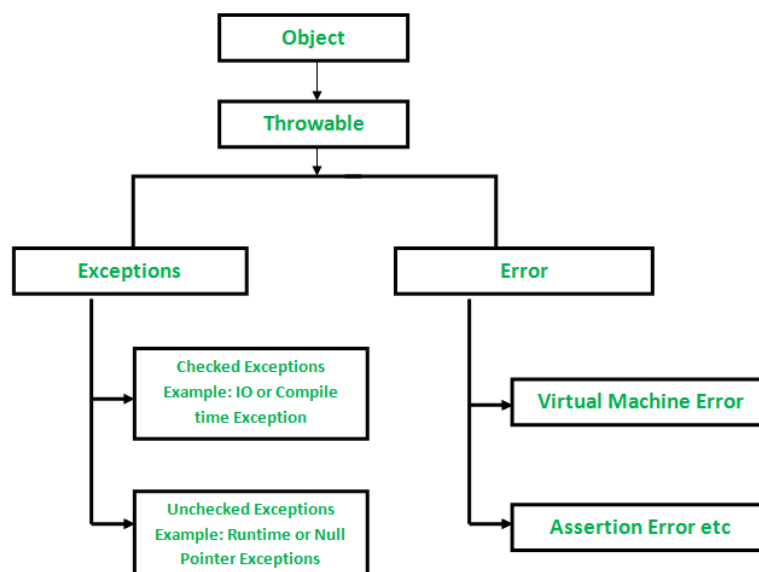
Wyjątek i błąd

Bardzo często kod, który tworzymy może działać nieprawidłowo. Niestandardowe działanie aplikacji może prowadzić do przerwania jej działania. Język Java rozdziela nieprawidłowe działanie aplikacji na dwa rodzaje: wyjątki i błędy.

W Javie wyjątki należą do specjalnej klasy **Exception** – jeżeli w trakcie działania programu wydarzy się coś „niedozwolonego” obiekty tej klasy są „wyrzucane” (ang. throw) i przerywają normalną pracę programu. Przykładem takiej sytuacji może być odwołanie się do elementu tablicy, który nie istnieje lub wykonanie niedozwolonego działania – dzielenie przez 0. Istnieje kilka narzędzi do wyłapywania takich przypadków i „naprawy” programu w trakcie jego działania (czyt. poprawnej obsługi wyjątków).

Klasa **Error** jest odpowiedzialna za wskazywanie poważnych usterek w działaniu aplikacji, które nie powinny być wyłapane i obsłużone (w przeciwieństwie do wyjątków). Błędy są krytycznymi warunkami, które występują z powodu braku zasobów systemowych i nie mogą być obsługiwane przez kod programu. ponieważ kod nie jest za nie odpowiedzialny np. błąd braku pamięci lub błąd przepełnienia stosu (*OutOfMemory*, *StackOverflow*) są błędami systemu. Konsekwencją wystąpienia błędu jest nieprawidłowe zakończenie programu.

Wszystkie typy wyjątków i błędów są podklasami klasy **Throwable**, która jest bazową klasą w hierarchii którą przedstawiono na rysunku 6.1.



Rys. 6.1. Hierarchia klasy Throwable.

Sama klasa Throwable nie ma większego zastosowania ale każda klasa która po niej dziedziczy może „wyrzucać” przy użyciu słowa **throws**. W hierarchii wyjątków występuje jeszcze jedna klasa – **RuntimeException**. Jest to grupa wyjątków wywoływanych głównie z niedopracowania kodu – zaliczają się do niej takie klasy jak np.: *NullPointerException* (wywołanie metody na obiekcie który jest null’em). Nie ma przymusu obsługi wyjątków z tej grupy – każda linijka kodu może potencjalnie powodować wyrzucenie wyjątku z tej grupy.

Obsługa wyjątków

Do obsługi wyjątków służy blok try – catch – finally. W części try umieszczany jest kod mogący potencjalnie wyrzucić wyjątek (np. próba dzielenia przez 0). Po części try następuje catch, który ma za zadanie wyłapać wyjątek konkretnego rodzaju i odpowiednio zareagować. Na poniższym przykładzie przedstawiono metodę dzielącą liczbę x przez liczbę podaną przez użytkownika:

```

public static void dziel(int x) {
    Scanner skaner = new Scanner(System.in);
    try {
        int y = skaner.nextInt();
        System.out.println("Twój wynik to: " + x / y);
    } catch (ArithmeticException e) {
        System.err.println("Podaj poprawny dzielnik");
        dziel(x);
    } catch (InputMismatchException e) {
        System.err.println("Podaj poprawny dzielnik");
        dziel(x);
    }
}
    
```



Metoda main składa się z jednej linijki:

```
public static void main(String[] args) {  
    dziel(100);  
}
```

Po wprowadzeniu przez użytkownika wartości 0 zostanie wyrzucony wyjątek *ArithmeticException* który zostanie obsługowany w sekcji catch. Następuje tam informacja o błędnie podanym dzielniku oraz ponowne wywołanie metody *dziel()*. Dzięki temu użytkownik ma szansę ponownie podać poprawną już wartość dzielnika.

Należy zauważyć, że poza zwykłym wprowadzeniem przez użytkownika liczby 0 może wystąpić kolejny problem – wprowadzenie dowolnego innego znaku np. litery czy znaku specjalnego. Próba odczytania kolejnej liczby całkowitej z ciągu „aaaaaaa” będzie skutkowałą wyrzuceniem wyjątku *InputMismatchException* dla którego teoretycznie nie ma konieczności obsługi.

Jak pokazano możliwe jest wyłapywanie więcej niż jednego wyjątku dla kodu (może to być nawet jedna linijka) w sekcji try. Dodawane są kolejne sekcje catch w kolejności od najbardziej szczegółowych przypadków do ogólnych. Z tego względu należy znać hierarchię klasy Exception. Poprawne zaklasyfikowanie wyjątku to podstawa obsługi.

Część finally która następuje po ostatnim catchu wykona się zawsze, niezależnie od tego czy wyjątek wystąpi czy też nie. W tej części zwykle następuje „sprzątanie” np. zamyka się wszystkie otwarte strumienie, pliki, połączenia. Dodatkowo jeżeli metoda zwraca jakąś wartość to return umieszczone w części finally zawsze się wywoła (mimo umieszczenia return z inną wartością w części try lub catch).

Ważna uwaga! Jeśli nie można rozwiązać problemu który doprowadził do zgłoszenia wyjątku to przynajmniej należy wyświetlić aktualny stan stosu korzystając z metody `printStackTrace()` dla każdej klasy wyjątku.

Drugim sposobem obsługi wyjątków jest zastosowanie klauzuli **throws**. Polega on na zadeklarowaniu w sygnaturze metody konkretnego wyjątku (lub wyjątków). Takie oznaczenie sygnalizuje, że dana metoda może powodować wyjątek danego typu. Zanim wyjątek zostanie obsługowany należy poprawnie go zaklasyfikować do istniejących lub stworzyć własny dziedziczący np. po *IOException*. Wiedząc jakiego rodzaju wyjątek będzie obsługiwany można skorzystać z klauzuli throws w metodzie po liście parametrów zgodnie z poniższym przykładem:

```
public static void znowDzielenie(int x) throws  
ArithmeticException {  
  
    Scanner skaner = new Scanner(System.in);  
    int y = skaner.nextInt();  
    if (y == 0)  
        throw new ArithmeticException("Podaj poprawny  
mianownik");  
    else  
        System.out.println("Twój wynik to: " + x / y);  
}
```



W ten sposób wyjątek może zgłaszać każda metoda – zwłaszcza jeśli wyjątek nie należy do dostępnych odgórnie. Innymi słowy w ten sposób można pisać własne klasy wyjątków oraz metody które je zgłaszają. Dalsza droga jest już znana – wykorzystanie konstrukcji try – catch do obsługi własnego wyjątku.

Zadanie 6.1. Kalkulator

Napisz prosty program konsolowy realizujący funkcjonalności kalkulatora:

- Wykonywanie podstawowych działań (dodawanie, odejmowanie, mnożenie, dzielenie, pierwiastkowanie, potęgowanie) i wyświetlanie wyniku.

Zadbaj o odpowiednią obsługę wyjątków. Załóż, że użytkownik może wprowadzić dowolny znak (w przypadku wprowadzenia błędnych danych wyświetl komunikat i ponów prośbę o podanie danych).

Zadanie 6.2. Pobieranie liczby rzeczywistej

Napisz metodę pobierającą od użytkownika liczbę zmiennoprzecinkową, rozdzielającą liczbę na część oraz mantysę, a następnie zwracającą iloraz części i mantysy. W przypadku, gdy nie jest to możliwe wypisz komunikat i ponów pobieranie liczby. Załóż, że użytkownik może wprowadzić dowolny znak.

Zadanie 6.3. Pobieranie liczby całkowitej

Napisz metodę pobierającą o użytkownika liczbę całkowitą z zakresu $\langle 99,999 \rangle$. Zwróć sumę kwadratów jej cyfr. Załóż, że użytkownik może wprowadzić dowolny znak.



LABORATORIUM 7. WYKORZYSTANIE KOLEKCJI.

Cel laboratorium:

Wprowadzenie do kolekcji, rodzaje kolekcji. Praca z gotowym kodem – wprowadzanie modyfikacji.

Zakres tematyczny zajęć:

- Kolekcje – klasy i interfejsy.
- Podstawowe operacje na kolekcjach.
- Praca z gotowym kodem.

Pytania kontrolne:

1. Czym jest kolekcja?
2. Czy jest różnica między kolekcją, a tablicą?
3. Jakie znasz typy kolekcji.
4. Jakie operacje można przeprowadzić na kolekcjach?
5. Jakie są specjalne zastosowania kolekcji?

Kolekcje

Program obejmujący wyłącznie ustaloną liczbę obiektów, których czas życia jest znany, to program dość prosty.

W programie zakładającym jakąkolwiek interakcję z użytkownikiem na zasadach bardziej skomplikowanych niż podanie dwóch liczb i wypisanie ich sumy nie jest znana faktyczna liczba obiektów ani nawet ich dokładny typ. Tworzenie „sztywnych” referencji postaci:

NazwaKlasy nazwaObiektu;

nie może być stosowane we wszystkich rozwiązaniach. Rozwiązaniem tego problemu wydaje się być tablica obiektów (referencji do obiektów) ale ma ona ograniczenia – rozmiar tablicy musi być stały i znany w momencie jej utworzenia. Język Java w swojej bibliotece użytkowej posiada szeroki zbiór kolekcji. ***Collections*** jest klasą narzędziową wyposażoną w metody statyczne do wykonywania operacji na obiektach klas implementujących interfejs ***Collection***. W hierarchii interfejsów można wyróżnić podstawowe podinterfejsy tj. ***List***, ***Set*** i ***Queue*** oraz niezależny od interfejsu ***Collection*** interfejs ***Map***. W interfejsie zdefiniowane są metody wspólne dla danego rodzaju kontenerów np. metody dodawania elementu do kolekcji: ***add(E e)*** lub zwrócenie elementu o danym indeksie: ***get(int index)***. Dodatkową uwagę warto zwrócić na metodę ***size()***, która zwraca liczbę elementów w kolekcji i każde wywołanie może dać inny wynik (o ile kolekcja zmieniła rozmiar). Kolekcje oferują wyszukane sposoby przechowywania obiektów dla zaskakująco wielu problemów programistycznych. Każdy rodzaj kolekcji podlega ściśle określonym regułom. Lista typu ***List*** przechowuje elementy w określonej kolejności, zbiór ***Set*** nie zawiera duplikatów elementów, a kolejka ***Queue*** porządkuje elementy zgodnie z dyscypliną kolejki. Mapa ***Map*** grupuje pary obiektów według typu klucz – wartość i pozwala wydobyć konkretną wartość dla danego klucza.



Dodawanie grupy elementów do kolekcji

W interfejsie Collection (i wszystkich dziedziczących po nim) występuje metoda służąca do dodawania pojedynczego elementu do kolekcji. Jednak w wielu przypadkach może to być niewystarczające – dodawanie pojedynczo (metodą add()) np. 200-elementowej tablicy mija się z celem. W klasach Collections i Arrays są metody grupujące elementy do postaci kolekcji zgodnie z interfejsem Collection. Metoda statyczna asList() z klasy Arrays przyjmuje jako argument tablicę lub listę elementów wymienionych po przecinku i zwraca obiekt klasy List. Metoda statyczna addAll() z klasy Collections przyjmuje kolekcję (zgodną z interfejsem Collection) np. ArrayList lub tablicę i dodaje całą zawartość do konkretnej kolekcji.

Wypisywanie elementów

W celu wygenerowania reprezentacji tablicy należy użyć z metody toString() z klasy Arrays – w przypadku kolekcji nie jest wymagane użycie żadnych pośrednich operacji. Tworząc metodę która w wyniku swojego działania zwraca kolekcję to wykorzystując strumień wyjściowy i metodę println – System.out.println(wypiszKolekcje()) na konsoli zostanie wypisana zawartość całej kolekcji.

```
public static ArrayList<String> wypiszKolekcje() {
    ArrayList<String> lista = new ArrayList<>();
    for (int i = 0; i < 25; i++) {
        lista.add("Element kolekcji nr: " + i);
    }
    return lista;
}
```

Dostęp do poszczególnych elementów kolekcji można uzyskać metodą get(int index) – wówczas wypisanie wygląda jak poniżej:

```
public static void wypiszKilka(ArrayList<String> lista) {
    for (int i = 0; i < lista.size(); i = i + 2) {
        System.out.println(lista.get(i));
    }
}
```

Wywołanie sekwencji metod w main będzie skutkowało wypisaniem elementów o parzystych indeksach od 0 do 24.

```
wypiszKilka(wypiszKolekcje());
```



Interfejs List

Skorzystanie z interfejsu List zobowiązuje do zachowania kolejności elementów w kolekcji – uzupełnia interfejs Collection o metody pozwalające na dodawanie elementów do kolekcji lub usuwanie elementów ze środka kolekcji. Podstawowy podział to ArrayList i LinkedList:

- ArrayList – podstawowy podtyp interfejsu List, cechuje go swobodny dostęp do elementów, wolniejszy przy wstawianiu i usuwaniu elementów z listy,
- LinkedList – optymalny dostęp sekwencyjny, efektywne operacje wstawiania i usuwania elementów, swobodny dostęp działa stosunkowo wolno, bogaty zestaw funkcji.

Wybrane metody z interfejsu List przedstawiono w tabeli 7.1.

Tab.7.1 Wybrane metody interfejsu List.

Modyfikator i typ	Metoda	Opis
boolean	<i>contains(Object o)</i>	Zwraca prawdę jeśli element jest obecny w kolekcji
boolean	<i>equals(Object o)</i>	Porównuje obiekt z listą pod kątem równości
E	<i>remove(int index)</i>	Usuwa element na konkretnej pozycji z listy
boolean	<i>remove(Object o)</i>	Usuwa pierwsze wystąpienie określonego elementu z listy, o ile występuje
E	<i>set(int index, E element)</i>	Zastępuje element na konkretnej pozycji określonym elementem

Klasa Stack

Stos (ang. stack) może być interpretowany jako kolekcja typu last – in, first – out (LIFO). Wszystko co zostanie włożone na stos (operacja push) jako ostatnie, jest pierwszym które stos może opuścić (operacja pop). Funkcje stosu posiada również LinkedList i może być używana zamiast własnej klasy. Do uzyskania jedynie zachowania typowego dla stosu należy zrezygnować z dziedziczenia – uzyskana w ten sposób klasa będzie posiadała te same metody klasy LinkedList. Aby wykorzystać klasę Stack we własnym kodzie należy podać pełną nazwę pakietu lub zmienić nazwę klasy – w przeciwnym wypadku może dojść do kolizji nazw z klasą Stack z pakietu java.util.

Interfejs Set

Zbiór (ang. set) jak zostało już wspomniane nie może zawierać więcej niż jednego egzemplarza danej wartości. Próba dodania kolejnego obiektu (identycznego z istniejącym w kolekcji) zostanie zignorowana. Dzięki temu mechanizmowi w zbiorze elementy nie dublują się. Najpopularniejsze zastosowanie zbioru to przeprowadzenie testu przynależności czego wynikiem może być stwierdzenie obecności obiektu w danym zbiorze. Operacja wyszukiwania elementu w zbiorze została zoptymalizowana pod kątem szybkości. Interfejs



Set jest rozszerzenie interfejsu Collection ale nie posiada żadnych dodatkowych funkcji jak w przypadku interfejsu List – zachowuje się jednak inaczej. Jest to sztandarowy przykład pogodzenia mechanizmów dziedziczenia i polimorfizmu - inne zachowanie w zależności od potrzeb.

Interfejs Map

Mapa (ang. map) pozwala na przeprowadzenie odwzorowania obiektów na inne obiekty, co jest niezwykle istotne przy rozwiązywaniu niektórych problemów programistycznych. Przykładowo chcąc sprawdzić ilość wystąpień danej wartości pseudolosowej wygenerowanej przy użyciu klasy Random można stworzyć powiązanie (odwzorowanie) klucz – wartość, gdzie kluczem byłaby wartość wylosowana, natomiast wartością ilość wystąpień tej wartości w danej sekwencji pseudolosowej. Kontener Map może być podobnie jak tablice i inne kolekcje rozbudowywany do wielu wymiarów - wystarczy stworzyć kontener Map, którego elementami są inne kontenery Map. W zależności od tego ile wymiarów jest nam potrzebne ta sytuacja może się powtarzać, aż do ostatniego wymiaru zawierającego pożądany typ danych.

Interfejs Queue

Kolejka (ang. queue, IPA: /kju:/) to kontener typu first – in, first – out (FIFO). Wszystkie elementy są dodawane na końcu, a pobierane z początku – jak w typowej kolejce. Ten rodzaj kontenera jest bardzo przydatny w programowaniu współbieżnym, ponieważ pozwalają bezpiecznie przekazywać obiekty między zadaniami (więcej informacji o współbieżności w Laboratorium 10). Klasa LinkedList (implementuje interfejs Queue) i posiada metody odpowiednie dla zachowania kolejki i może być używana w roli kolejki.

Praca z gotowym kodem

Poniżej przedstawiono dwie klasy – pierwsza (Okno) odpowiedzialna jest za utworzenie okna aplikacji, natomiast druga (Panel) wypełnia wnętrze okna. Aktualne działanie aplikacji wygląda następująco – po uruchomieniu projektu zostaje otwarte okno z czarnym tłem. Każdorazowo przy wciśnięciu przycisku myszy w obszarze okna pojawia się nowa kula o losowym kolorze, poruszająca się w losowym kierunku z losową prędkością. Kule odbijają się od ścian i przenikają się wzajemnie. Każda kula jest dodawana do kolekcji ArrayList. Dokładne przedstawienie aplikacji z graficznym interfejsem użytkownika zostanie poruszone w laboratorium 11. Przeanalizuj poniższy kod:

```
package com.mycompany.kolekcje;

import java.awt.Dimension;
import javax.swing.JFrame;

public class Okno {

    public static void main(String[] args) {

        JFrame frame = new JFrame("Moje okno!");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```




```
        frame.getContentPane().add(new Panel());
        frame.setPreferredSize(new Dimension(800, 600));
        frame.pack();
        frame.setVisible(true);
    }
}
package com.mycompany.kolekcje;

import java.awt.Color;
import java.awt.Graphics;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.util.ArrayList;
import javax.swing.JPanel;
import javax.swing.Timer;

public class Panel extends JPanel {

    private ArrayList<Kula> listaKul;
    private int size = 20;
    private Timer timer;
    private final int DELAY = 33;
    //dla 30fps -> 1s/30 = 0,033s

    public Panel() {

        listaKul = new ArrayList<>();
        setBackground(Color.BLACK);

        addMouseListener(new Event());

        timer = new Timer(DELAY, new Event());
        timer.start();
    }

    @Override
    protected void paintComponent(Graphics g) {
        super.paintComponent(g);

        for (Kula k : listaKul) {
            g.setColor(k.color);
            g.drawOval(k.x, k.y, k.size, k.size);
        }
        g.setColor(Color.YELLOW);
        g.drawString(Integer.toString(listaKul.size()), 40, 40);
    }
}
```



```
private class Event implements MouseListener,
ActionListener {

    @Override
    public void mouseClicked(MouseEvent e) {
    }

    @Override
    public void mousePressed(MouseEvent e) {
        listaKul.add(new Kula(e.getX(), e.getY(), size));
        repaint();
    }

    @Override
    public void mouseReleased(MouseEvent e) {
    }

    @Override
    public void mouseEntered(MouseEvent e) {
    }

    @Override
    public void mouseExited(MouseEvent e) {
    }

    @Override
    public void actionPerformed(ActionEvent e) {
        for (Kula k : listaKul) {
            k.update();
        }
        repaint();
    }
}

private class Kula {

    public int x, y, size, xspeed, yspeed;
    public Color color;
    private final int MAX_SPEED = 5;

    public Kula(int x, int y, int size) {
        this.x = x;
        this.y = y;
        this.size = size;
    }
}
```



```
        color = new Color((float) Math.random(), (float)
Math.random(), (float) Math.random());
        xspeed = (int) (Math.random() * MAX_SPEED * 2 -
MAX_SPEED);
        yspeed = (int) (Math.random() * MAX_SPEED * 2 -
MAX_SPEED);
    }

    public void update() {
        x += xspeed;
        y += yspeed;

        if (x <= 0 || x >= getWidth()) {
            xspeed = -xspeed;
        }
        if (y <= 0 || y >= getHeight()) {
            yspeed = -yspeed;
        }
    }
}
```

Wykonaj poniższe zadania.

Zadanie 7.1. Zmodyfikuj kod zgodnie z podpunktami

Nowo powstałe kule mają prędkość wylosowaną z zakresu $\langle -\text{MAX_SPEED}, \text{MAX_SPEED} \rangle$, dla stałej $\text{MAX_SPEED} = 5$ oznacza to zakres $\langle -5, 5 \rangle$. Istnieje możliwość wylosowania prędkości 0 (w osi x, y lub obu).

- Zmodyfikuj kod aby wyeliminować możliwość utworzenia kuli o składowej prędkości równej 0.

Kule odbijają się od granicy okna i przenikają się wzajemnie.

- Zmodyfikuj kod dodając kolizję między kulami (można wzorować się na zderzeniach sprężystych niecentralnych).

Kule mają stały rozmiar.

- Dodaj możliwość zmiany rozmiaru kolejnych kul przy pomocy rolki myszy (kule dodane przed obrotem rolki zachowują rozmiar).
- Sprawdź działanie aplikacji dla innych wartości stałej DELAY, np. 16, 42, 100 (odpowiednio dla 60, 24 i 10 fps).
- Dodaj kilka zdarzeń w zależności od działania myszki np. wejście/wyjście kursora w obszar okna aplikacji – włączenie/wyłączenie ruchu kul.

Zadanie 7.2. Ciągłe pobieranie

Napisz program pobierający od użytkownika liczby całkowite dopóki nie zostanie wprowadzona liczba 0. Po zakończeniu pobierania podaj ile wprowadzono liczb, ich sumę, oraz iloczyn (nie wliczaj 0).



Fundusze Europejskie
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny



Zadanie 7.3. Sortowanie

Napisz program pobierający od użytkownika liczby całkowite dopóki wartość bezwzględna z ich sumy lub iloczynu nie przekroczy odpowiednio 250 lub 3000000. Po zakończeniu pobierania wprowadzone liczby mają zostać posortowane od największej do najmniejszej i wyświetlone.

Zadanie 7.4. Korekcja sumy kolekcji

Napisz program pobierający od użytkownika liczby całkowite dopóki ich suma nie będzie równa 64. Jeżeli wartość zostanie przekroczona największa liczba z wprowadzonych ma zostać usunięta i pobieranie ma odbywać się dalej. Wykorzystaj sortowanie.

Zadanie 7.5. Korekcja iloczynu kolekcji

Napisz program pobierający od użytkownika liczby całkowite dopóki ich iloczyn nie będzie równy 256. Jeżeli wartość zostanie przekroczona najmniejsza liczba z wprowadzonych ma zostać usunięta i pobieranie ma odbywać się dalej. Wykorzystaj sortowanie.

LABORATORIUM 8. WYRAŻENIA LAMBDA W JĘZYKU JAVA.

Cel laboratorium:

Zapoznanie ze składnią i zastosowaniem wyrażeń lambda w języku Java.

Zakres tematyczny zajęć:

- Wyrażenie lambda.
- Interfejs funkcyjny.

Pytania kontrolne:

1. Jak wygląda składnia wyrażenia lambda?
2. Co to jest interfejs funkcyjny?
3. Czy w języku Java można programować funkcyjnie?
4. Do czego wykorzystuje się wyrażenia lambda?

Wyrażenie lambda

Standardowo chcąc przypisać wartość zwróconą przez metodę należy w pierwszej kolejności odpowiednio zdefiniować metodę, a następnie wywołać ją wykorzystując operator przypisania „=”. W przypadku, gdy oczekiwany argument ma typ interfejsu funkcyjnego można wykorzystać wyrażenie lambda (ang. Lambda Expression). Wyrażenie lambda umożliwia przekazanie w jaki sposób ma się wykonać coś, co jest (tylko) parametrem metody. Interfejs funkcyjny posiada wyłącznie jedną metodę abstrakcyjną. Przykładem interfejsu funkcyjnego jest **ActionListener** z pakietu Swing. Przeanalizuj poniższy kod:

```
public interface TestowyInterfejs {  
  
    void abstrakcyjnaMetodaX();  
  
}
```

Składnia wyrażenia lambda wygląda następująco:

```
(lista_parametrów) -> {ciało_wyrażenia}
```

Lista parametrów analogicznie do „tradycyjnej” metody zawiera parametry przekazywane do wnętrza (ciała) wyrażenia lambda. Analogicznie do zwykłych metod lista może być pusta (). Jeżeli wyrażenie ma tylko jeden parametr to nawiasy okrągłe są opcjonalne. Ciałem wyrażenia jest kod opisujący jego działanie – jeżeli zajmuje jedną linijkę można zrezygnować z nawiasów klamrowych {}.

TestowyInterfejs jest szablonowym interfejsem funkcyjnym posiadającym wyłącznie jedną metodę abstrakcyjną. Jeżeli np. w klasie Main zostanie zdefiniowana metoda:

```
public static void metodaY(int x, int y, TestowyInterfejs ti)  
{
```



Fundusze Europejskie
Wiedza Edukacja Rozwój



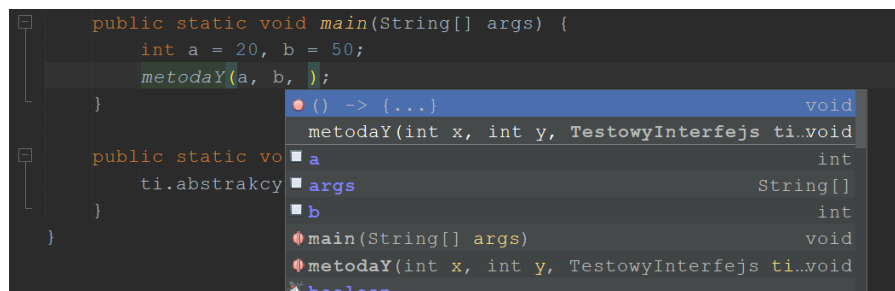
Rzeczpospolita
Polska

Unia Europejska
Europejski Fundusz Społeczny



```
ti.abstrakcyjnaMetodaX();
}
```

to wywołując metodę *metodaY()* i omijając jawny mechanizm implementacji interfejsu można przekazać w miejsce parametru (*TestowyInterfejs*) wyrażenie lambda (np. używając skrótu CTRL + Spacja). Wówczas definicja wyrażenia lambda zostanie podana zgodnie ze składnią przedstawioną powyżej.



Rys.8.1. Wykorzystanie wyrażenia lambda.

```
public static void main(String[] args) {
    int a = 20, b = 50;
    metodaY(a, b, () -> {
        System.out.println(a * b);
    });
}
```

Wynikiem działania takiego prostego programu jest wypisanie w konsoli liczby 1000. Przeanalizuj kod jeszcze raz. W dużym rozwinięciu nastąpiło przekazanie metody jako parametru do metody. Programowanie funkcyjne nie było możliwe w Javie aż do JDK8.

Wyrażenia lambda usprawniają również biblioteki kolekcji, ułatwiając iterację, filtrowanie i wyodrębnianie danych z kolekcji. Ponadto nowe funkcje współbieżności poprawiają wydajność w środowiskach wielordzeniowych.

Zadanie 1.1. Wypisanie listy

Napisz interfejs zawierający jedną metodę abstrakcyjną przyjmującą jako parametr łańcuch znaków. Stwórz listę (np. *ArrayList*) zawierającą spis przedmiotów do zaliczenia (klasa *String*). Napisz metodę przyjmującą jako parametry: listę łańcuchów znaków oraz stworzony interfejs – metoda ma wypisać wszystkie elementy listy. Użyj wyrażenia lambda przy wywołaniu metody.

Zadanie 1.2. Sortowanie tablicy

Wykorzystując interfejs funkcyjny *Comparator<T>* napisz metodę sortującą tablicę liczb całkowitych. Metoda *compare(T o1, T o2)* zwraca liczbę całkowitą zgodnie z funkcją signum (ujemną, zero lub dodatnią) jeśli pierwszy argument jest odpowiednio (mniejszy, równy, większy) od drugiego. Użyj wyrażenia lambda.



Zadanie 1.3. Sortowanie listy

Wykorzystując interfejs funkcyjny *Comparator<T>* napisz metodę sortującą listę łańcuchów znaków w zależności od długości. Metoda `compare(T o1, T o2)` zwraca liczbę całkowitą zgodnie z funkcją `signum` (ujemną, zero lub dodatnią) jeśli pierwszy argument jest odpowiednio (mniejszy, równy, większy) od drugiego. Użyj wyrażenia `lambda`.

LABORATORIUM 9. OPERACJE WEJŚCIA – WYJŚCIA, OBSŁUGA PLIKÓW.

Cel laboratorium:

Zapoznanie z operacjami wejścia – wyjścia, poznanie klas do obsługi wejścia oraz wyjścia. Obsługa plików.

Zakres tematyczny zajęć:

- Operacje wejścia wyjścia.
- Klasy obsługujące strumienie.
- Zapis do plików.
- Odczyt z plików.
- Standardowe wejście – wyjście.

Pytania kontrolne:

1. Co jest źródłem wejścia, a co wyjścia.
2. Jakie znasz klasy do obsługi plików – czym się różnią?
3. W jaki sposób można odczytać zawartość pliku?
4. Jak można „poruszać się” po pliku?
5. Czy można zmienić standardowe wyjście?
6. Czy i jak często można zmieniać standardowe wejście?

Wejście - wyjście

Różne źródła wejścia i wyjścia tj. konsola, pliki czy połączenie sieciowe wymagają uwzględnienia wielu sposobów na komunikację – tryb sekwencyjny, swobodny dostęp, wczytywanie wierszy, słów, znaków czy bitów. W języku Java biblioteki zostały przepełnione bardzo dużą liczbą klas – początkowo może to działać odstraszaюще (ironicznie pierwotne projekty zakładały ograniczenie liczby klas). Początkowe wersje biblioteki wejścia – wyjścia były poszerzane w miarę rozrostu JDK o istotne zmiany. W rezultacie wiele klas wymaga dobrego poznania zanim zrozumie się działanie biblioteki wejścia – wyjścia na tyle, żeby używać jej właściwie. W tym laboratorium zostanie przedstawiona różnorodność klas odpowiadających za operacje wejścia – wyjścia.

Klasa File

Bardzo rzadko zdarza się, że nazwa klasy nie jest poprawnie skonstruowana. W tym przypadku klasa **File** nie odnosi się bezpośrednio do pliku tylko do nazwy pliku lub zbioru plików zebranych w jednym katalogu. Do odnalezienia zbioru plików można wykorzystać metodę **list()**, która zwraca tablicę obiektów klasy String. Liczba elementów (nazw plików) jest ustalona więc tablica jest lepszym wyborem niż kontener (np. ArrayList). Dla kolejnych katalogów wystarczy utworzyć nowy obiekt File.

Tworzenie katalogów

Wykorzystując klasę `File` można również utworzyć katalog lub całą ścieżkę jeśli nie istnieje. Dodatkowo jest możliwość wglądu we właściwości plików tj. rozmiar, prawa dostępu, data ostatniej modyfikacji oraz możliwość usunięcia pliku.

Strumień

Strumień (ang. stream) jest abstrakcyjnym pojęciem reprezentującym dowolne źródło wejściowe lub wyjściowe danych jako obiektu zdolnego do wysyłania lub odbierania porcji danych. To co dzieje się z danymi wewnątrz urządzenia wejścia – wyjścia jest ukryte. W języku Java biblioteki wejścia – wyjścia są podzielone według wejścia (ang. input) oraz wyjścia (ang. output). Wykorzystując mechanizm dziedziczenia wszystkie klasy rozszerzone przy użyciu klas ***InputStream*** lub ***Reader*** mają metody odczytu jednego bajta lub tablicy bajtów – ***read()***. Analogicznie klasy rozszerzające ***OutputStream*** lub ***Writer*** posiadają metody do zapisu jednego bajta lub tablicy bajtów – ***write()***.

W rzeczywistości nie zachodzi potrzeba użycia podanych wyżej metod bezpośrednio – wykorzystują je klasy podrzędne dostarczając jednocześnie bardziej użyteczny interfejs. Obiekt strumieniowy tworzy się zazwyczaj poprzez nawarstwienie kilku obiektów różnych klas. Pomocny jest tutaj podział na kategorie wejścia – dziedziczące po `InputStream` i wyjścia – dziedziczące po `OutputStream`.

Klas pochodnych po wyżej wymienionych jest wiele i w zależności od źródła danych należy wykorzystać odpowiednie z nich. Lista oficjalnych klas wykorzystujących `InputStream` lub `OutputStream` jest dostępna w dokumentacji na pozycji znane podklasy (Direct Known Subclasses).

Klasy Reader i Writer

Wraz z rozwojem bibliotek Javy i pojawieniem się „nowszych” (pochodzących z JDK 1.1) klas `Reader` i `Writer` pojawiło się pytanie czy „stare” klasy `InputStream` i `OutputStream` będą dalej wykorzystywane. Klasy `Reader` i `Writer` zapewniają zgodną z Unicode (znakową) obsługę wejścia – wyjścia. Niejednokrotnie stosowanie niektórych funkcji oryginalnej biblioteki jest niezbędne. Użycie klas w hierarchii „bajtowej” (`InputStream/OutputStream`) wraz z klasami w hierarchii „znakowej” (`Reader/Writer`) jest możliwe przy użyciu klas pośredniczących odpowiednio: `InputStreamReader` konwertuje obiekt `InputStream` na `Reader`, natomiast `OutputStreamWriter` konwertuje `OutputStream` na `Writer`.

Celem wprowadzenia klas `Reader` i `Writer` były dążenie do umożliwienia obsługi standardu międzynarodowego. Stara hierarchia obsługiwała strumienie 8-bajtowe i nie dostarczała metod do obsługi strumieni 16-bitowych (typ `char` jest właśnie 16-bitowy znakiem Unicode). Poza tymi udogodnieniami zaprojektowano, by operacje wykonywane przy użyciu nowych klas były szybsze.

Klasa RandomAccessFile

Klasa ***RandomAccessFile*** jest wykorzystywana przy obsłudze plików zawierających rekordy o znanym rozmiarze, tak by można było przemieszczać się między nimi za pomocą metody ***seek()***, czytać je lub edytować. Nie muszą mieć tego samego rozmiaru, wystarczy jeśli będzie się dało określić jak są duże i gdzie znajdują się w pliku. Klasa ta nie jest częścią hierarchii `InputStream` lub `OutputStream` – nie ma z nimi nic wspólnego poza implementacją



(podobnie jak `DataInputStream` i `DataOutputStream`) interfejsów `DataInput` i `DataOutput`. Jest to klasa niezależna, napisana od zera ze wszystkimi metodami. Zasadniczo klasa jest połączeniem `DataInputStream` i `DataOutputStream` z dodatkowymi metodami do poruszania się po pliku. Należą do nich metody `getFilePointer()`, `seek()`, `length()`. Poza tym konstruktor wymaga drugiego argumentu (poza nazwą) wskazującego na dostęp do pliku: swobodny odczyt („r”), czytanie i dopisywanie („rw”). Brak jest obsługi wyłącznie zapisu. Tylko klasa `RandomAccessFile` posiada gotowe metody do przeszukiwania plików.

Zapis i odczyt plików

Zapis do pliku może być zrealizowany z wykorzystaniem klasy `FileWriter` jak pokazano na poniższym przykładzie. Należy zwrócić uwagę na konieczność obsługi wyjątku klasy `IOException`.

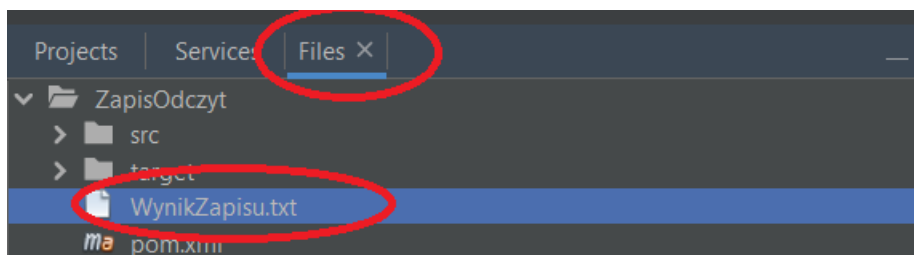
```
public static void prostyZapis() {  
  
    File file = new File("WynikZapisu.txt");  
    try {  
        FileWriter fileWriter = new FileWriter(file,  
true);  
        for (int i = 0; i < 10; i++) {  
            fileWriter.append("To jest dopisana linijka  
tekstu nr: " + i + "\n");  
        }  
        fileWriter.close();  
    } catch (IOException ex) {  
        System.err.println(ex.getCause());  
    }  
}
```

Najpierw tworzona jest instancja klasy `File` (można pominąć to i niejawnie utworzyć obiekt `File` w konstruktorze `FileWriter`). Następnym krokiem jest wyłapanie ewentualnych wyjątków blokiem `try – catch (IOException)`. Tworzony obiekt `FileWriter` posiada argument wskazujący na plik oraz opcjonalnie wartość `true` oznaczającą możliwość dopisywania kolejnych linii do pliku – drugie uruchomienie metody `prostyZapis()` dopisze do już istniejącego pliku ciąg znaków (bez drugiego argumentu plik zostałby nadpisany od początku). Po pętli dopisującej kolejne linie następuje zamknięcie strumienia – należy zawsze o tym pamiętać. Zmienić się może miejsce wywołania metody `close()` w zależności od potrzeb. Brak zamknięcia może skutkować:

- nie zapisaniem danych
- przekroczeniem limitu jednocześnie otwartych plików (każdy system operacyjny ma limit jednocześnie otwartych plików)

Po wywołaniu metody plik z zapisaną zawartością można znaleźć w katalogu projektu (o ile nie została podana ścieżka) – w środowisku NetBeans zakładka Files (drzewo projektu).





Rys. 9.1. Lokalizacja pliku po zapisie.

Odczyt z pliku również może być realizowany przy użyciu wielu klas. `FileReader` oferuje dostęp znak po znaku, `BufferedReader` linijka po linijce. Przy tworzeniu instancji parametrami konstruktora mogą być obiekty tworzone na bieżąco jak w poniższym przykładzie.

```
public static void prostyOdczyt() {  
  
    try {  
        BufferedReader bufferedReader = new  
BufferedReader(new FileReader(new File("WynikZapisu.txt")));  
        String linia = null;  
  
        while ((linia = bufferedReader.readLine()) !=  
null) {  
            System.out.println(linia);  
        }  
        bufferedReader.close();  
    } catch (FileNotFoundException ex) {  
        System.out.println("Pliku nie odnaleziono!");  
        System.err.println(ex.getCause());  
    } catch (IOException ex) {  
        System.out.println("Błąd odczytu pliku  
spowodowany:");  
        System.err.println(ex.getCause());  
    }  
  
}
```

Odczyt linijka po linijce najprościej zaimplementować przy użyciu klasy `BufferedReader`. Przy otwieraniu pliku należy zawsze uwzględnić jego nieistnienie lub błąd w podawanej nazwie (literówka) dlatego należy wyłapywać wyjątek klasy `FileNotFoundException`. W dalszej części w pętli `while` która działa dopóki plik nie zakończy się (koniec pliku oznaczony jest nullem) odczytywana jest każda kolejna linijka. Po odczycie obowiązkowo należy zamknąć strumień danych.

Przedstawione przykłady nie są jedynymi słusznymi. Istnieje wiele klas przy użyciu których możliwe jest odczytanie pliku (np. `Scanner`) i należy korzystać z nich w zależności od sytuacji.



Przekierowanie standardowego wejścia – wyjścia

Standardowa klasa Javy System pozwala na przekierowanie strumieni wejścia, wyjścia oraz wyjścia błędu przy użyciu metod statycznych:

- `setIn(InputStream)`
- `setOut(PrintStream)`
- `setErr(PrintStream)`

Jest to wygodne jeżeli na ekran kierowanych jest bardzo dużo informacji i nie jest możliwe przeczytanie wszystkiego. Przekierowanie wejścia może być pomocne dla programów wykorzystujących tradycyjną konsolę (wiersz poleceń) przy wielokrotnym testowaniu danej sekwencji. Nowym standardowym wejściem – wyjściem zamiast np. konsoli może być plik lub grupa plików. W trakcie działania programu takie przekierowanie może zachodzić wiele razy. W tym przykładzie potwierdza się sens istnienia klas `InputStream` i `OutputStream`, ponieważ wejście – wyjście działa na strumieniach bajtów (a nie znaków) – klasy `Reader` i `Writer`, mimo że są nowsze nie miałyby tu zastosowania.

Sterowanie procesami zewnętrznymi

Niektóre wyzwania programistyczne wymagają czasami uruchomienia z wnętrza programu innego programu systemu operacyjnego lub wysłania do danego programu danych wejściowych i przechwycenia wyjścia. W języku Java istnieją klasy zapewniające odpowiednią obsługę takich problemów. Celem tego laboratorium nie jest uruchamianie zewnętrznych aplikacji jednak warto zdawać sobie sprawę z możliwości. Pomocne mogą okazać się klasy `Runtime` i `ProcessBuilder`.

Zadanie 9.1. Zapis kolizji

Wróć do zadania 7.1 dotyczącego zderzenia kul. Zmodyfikuj/dodaj kod tak aby każda kolizja została zapisywana do pliku – niezbędne informacje to współrzędne kul w momencie zderzenia oraz rozmiar (przy dużej liczbie kul rozważ buforowanie i zapis np. co 30 wykrytych kolizji).

Zadanie 9.2. Odczyt kolizji

Wykorzystaj kod z zadania 7.1 oraz 9.1 – odczytaj plik z zarejestrowanymi kolizjami. Stwórz okno analogiczne do zadania 7.1 ale bez animacji. Użyj odczytanych danych do zaznaczenia miejsc kolizji (użyj metody rysującej `drawOval()`).

Zadanie 9.3. Szyfr Cezara – plik

Wykorzystaj kod z zadania 3.4 – napisz analogiczną wersję szyfrującą i deszyfrującą tekst w plikach. Wykorzystaj poniższy tekst do stworzenia pliku tekstowego który będzie szyfrowany i deszyfrowany.

Treść pliku:

Jan Brzechwa
Entliczek-pentliczek



Fundusze Europejskie
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny



Entliczek-pentliczek, czerwony stoliczek,
A na tym stoliczku pleciony koszyczek,

W koszyczku jabłuszko, w jabłuszku robaczek,
A na tym robaczku zielony kubraczek.

Powiada robaczek: "I dziadek, i babka,
I ojciec, i matka jadali wciąż jabłka,

A ja już nie mogę! Już dosyć! Już basta!
Mam chęć na befsztyczek!" I poszedł do miasta.

Szedł tydzień, a jednak nie zmienił zamiaru,
Gdy znalazł się w mieście, poleciał do baru.

Są w barach - wiadomo - zwyczaje utarte:
Podchodzi doń kelner, podaje mu kartę,

A w karcie - okropność! - przyznacie to sami:
Jest zupa jabłkowa i knedle z jabłkami,

Duszone są jabłka, pieczone są jabłka
I z jabłek szarlotka, i komput [placek], i babka!

No, widzisz, robaczku! I gdzie twój befsztyczek?
Entliczek-pentliczek, czerwony stoliczek.



LABORATORIUM 10. WSPÓLBIEŻNOŚĆ W JĘZYKU JAVA.

Cel laboratorium:

Wprowadzenie pojęcia współbieżności, tworzenie wątków.

Zakres tematyczny zajęć:

- Metody tworzenia wątków.
- Stany wątków.
- Obsługa wątków.

Pytania kontrolne:

1. Co to jest wątek?
2. W jaki sposób można stworzyć wątek w języku Java?
3. W jakim celu stosuje się wątki?
4. Czy program może zakończyć pracę jeśli jakieś wątki nie ukończyły swoich zadań?
5. W jaki sposób rozwiązać problem dostępu do zasobów współdzielonych?
6. Czy można ustalić który wątek będzie się wykonywał jako pierwszy?

Współbieżność

Do tego momentu wszystkie programy popelnione w ramach laboratorium języka Java miała charakter sekwencyjny – wszystkie instrukcje w programach wykonywały się jedna za drugą. W przypadku niektórych problemów (wyzwań) programistycznych wygodniejszym podejściem jest uruchomienie kilku części programu równolegle. Takie rozwiązanie może przyczynić się do znacznego przyspieszenia wykonywania programu.

Wątek to pojedynczy, sekwencyjny przepływ sterowania w obrębie procesu (programu). W języku Java klasą odpowiedzialną za wątki jest klasa **Thread**. Wątek może być więc niezależnym stosem wywołań lub obiektem klasy Thread, w zależności od kontekstu należy rozróżnić te określenia. Każda uruchamiająca się aplikacja Javy uruchamia jednocześnie wątek główny – na samym dole stosu wywołań jest metoda main(). Odpowiedzialność za uruchomienie wątku głównego (i innych) ciąży na JVM. Nowe wątki mogą być uruchamiane na podstawie kodu – programowo.

Tworząc kilka wątków można uzyskać jednoczesne (szybsze) wykonywanie kilku różnych zadań. Z tego powodu bardzo skomplikowane zadania należy rozbijać na mniejsze, a te z kolei mogą być przydzielane nowym wątkom.

Podstawowym sposobem przydzielenia zadania jest wykorzystanie metody run(). Metoda ta jest dostępna na dwa sposoby:

1. Implementacja interfejsu **Runnable** (ewentualnie **Callable**) i wiążący się z tym import metod w tym (wyłącznie) metody run() (odpowiednio call()),
2. Rozszerzenie klasy Thread poprzez mechanizm dziedziczenia i wykorzystanie metody run() z klasy bazowej.

Różnica między interfejsami Runnable i Callable jest taka, że w wyniku działania metody run() nie jest zwracana żadna wartość, natomiast w przypadku call() jest. Dane zadanie, które ma zostać wykonane w oddzielnym wątku musi zostać dodane do metody run(). Nowy wątek należy stworzyć jako obiekt klasy Thread. Po wywołaniu na nim metody start(), metoda run



przypisana do danej klasy (implementującej interfejs lub rozszerzającej klasę) zostanie wywołana.

Stan wątków

Nowy wątek może występować w trzech stanach: nowy, uruchamialny i uruchomiony. Pierwszy przypadek opisuje sytuację, kiedy wątek został utworzony ale nie uruchomiony:

```
Thread watek = new Thread();
```

Po wywołaniu metody:

```
watek.start();
```

wątek przechodzi do stanu określonego jako uruchamialny – oznacza to, że wątek jest gotowy do działania i czeka, aż zostanie wybrany i rozpocznie pracę. W tym momencie nowy stos używany przez ten wątek już istnieje. Ostatnim stanem wątku jest uruchomiony – jest aktualnie wykonywanym wątkiem. Decyzję o uruchomieniu podejmuje mechanizm zarządzający wątkami. Można wpłynąć na decyzję o uruchomieniu ale nie można wymusić zmiany stanu z „uruchamialny” na „uruchomiony”. Jeśli wątek jest w stanie uruchomiony to tylko ten jeden wątek posiada aktywny stos wywołań.

Dodatkowo jeśli wątek będzie uznany za uruchamialny (wywołana metoda start()) będzie mógł zmienić stan na uruchomiony, uruchamialny lub zablokowany (uśpiony, czekający na zakończenie operacji wykonywanych przez inny wątek, czekający na pojawienie się danych w strumieniu, czekający na usunięcie blokady).

Usypianie wątków

W celu kontrolowania zachowania wątku można wykorzystać metodę sleep(), która zawiesza wykonanie wątku na określony czas. Wywołanie metody sleep() może spowodować zgłoszenie wyjątku InterruptedException – wyjątki nie są propagowane do wątku metody main należy je obsługiwać lokalnie w obrębie danego wątku.

Przełączanie wątków

Metoda yield() jest sugestią dla planisty wątków (ang. thread scheduler), że można przełączyć procesor do innych zadań (bo np. w danym wątku zakończono obliczenia). Wywołanie yield() jest opcjonalne. Wywołanie metody yield() sugeruje możliwość uruchomienia innego wątku o tym samym priorytecie. Kolejność wykonywania grupy wątków nie jest określona jeśli na wznowienie oczekuje kilka zablokowanych wątków to planista w pierwszej kolejności będzie wznowiał te o wyższym priorytecie. Priorytet nie może być przyczyną zakleszczenia, wpływa jedynie na częstość wykonywania. Ustawianie i odczytywanie priorytetu może być wykonane przy pomocy metod setPriority() i getPriority().

Wątki – demony

Demonem (ang. daemon) nazywany jest wątek, który powinien zapewnić ogólne usługi w tle programu w trakcie jego działania ale nie jest bezpośrednio związany z główną częścią programu. Kiedy wszystkie wątki nie będące demonami zakończą pracę – program



również. W odwrotnej sytuacji, jeśli istnieją wątki nie będące demonami to program się nie zakończy. W celu zdefiniowania wątku jako demona należy użyć metody `setDaemon()` zanim wątek zostanie uruchomiony.

Łączenie wątków

Wątek może wywołać metodę `join()` innego wątku, co spowoduje, że realizacja wątku wywołującego zostanie wznowiona dopiero po zakończeniu wątku wywoływanego. Metodę `join()` można również wywołać z argumentem określającym czas oczekiwania – dzięki temu nawet jeśli wątek docelowy nie zostanie zakończony w podanym czasie to wywołanie metody `join()` i tak się zakończy. W celu przerwania działania metody `join()` można wywołać metodę `interrupt()` wątku wywołującego – należy zastosować blok `try – catch`.

Współdzielenie zasobów

W przypadku programów wielowątkowych istnieje możliwość, że dwa lub więcej wątków będzie „chciało” użyć tego samego zasobu równocześnie. Należy zapobiec takiej kolizji przy dostępie do zasobów. W celu rozwiązania problemu kolizji wykorzystywane jest szeregowanie dostępu do zasobów wspólnych. Oznacza to, że w danej chwili tylko jedno zadanie może mieć dostęp do wspólnego zasobu. Wykorzystuje się do tego klauzulę blokującą – **synchronized**. Gdy wątek chce wykonać fragment kodu oznaczonego słowem kluczowym `synchronized`, mechanizm zabezpieczający sprawdza czy blokada jest wolna, a następnie zakłada blokadę, wykonuje kod i zwalnia blokadę.

Deklaracja metody wygląda następująco:

```
synchronized void metoda() {  
    //instrukcje  
}
```

Współdziałanie wątków

Wykonując wiele zadań w obrębie jednego programu należy pamiętać o sytuacjach niebezpiecznych jak np. jednoczesny dostęp do wspólnych zasobów. Z drugiej strony należy wykorzystać wszystkie istniejące mechanizmy umożliwiające współpracę między wątkami. Do takich mechanizmów należy wymiana z potwierdzeniem – polega to na zawieszeniu wykonywania zadania do czasu zajścia określonych okoliczności zewnętrznych, które umożliwiają dalsze wykonywanie zadania. Wymiana komunikatów z potwierdzeniem jest możliwa dzięki metodom `wait()` i `notifyAll()` z klasy `Object`. Jej uzupełnieniem jest klasa `Condition` z metodami `await()` i `signal()`.

Metoda `wait()` pozwala na zawieszenie wykonania zadania w oczekiwaniu na zmianę warunków jego wykonania pozostających poza kontrolą wątku. Zmiana warunków jest sygnalizowana metodami `notify()` lub `notifyAll()` – informują one o zajściu jakichś zmian. Wówczas przeprowadzane jest wybudzanie wątku i sprawdzanie czy okoliczności pozwalają na podjęcie dalszego wykonywania zadania. Należy rozróżnić, że metody `sleep()` i `yield()` nie zwalniają blokady obiektu, natomiast wywołanie metody `wait()` w obrębie metody synchronizowanej wymusza zawieszenie wątku i zwolnienie blokady danego obiektu. Pozwala to innym metodom synchronizowanym w wątku mogą być wywołane podczas oczekiwania. W klasie `Object` występują trzy wersje metody `wait()` – pierwsza nie przyjmuje argumentów i działa bez ograniczenia czasowego do momentu wywołania metody `notify()`



lub `notifyAll()`. Druga i trzecia przyjmują argumenty w milisekundach i wymuszają wstrzymanie wątku na określony czas – analogicznie do metody `sleep()`.

Zadanie 10.1. Losowanie liczb w wielu wątkach

Napisz klasę implementującą interfejs `Runnable` z metodą losującą liczby całkowite z zakresu $<0,100>$. Wywołaj metodę w kilku wątkach. Czy wylosowane liczby są takie same? Czy możliwa jest modyfikacja zapewniająca stuprocentowo, że wynik będzie identyczny?

Zadanie 10.2. Konsument i producent

Rozwiąż modelowy problem jednego konsumenta i jednego producenta za pomocą metod `wait()` i `notifyAll()`. Producent nie może przepełnić bufora konsumenta, więc nie może produkować szybciej niż konsument konsumuje. Jeśli z kolei konsument jest szybszy w konsumowaniu zadbaj o to aby w żadnym razie nie pobrał tych samych danych więcej niż raz. Nie przyjmuj żadnych założeń dotyczących względnej szybkości wykonywania zadań producenta i konsumenta.

Zadanie 10.3. Automat do naleśników

Napisz program symulujący działanie maszyny wykonującej trzy zadania: smażenie naleśników, smarowanie ich dżemem, zwijanie w rulon. Naleśniki mają być przekazywane jak na taśmie produkcyjnej do kolejnych etapów. Wykorzystaj kolejkę synchronizowaną. Każdy konsumowany naleśnik musi być usmażony, posmarowany dżemem i zwinięty w rulon.



LABORATORIUM 11. TWORZENIE APLIKACJI Z GRAFICZNYM INTERFEJSEM UŻYTKOWNIKA Z OBSŁUGĄ ZDARZEŃ.

Cel laboratorium:

Zapoznanie z aplikacjami opartymi o graficzny interfejs użytkownika. Skład pakietu Swing. Nauka tworzenia aplikacji GUI.

Zakres tematyczny zajęć:

- Graficzny interfejs użytkownika.
- Komponenty.
- Zdarzenia.
- Metody nasłuchujące.

Pytania kontrolne:

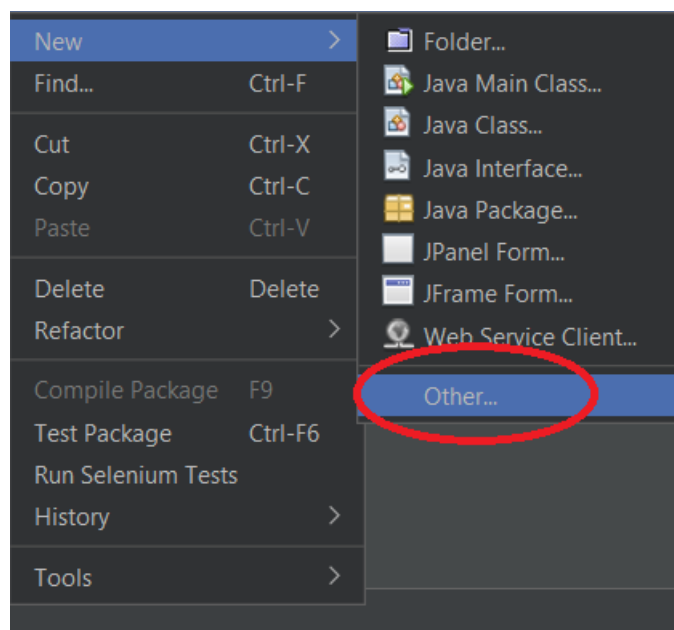
1. Rozwiń skrót GUI.
2. Czym jest Swing?
3. Czym są komponenty?
4. Skąd wiadomo, że użytkownik wykonał działanie (np. wcisnął przycisk)?
5. Czy dany komponent może obsługiwać tylko jeden rodzaj zdarzeń w danym programie?

Graficzny interfejs użytkownika

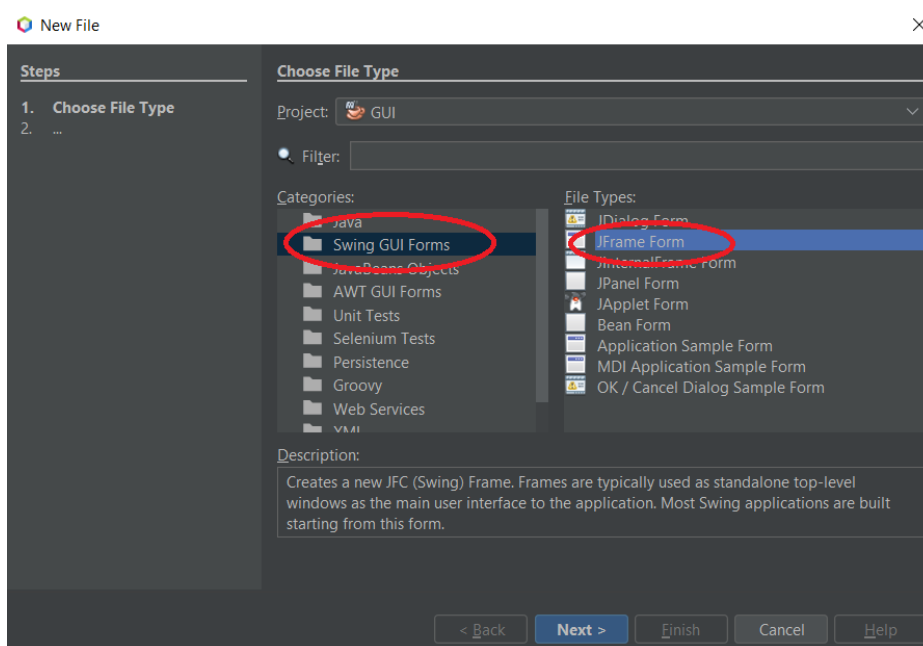
Graficzny interfejs użytkownika (ang. Graphical User Interface, GUI) jest formą prezentacji informacji oraz sposobem na interakcje z użytkownikiem. Pakietem odpowiadającym za GUI w języku Java w przeszłości był AWT (Abstract Window Toolkit – do JDK 1.2), natomiast aktualnie jest Swing (od JDK 1.2) oraz JavaFX (od JDK 7). JavaFX jest aktualnie rekomendowanym pakietem wspierającym aplikacje wyposażone w GUI. Jednak początkowe laboratorium zakłada zapoznanie się z pakietem Swing – kolejny pakiet (JavaFX) zostanie wprowadzony na poziomie zaawansowanym. Ta część laboratoryjna przedstawia zagadnienia dotyczące wyłącznie pakietu Swing. Środowisko NetBeans posiada wiele udogodnień jeśli chodzi o tworzenie aplikacji GUI tj. gotowa paleta komponentów, okno właściwości, okno projektowania i edytor kodu. Poniżej przedstawiono instrukcję krok po kroku jak stworzyć aplikację metodą drag&drop.

Pierwszym krokiem po utworzeniu projektu jest stworzenie klasy z kategorii Swing GUI Form – JFrame Form (rys.11.1 – 11.2). Tak utworzona klasa będzie wyposażona we wszystkie udogodnienia dostarczone przez środowisko do rozwijania aplikacji. Następnie należy wybrać nazwę dla klasy i zakończyć przyciskiem Finish (rys. 11.3). W wyniku tej sekwencji działań otrzymane okno powinno wyglądać jak na rysunku 11.4. Czerwonym kolorem (1) zaznaczono przełącznik do edytora kodu. Zielonym kolorem (2) zaznaczono przełącznik do okna projektowania. Niebieskim kolorem (3) zaznaczono paletę komponentów Swing. Szarym kolorem (4) oznaczono okno właściwości komponentu – aktualnie JFrame. Kolorem żółtym (5) oznaczono widok projektowanej aplikacji. Metodą drag&drop należy umieścić pożądane komponenty z palety (3) w konkretnym miejscu okna aplikacji (5). Zmiana właściwości każdego komponentu jest możliwa przy użyciu okna właściwości.

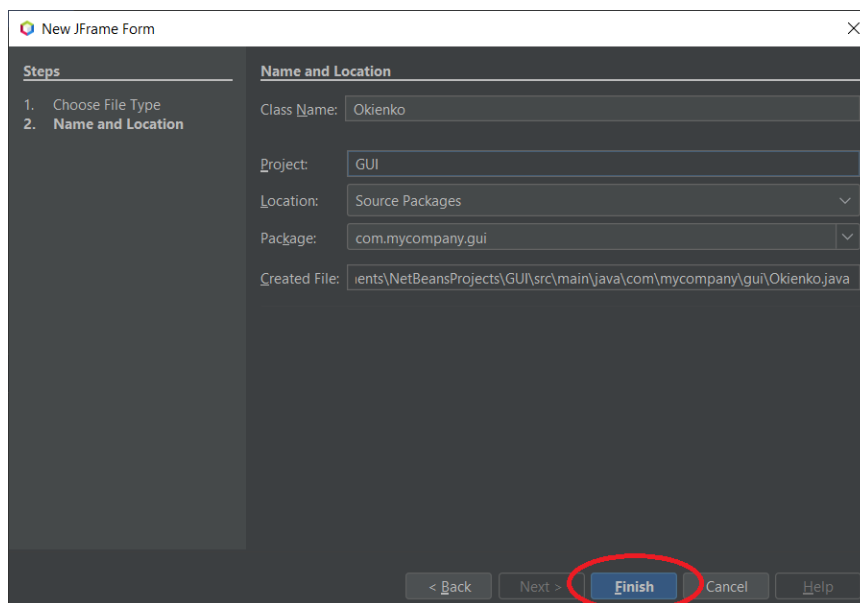




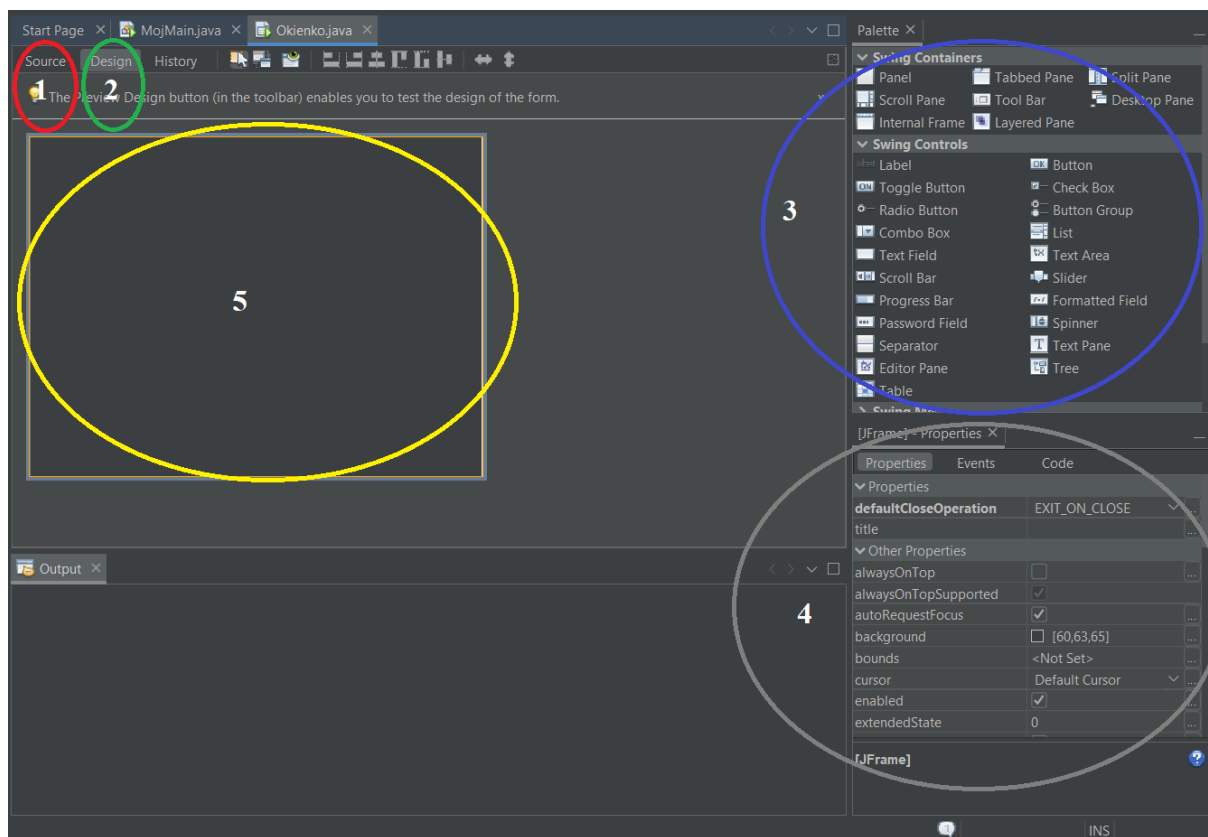
Rys.11.1. Tworzenie klasy opartej o Swing.



Rys.11.2. Wybór kategorii klasy.



Rys.11.3. Nadanie nazwy i zakończenie.

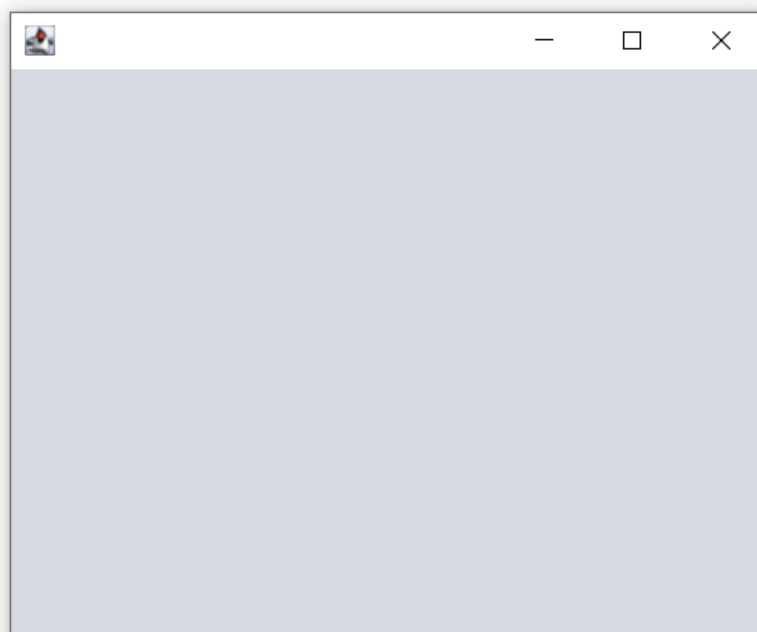


Rys.11.4. Utworzona klasa z formatką GUI.

Komponenty

Pakiet Swing posiada wiele komponentów z których każdy jest odrębną klasą. Są one pogrupowane według kategorii: kontenery, kontrolki, menu, okna, filtry. Dodatkowo w palecie występują elementy z przestarzałej biblioteki AWT (gdyby istniała potrzeba ich wykorzystania). Komponenty odpowiadające za interakcję (kontrolki) są najliczniejszą grupą – w jej skład wchodzi m.in.: przyciski, pola tekstowe, etykiety, suwaki i inne.

Po dodaniu wybranych komponentów należy (o ile istnieje taka potrzeba) oprogramować ich działanie, czyli dla każdego komponentu, który ma reagować z użytkownikiem musi powstać kod realizujący dane zadanie/-a. Za taki mechanizm odpowiedzialne są zdarzenia (ang. event) oraz metody nasłuchujące (ang. listener). Po uruchomieniu projektu utworzonego zgodnie z rysunkami 11.1 – 4 pojawi się puste okno, bez żadnych komponentów i zdarzeń – rysunek 11.5.



Rys.11.5. Okno JFrame bez żadnych komponentów i zaprogramowanych zdarzeń.

Zdarzenia i metody nasłuchujące

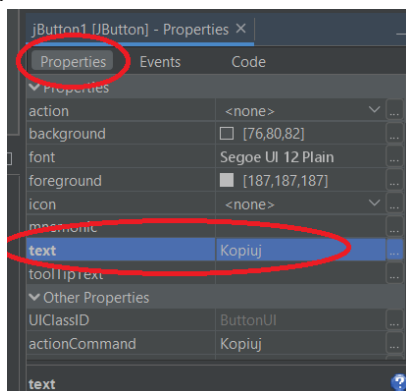
Podstawą programowania sterowanego zdarzeniami jest wiązanie zdarzeń z kodem, który jest reakcją na te zdarzenia. W pakiecie Swing interfejs (komponenty graficzne) został oddzielony od implementacji (kod odpowiedzialny za reakcje na zdarzenie). Każdy komponent może zgłaszać zdarzenia które mogą dla niego zachodzić, a każde zdarzenie może być zgłoszone osobno. Każdy komponent ma przypisane zdarzenie, które można uznać za najbardziej oczywiste i odpowiednią metodę nasłuchującą np. przycisk (klasa JButton) będzie posiadał metodę sprawdzającą czy został wciśnięty.

Tworząc aplikację bez wsparcia środowiska w postaci formatki (bez GUI form) komponenty mogą być tworzone na podstawie kodu (bez przeciągania i upuszczania). Wówczas do obsługi zdarzeń należy dla konkretnego zdarzenia wywołać metodę określającą odbiornik zdarzenia (np. przycisk klasy JButton) – `addActionListener()` z argumentem, który jest obiektem implementującym interfejs `ActionListener`. Analogiczna sytuacja wystąpiła w Laboratorium 7 przy obsłudze interfejsu myszki (`MouseListener`), gdzie w konstruktorze

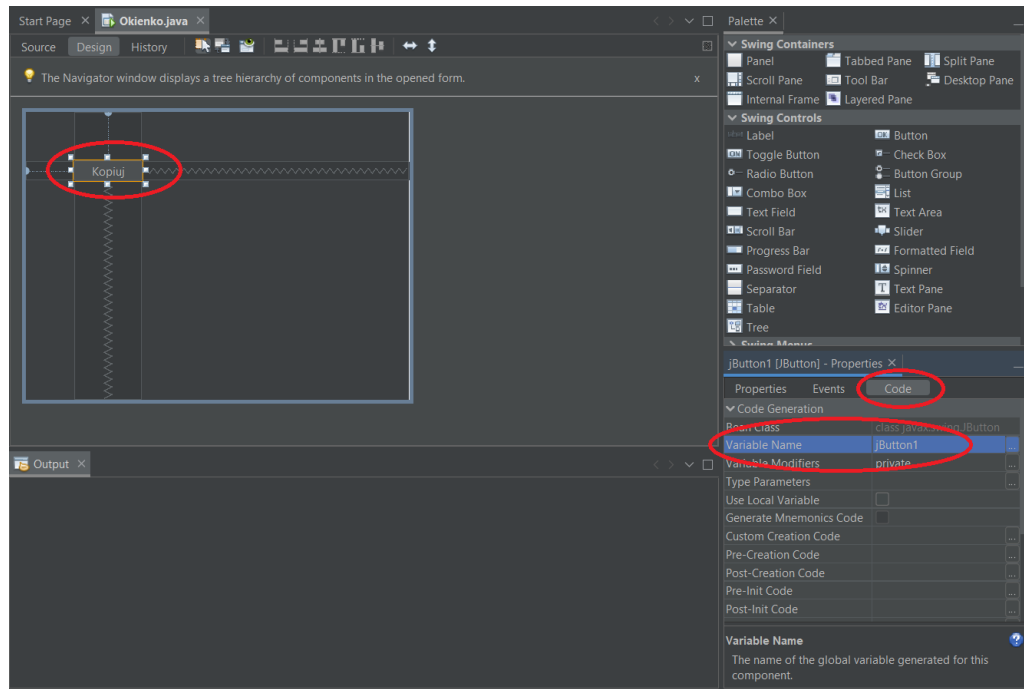
wywołano metody `addMouseListener` z odpowiednim argumentem. Po wciśnięciu przycisku zostanie wywołana metoda `actionPerformed()`.

Tworzenie aplikacji GUI

W tej części zostanie pokazane krok po kroku jak obsłużyć zdarzenie dla konkretnego komponentu. Wykorzystano tu komponenty takie jak przycisk, pole tekstowe i etykieta. Pierwszym krokiem jest umieszczenie pożądaných komponentów w obszarze ramki `JFrame`. Każdy element jest obiektem konkretnej klasy (np. `JButton`, `TextField`, `JLabel` itd.) więc możliwa (i zalecana) jest zmiana nazwy tego obiektu – jeżeli w danym interfejsie znajduje się osiem przycisków warto nazwać je zgodnie z przeznaczeniem zamiast używać domyślnych nazw tj. `jButton1`, `jButton2`... aż do `jButton8`. Dodatkowo dla każdego elementu można zdefiniować wyświetlany tekst (np. na przycisku, w polu tekstowym, etykiecie itd.). **Ważna uwaga!** Nazwa obiektu i tekst wyświetlany na komponencie nie jest tym samym (choć domyślnie mogą mieć takie same wartości). Na rys. 11.6-7 pokazano zmianę wyświetlanego tekstu bez zmiany nazwy obiektu. Poza wspomnianymi właściwościami zmienić można m.in. rozmiar i czcionkę, dodać ikonę (do wybranych komponentów), wyrównanie, kolor tła, dostęp, widoczność i wiele innych.



Rys. 11.6. Zmiana właściwości komponentu – wyświetlany tekst.

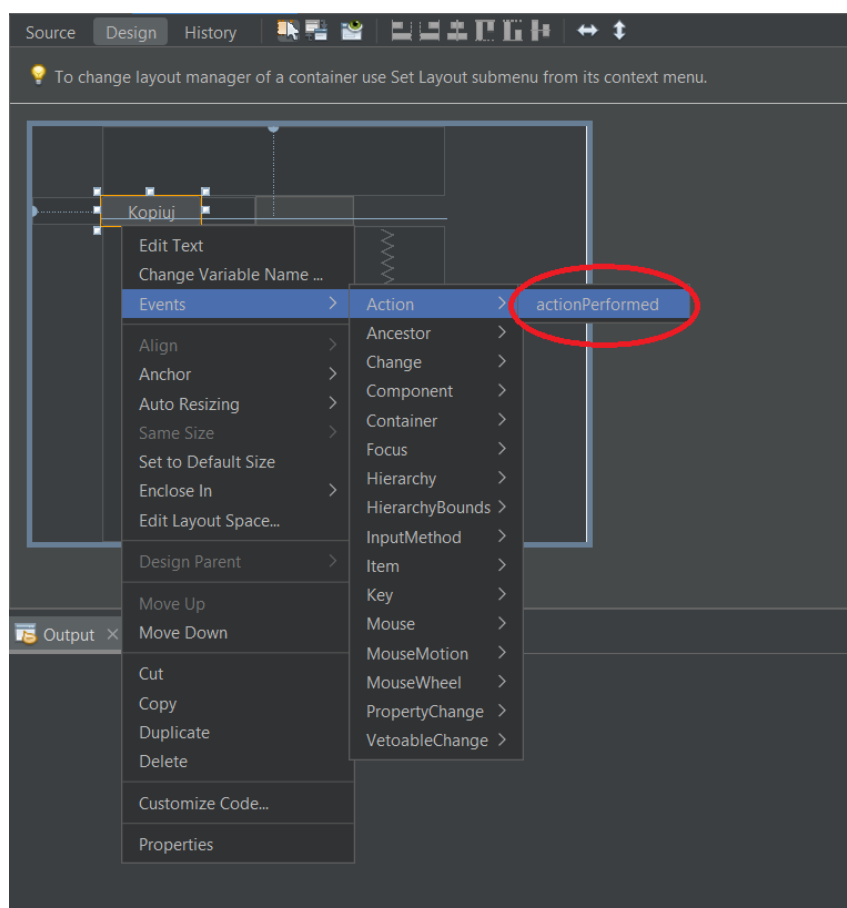


Rys. 11.7. Nazwa obiektu – komponentu.

```
// Variables declaration - do not modify
private javax.swing.JLabel etykieta;
private javax.swing.JButton kopiuujBtn;
private javax.swing.JTextField poleTxt;
// End of variables declaration
```

Rys.11.8. Nowe nazwy komponentów.

Zmiana nazw komponentów jest możliwa tylko z poziomu okna właściwości w widoku projektowym – w kodzie nie ma takiej możliwości (rys.11.8). Po dodaniu pozostałych komponentów i zmianie wyświetlanego tekstu oraz nazw należy stworzyć odpowiednie zdarzenie które będzie skutkowało reakcją. W tym przypadku po wpisaniu tekstu w pole tekstowe i wciśnięciu przycisku, pole tekstowe ma zostać wyczyszczone, a tekst ma pojawić się w etykiecie. Zdarzenie będzie generowane po naciśnięciu przycisku więc dla tego komponentu ma zostać wywołana odpowiednia metoda. Należy kliknąć PPM na przycisk i wybrać Events → Action → actionPerformed. Dostępnych jest wiele innych gotowych interfejsów do obsługi zdarzeń – w zależności od „zadania” należy wybrać odpowiedni. Na rysunku 11.9 przedstawiono wybór przykładowego zdarzenia.



Rys.11.9. Wybór zdarzenia do obsługi.

Po wybraniu zdarzenia środowisko przełączy się do edytora kodu i w odpowiednim miejscu wygeneruje metodę do obsługi zdarzenia. Rolą programisty jest uzupełnienie jej zgodnie z założeniami projektowymi – w tym przypadku należy pobierać tekst z pola tekstowego, umieścić go w etykiecie i wyczyścić pole tekstowe.

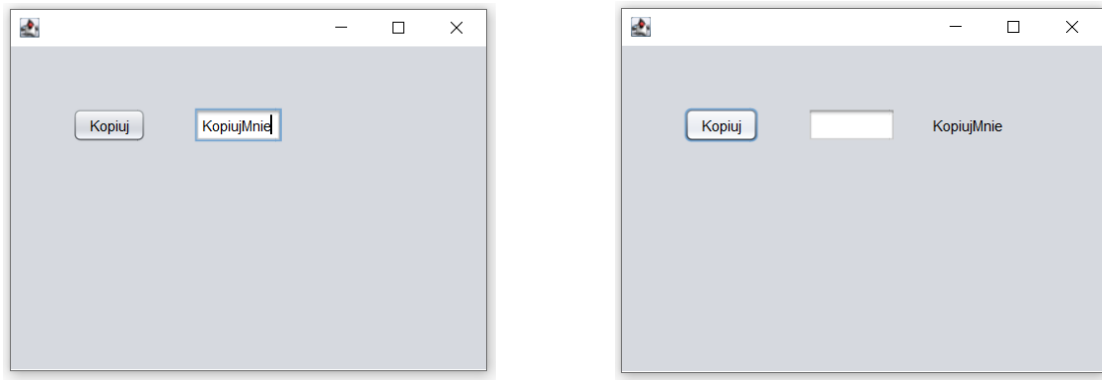
Uzupełniona metoda została pokazana poniżej – zwróć uwagę, że dla konkretnego komponentu jest generowana metoda używająca nazwy komponentu (po zmianie nazwy komponentu środowisko automatycznie zaktualizuje nazwę metody).

```
private void
kopiujBtnActionPerformed(java.awt.event.ActionEvent evt) {

    etykieta.setText(poleTxt.getText());
    poleTxt.setText("");
}
```

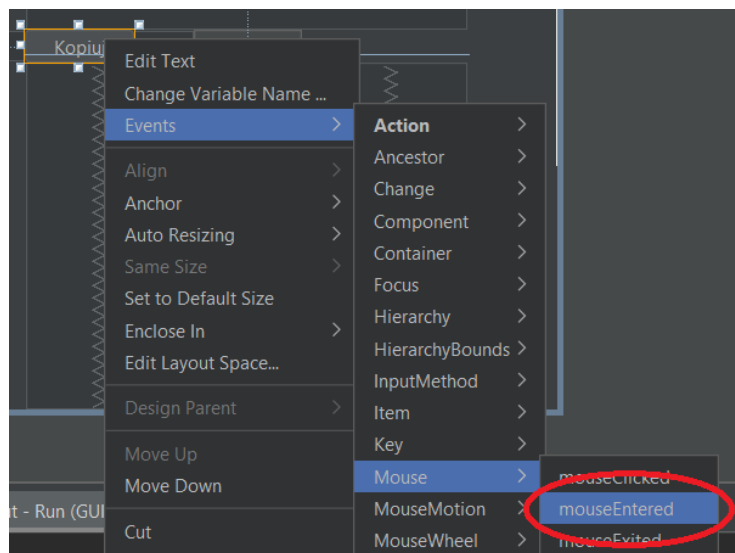
Działanie uruchomionej aplikacji pokazano na rys 11.10.





Rys.11.10. Działanie aplikacji.

Dany komponent może reagować na więcej niż jedno zdarzenie – przykładowo dla przycisku można zaprogramować kolejne dodając je analogicznie do pierwszego (rys.11.11).



Rys.11.11. Dodanie kolejnego zdarzenia dla tego samego komponentu.

Warto zauważyć, że po dodaniu konkretnej metody obsługującej zdarzenie komponent nie może mieć więcej metod przypisanych dla zdarzenia tego samego „rodzaju” – nie można rozróżnić kliknięcia myszą od kolejnego kliknięcia myszą.

W tym celu dodano kolejne zdarzenie – wykrycie myszki w obszarze przycisku oraz reakcję (zmiana koloru tła przycisku). Poniżej przedstawiono kod oraz działającą aplikację (rys.11.12).

```
private void kopiuujBtnMouseEntered(java.awt.event.MouseEvent  
evt) {  
  
    kopiuujBtn.setBackground(new  
Color((float)Math.random(), (float)Math.random(), (float)Mat  
h.random()));  
  
}
```

Każdorazowo przy wejściu kursora myszy w obszar przycisku zostanie wylosowany nowy kolor tła przycisku. Poza tym kliknięcie w przycisk realizuje już wcześniej napisaną metodę kopiowania tekstu z pola do etykiety.



Rys.11.12. Aplikacja z obsługą drugiego zdarzenia.

Z każdym używanym komponentem pakietu Swing należy zapoznać się dokładnie przed użyciem np. poprzez dokumentację oraz gotowe przykłady dostępne na oficjalnych stronach treningowych firmy Oracle.

Zadanie 11.1. Kalkulator

Napisz aplikację z graficznym interfejsem użytkownika spełniającą podstawowe funkcjonalności kalkulatora tj.:

- Wykonywanie podstawowych działań matematycznych (dodawanie, odejmowanie, mnożenie, dzielenie, pierwiastkowanie, potęgowanie)
- Wykonywanie dodatkowych działań – zamiana stopni na radiany i na odwrot
- Zamiana liczby systemu dziesiętnego na dwójkowy i na odwrot

Zadbaj o obsługę wyjątków i możliwość wprowadzania wyłącznie cyfr i separatorów. Wykorzystaj dowolne komponenty.

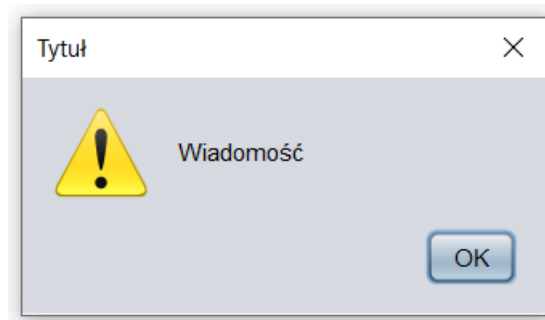
Zadanie 11.2. Układ równań

Napisz aplikację z graficznym interfejsem użytkownika pozwalającą na rozwiązywanie układu równań metodą wyznaczników. Uwzględnij możliwość rozwiązywania układu co najmniej 3 równań. Weź pod uwagę wszystkie możliwe rozwiązania i użyj odpowiedniego komunikatu.

Podpowiedź!

Do wyświetlenia wyniku możesz wykorzystać okienko klasy JOptionPane. Poniższy kod wywoła okno przedstawione na rys. 11.12.

```
JOptionPane.showMessageDialog(rootPane, "Wiadomość", "Tytuł",  
HEIGHT);
```



Rys. 11.12. Okno powiadomień JOptionPane.

LABORATORIUM 12. TWORZENIE APLIKACJI DO KOMUNIKACJI SIECIOWEJ.

Cel laboratorium:

Zapoznanie z podstawami obsługi komunikacji sieciowej w języku Java, stworzenie aplikacji wykorzystującej komunikację sieciową.

Zakres tematyczny zajęć:

- Tworzenie połączenia.
- Aplikacja klient – serwer.

Pytania kontrolne:

1. Czym jest gniazdo?
2. Jakich portów TCP nie należy używać w autorskich aplikacjach?
3. W jaki sposób nawiązuje się połączenie sieciowe?
4. Czy można uruchomić aplikację serwerową na tym samym komputerze co kliencką?

Ustanowienie połączenia

Nawiązanie połączenia z innym komputerem jest możliwe przy użyciu tzw. gniazda. W języku Java gniazdo jest reprezentowane przez obiekt klasy **Socket** z pakietu **java.net**. Połączenie sieciowe między dwoma komputerami jest relacją, dzięki której dwa programy wiedzą o swoim istnieniu – mogą się wzajemnie komunikować (przesyłać informacje lub po prostu bity). Do ustanowienia połączenia sieciowego niezbędne są dwie informacje – pierwsza to adres IP, a druga numer portu TCP.

Komputer pełniący rolę serwera dysponuje 65536 portami, z których (prawie) każdy być wykorzystywany przez aplikację pełniącą rolę serwera. Prawie każdy, ponieważ porty TCP od 0 do 1023 są zarezerwowane dla usług powszechnie używanych (np. HTTP – 80, HTTPS – 443, SMTP – 25, POP3 – 110). Dla własnych serwerów rekomendowane jest wybranie portu z zakresu 1024 – 65535.

Komunikator sieciowy

Dla lepszego zobrazowania w jaki sposób działa komunikacja sieciowa przygotowano przykład. Aplikacja – komunikator sieciowy składa się z pola tekstowego, przycisku i wielowierszowego pola tekstowego. W projekcie utworzono dwie klasy – jedna odpowiedzialna za część kliencką (Klient), a druga za część serwerową (Serwer). Aplikacja jest skonfigurowana w taki sposób aby można było uruchomić obie części na jednym komputerze (wykorzystując localhost). Jako pierwsza powinna zostać uruchomiona część serwerowa – należy PPM kliknąć na klasę na drzewie projektowym, a następnie wybrać Run File (ewentualnie zaznaczyć klasę i wykorzystać skrót Shift + F6). Analogicznie należy uruchomić część kliencką. Zapoznaj się z klasą Klient:

```
package com.mycompany.komunikacjasieciowa;

import java.awt.*;
```



```
import java.awt.event.*;
import java.io.*;
import java.net.*;
import javax.swing.*;

public class Klient {

    JTextArea odbiorWiadomosci;
    JTextField wiadomosc;
    BufferedReader czytelnik;
    PrintWriter pisarz;
    Socket gniazdo;

    public static void main(String[] args) {

        Klient klient = new Klient();
        klient.polaczMnie();
    }

    public void polaczMnie() {
        JFrame frame = new JFrame("Prosty klient czatu");
        JPanel panel = new JPanel();

        odbiorWiadomosci = new JTextArea(15, 50);
        odbiorWiadomosci.setLineWrap(true);
        odbiorWiadomosci.setWrapStyleWord(true);
        odbiorWiadomosci.setEditable(false);

        JScrollPane przewijanie = new
        JScrollPane(odbiorWiadomosci);

        przewijanie.setVerticalScrollBarPolicy(ScrollPaneConstants.VERTICAL_SCROLLBAR_ALWAYS);

        przewijanie.setHorizontalScrollBarPolicy(ScrollPaneConstants.HORIZONTAL_SCROLLBAR_NEVER);

        wiadomosc = new JTextField(20);

        JButton przyciskWyslij = new JButton("Wyslij");
        przyciskWyslij.addActionListener(new
        SluchaczPrzycisku());

        panel.add(przewijanie);
        panel.add(wiadomosc);
        panel.add(przyciskWyslij);
        konfiguruj();

        Thread watekOdbiorcy = new Thread(new Odbiorca());
    }
}
```



```
watekOdbiorcy.start();

frame.getContentPane().add(BorderLayout.CENTER,
panel);
frame.setSize(new Dimension(600, 400));
frame.setVisible(true);
}

private void konfiguruj() {
    try {
        gniazdo = new Socket("127.0.0.1", 2020);
        InputStreamReader czytnikStrm = new
InputStreamReader(gniazdo.getInputStream());
        czytnik = new BufferedReader(czytnikStrm);
        pisarz = new
PrintWriter(gniazdo.getOutputStream());
        System.out.println("Zakończono konfiguracje
sieci");
    } catch (IOException ex) {
        System.out.println("Konfiguracja sieci nie
powiodła się!");
        ex.printStackTrace();
    }
}

private class SluchaczPrzycisku implements ActionListener
{

    @Override
    public void actionPerformed(ActionEvent e) {
        try {
            pisarz.println(wiadomosc.getText());
            pisarz.flush();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
        wiadomosc.setText("");
        wiadomosc.requestFocus();
    }
}

public class Odbiorca implements Runnable {

    @Override
    public void run() {
        String wiad;
        try {
            while ((wiad = czytnik.readLine()) != null)
{
```



```
                System.out.println("Odczytano: " + wiad);
                odbiorWiadomosci.append(wiad + "\n");
            }
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

Zapoznaj się z klasą Serwer:

```
package com.mycompany.komunikacjasieciowa;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.net.ServerSocket;
import java.net.Socket;
import java.util.ArrayList;
import java.util.Iterator;

public class Serwer {

    ArrayList strumienieWyjsciowe;

    public class ObslugaKlienta implements Runnable {

        BufferedReader czytelnik;
        Socket gniazdo;

        public ObslugaKlienta(Socket gniazdo) {
            try {
                this.gniazdo = gniazdo;
                InputStreamReader reader = new
InputStreamReader(gniazdo.getInputStream());
                czytelnik = new BufferedReader(reader);

                } catch (Exception ex) {
                    ex.printStackTrace();
                }
            }

            @Override
            public void run() {
                String wiadomosc;
                try {
                    while ((wiadomosc = czytelnik.readLine()) !=
null) {
```

```
        System.out.println("Odczytano: " +
wiadomosc);
        rozeslijDoWszystkich(wiadomosc);
    }
    } catch (IOException ex) {
        ex.printStackTrace();
    }
}

public static void main(String[] args) {
    new Serwer().doRoboty();
}

public void doRoboty() {
    strumienieWyjsciowe = new ArrayList();
    try {
        ServerSocket gniazdoSerwera = new
ServerSocket(2020);

        while (true) {
            Socket gniazdoKlienta =
gniazdoSerwera.accept();
            PrintWriter pisarz = new
PrintWriter(gniazdoKlienta.getOutputStream());
            strumienieWyjsciowe.add(pisarz);

            Thread watekKlienta = new Thread(new
ObslugaKlienta(gniazdoKlienta));
            watekKlienta.start();
            System.out.println("Nawiązano połączenie!");
        }
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}

public void rozeslijDoWszystkich(String wiadomosc) {
    Iterator it = strumienieWyjsciowe.iterator();
    while (it.hasNext()) {
        try {
            PrintWriter pisarz = (PrintWriter) it.next();
            pisarz.println(wiadomosc);
            pisarz.flush();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
}
```

Aplikacja po uruchomieniu wygląda jak przedstawiono na rysunku 12.1. Użytkownik wprowadza tekst wiadomości w pole tekstowe. Po kliknięciu przycisku „Wyslij” wiadomość



Fundusze Europejskie
Wiedza Edukacja Rozwój

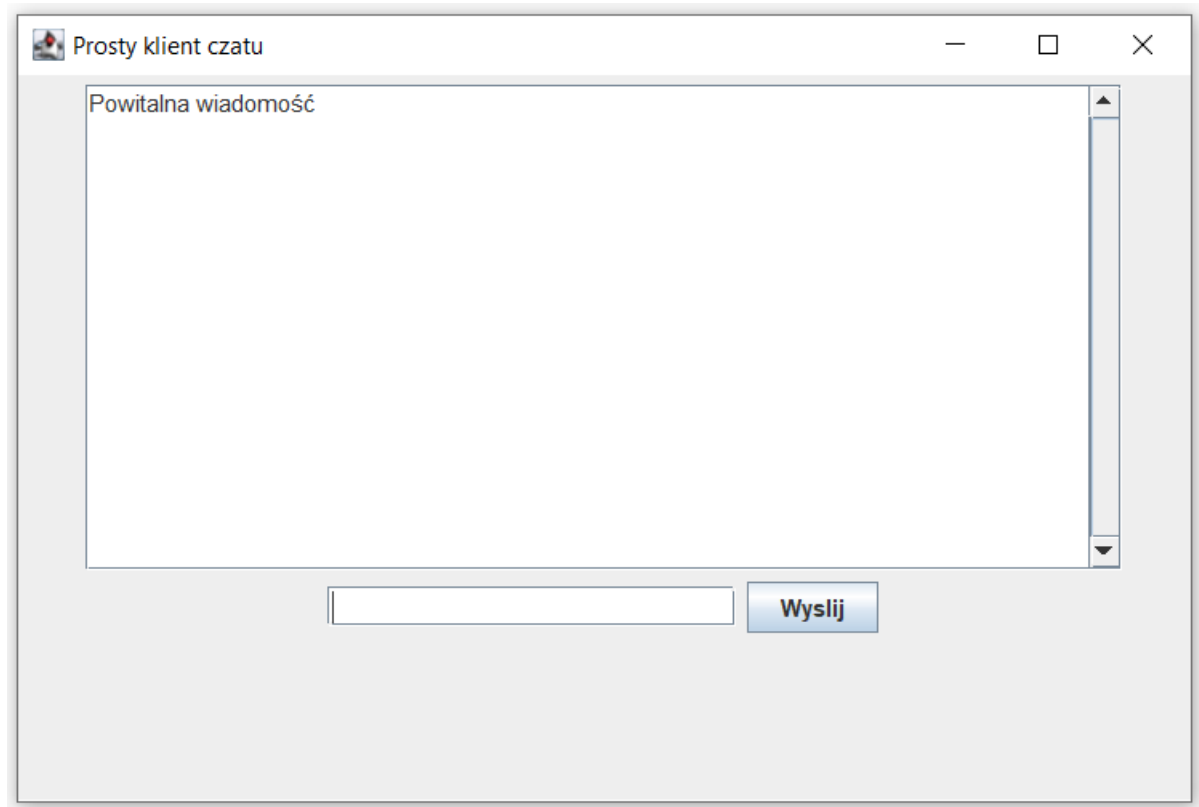


**Rzeczpospolita
Polska**

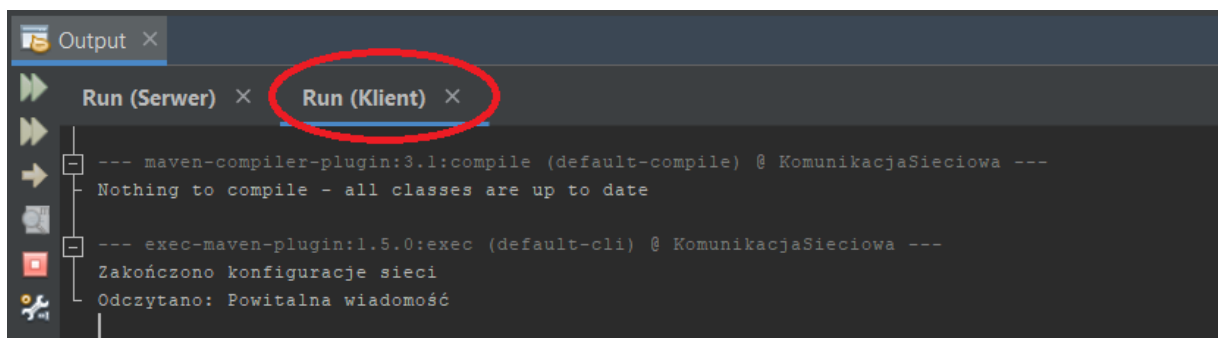
Unia Europejska
Europejski Fundusz Społeczny



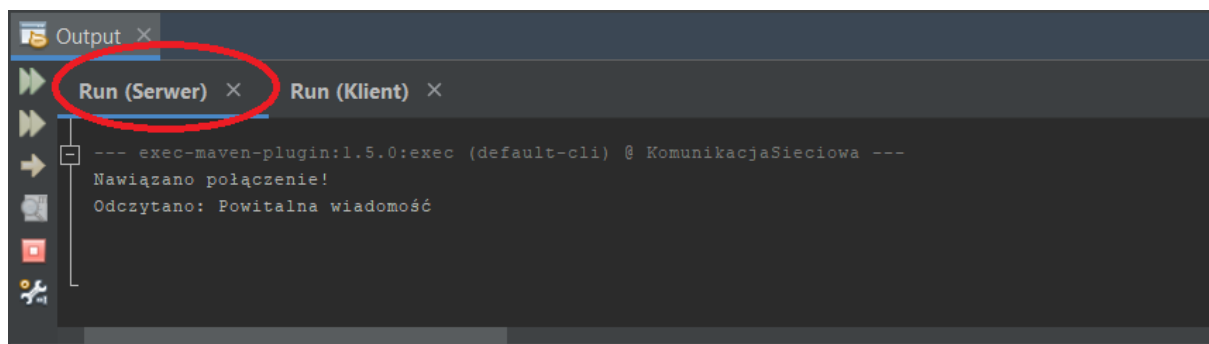
jest wysyłana na serwer. Wszystkie wiadomości znajdujące się na serwerze zostają pobrane i wyświetlone w wielowierszowym polu tekstowym. Na konsoli wyjściowej można zaobserwować przesłane wiadomości. Konsole wyjściowe poszczególnych części (klient, serwer) pokazano na rysunku 12.2 i 12.3.



Rys.12.1. Aplikacja wykorzystująca komunikację sieciową.



Rys.12.2. Konsola wyjściowa – klient.



Rys.12.3. Konsola wyjściowa – serwer.

Zadanie 12.1. Komunikator sieciowy – GUI

Wykorzystaj podany kod do stworzenia aplikacji GUI opartej na formatce Swing dostępnej w środowisku NetBeans. Rozbuduj aplikację o dodatkowe komponenty (np. etykiety) wyświetlające podstawowe dane klienta/ów. Przed podłączeniem klienta do serwera żądaj potwierdzenia (np. w konsoli). Wprowadź niezbędne modyfikacje.

Zadanie 12.2. Komunikator sieciowy - szyfr

Zmodyfikuj zadanie 1.1 w taki sposób, aby wiadomość przed wysłaniem na serwer była zaszyfrowana przy użyciu szyfru Vigenere'a. Deszyfrowanie ma być dostępne wyłącznie po stronie klienta – na serwerze wiadomość ma zostać przechowywana w postaci zaszyfrowanej.

LABORATORIUM 13. TWORZENIE APLIKACJI DO OBSŁUGI BAZY DANYCH.

Cel laboratorium:

Zapoznanie z obsługą baz danych w języku Java, stworzenie prostej aplikacji obsługującej bazę danych.

Zakres tematyczny zajęć:

- Baza danych.
- Podstawy SQL.
- Konfiguracja projektu.
- Aplikacja do obsługi bazy danych.

Pytania kontrolne:

1. Czym jest baza danych?
2. Jakie znasz polecenia SQL
3. Co jest niezbędne do obsługi bazy danych w języku Java?
4. Jakie klasy są używane do obsługi bazy danych?
5. W jaki sposób odczytać dane z bazy?
6. W jaki sposób dodać dane do bazy?
7. Czy w trakcie działania programu możliwe jest programowe usunięcie danych z tabeli?

Baza danych

Baza danych (ang. database) jest zbiorem danych uporządkowanych zgodnie z określonymi regułami. W przypadku relacyjnych baz danych są to dane w postaci tabeli wraz z relacjami pomiędzy nimi. Przechowywanie dużej ilości danych (powiązanych ze sobą różnymi relacjami) np. w pliku tekstowym nie jest zalecanym podejściem. W tym laboratorium zostanie przedstawiony sposób wykorzystania relacyjnych bazy danych w języku Java, bez wchodzenia w specyfikę języka SQL (ang. Structured Query Language) używanego do tworzenia i modyfikowania baz danych.

Niezbędne podstawy SQL

Język SQL został opracowany w latach 70., a więc ponad 20 lat przed stworzeniem Javy 1.0. Od tego czasu był wprowadzany w wielu projektach komercyjnych i wiązało się to z ciągłymi modyfikacjami.

Podstawowymi poleceniami języka SQL są:

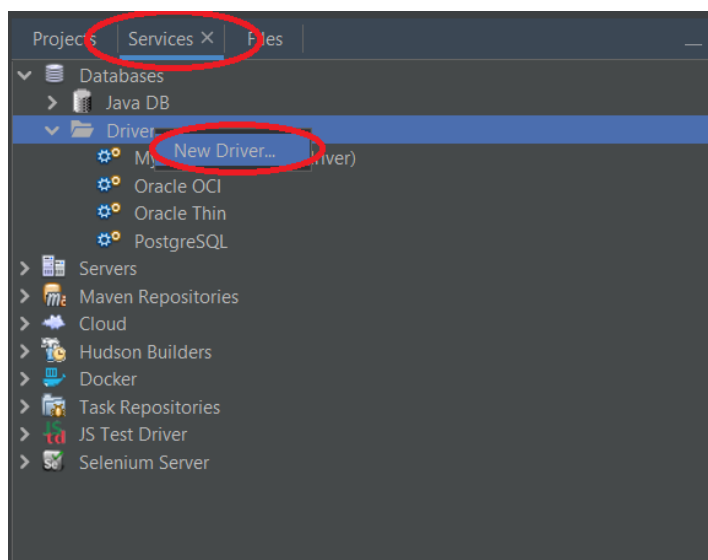
- CREATE – tworzy tabelę,
- DROP – usuwa tabelę,
- INSERT – umieszcza dane w tabeli,
- UPDATE – modyfikuje wiersze w tabeli,
- SELECT – pobiera dane z tabeli,
- DELETE – usuwa rekord z tabeli.



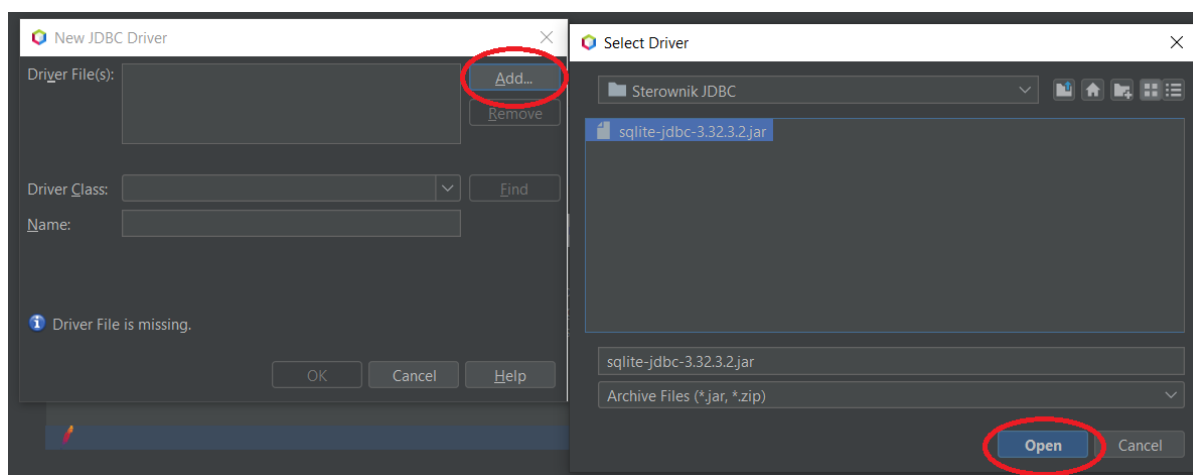
Naturalnie język SQL jest dużo bardziej rozbudowany ale na potrzeby utworzenia i modyfikacji prostej bazy danych przy użyciu oprogramowania napisanego w języku Java powyższe polecenia będą wystarczające.

Konfiguracja projektu

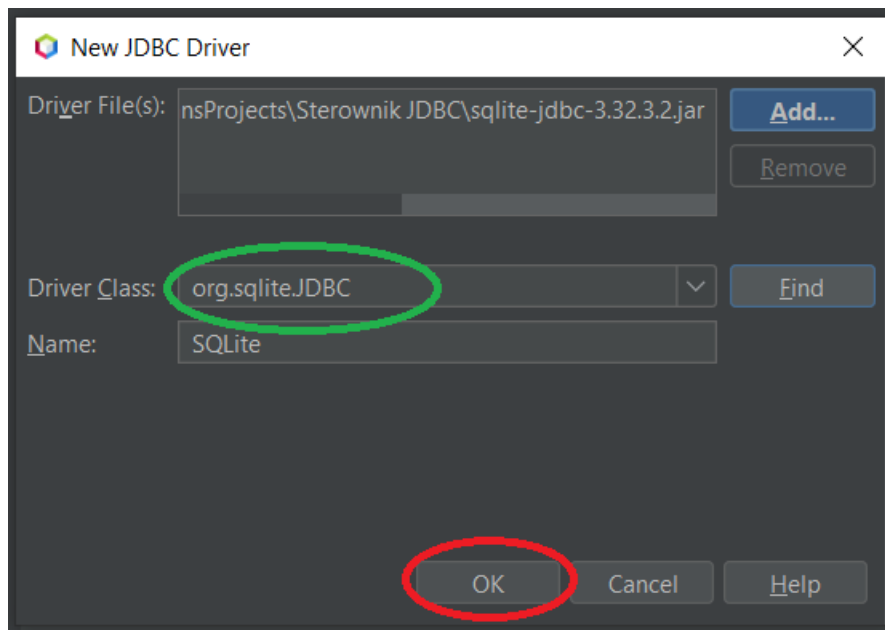
Korzystanie z bazy danych SQL w projekcie Java wymaga użycia specjalnej biblioteki ze sterownikiem JDBC (ang. Java DataBase Connectivity). W zależności od rodzaju bazy danych wymagany jest inny sterownik – dla przykładu zostanie wykorzystany sterownik do bazy danych SQLite. Jest to mała, szybka, wysoce wydajna i niezawodna baza danych (jest to baza bezserwerowa). Format pliku SQLite jest stabilny, wieloplatformowy i wstecznie kompatybilny. Sterownik JDBC można pobrać ze zdalnego repozytorium i dodać ręcznie do projektu lub wykorzystać funkcjonalność Maven’a do zarządzania zależnościami (wówczas należy dodać odpowiednią informację w pliku pom.xml). Dodanie ręczne pokazano na rysunku 13.1. – 3.



Rys.13.1. Dodanie sterownika JDBC – zakładka Services.



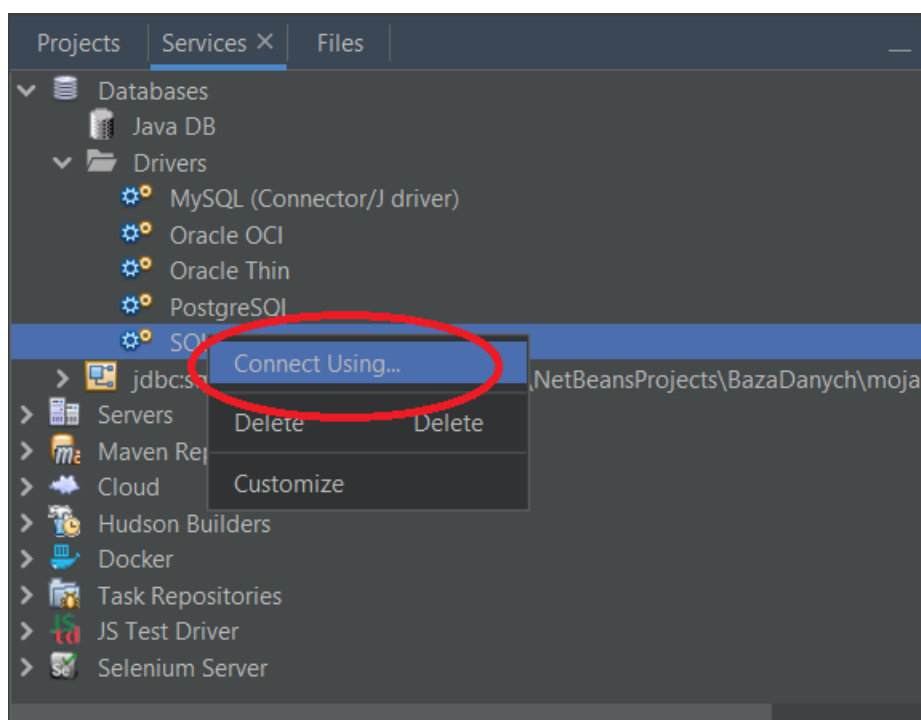
*Rys. 13.2. Dodanie pobranego pliku z rozszerzeniem *.jar.*



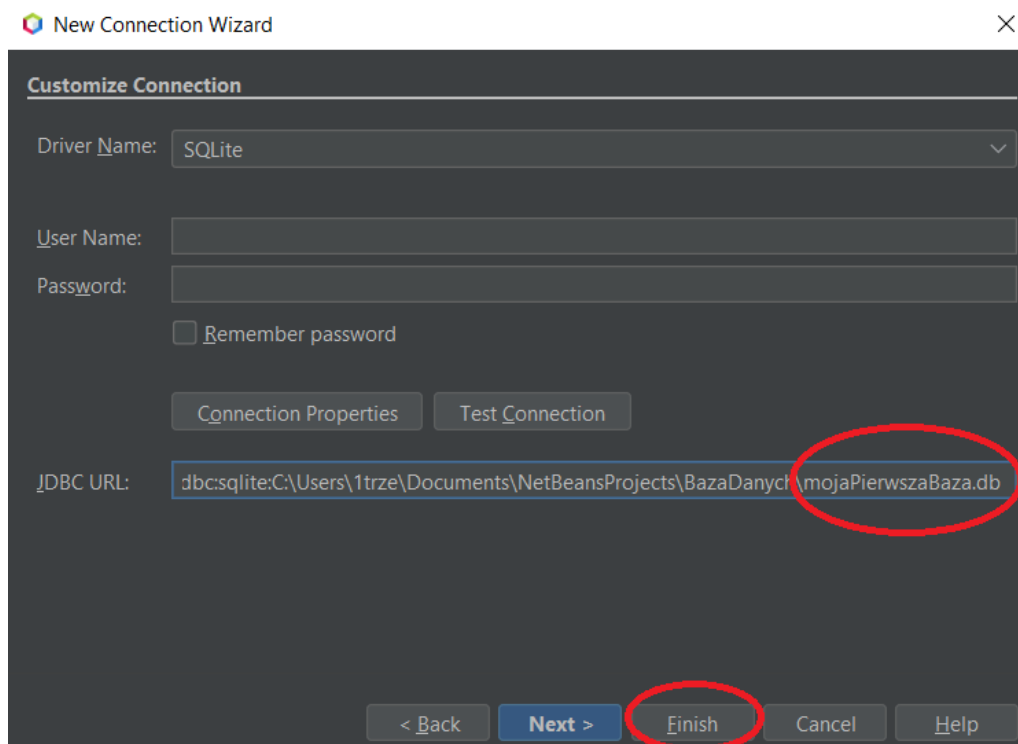
Rys.13.3. Dodany sterownik wraz z zaznaczoną na zielono nazwą klasy.

Po dodaniu sterownika należy utworzyć połączenie klikając PPM na dodany sterownik SQLite → Connect Using...(rys. 13.4), w kolejnym kroku (rys.13.5) podając ścieżkę dostępu do bazy danych:

jdbc:sqlite:C:\Users\...\NetBeansProjects\BazaDanych\mojaPierwszaBaza.db

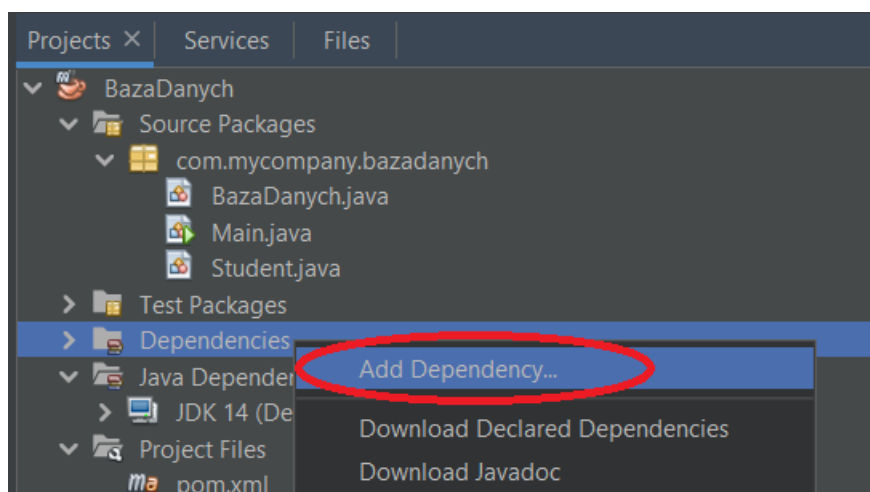


Rys.13.4. Tworzenie połączenia.

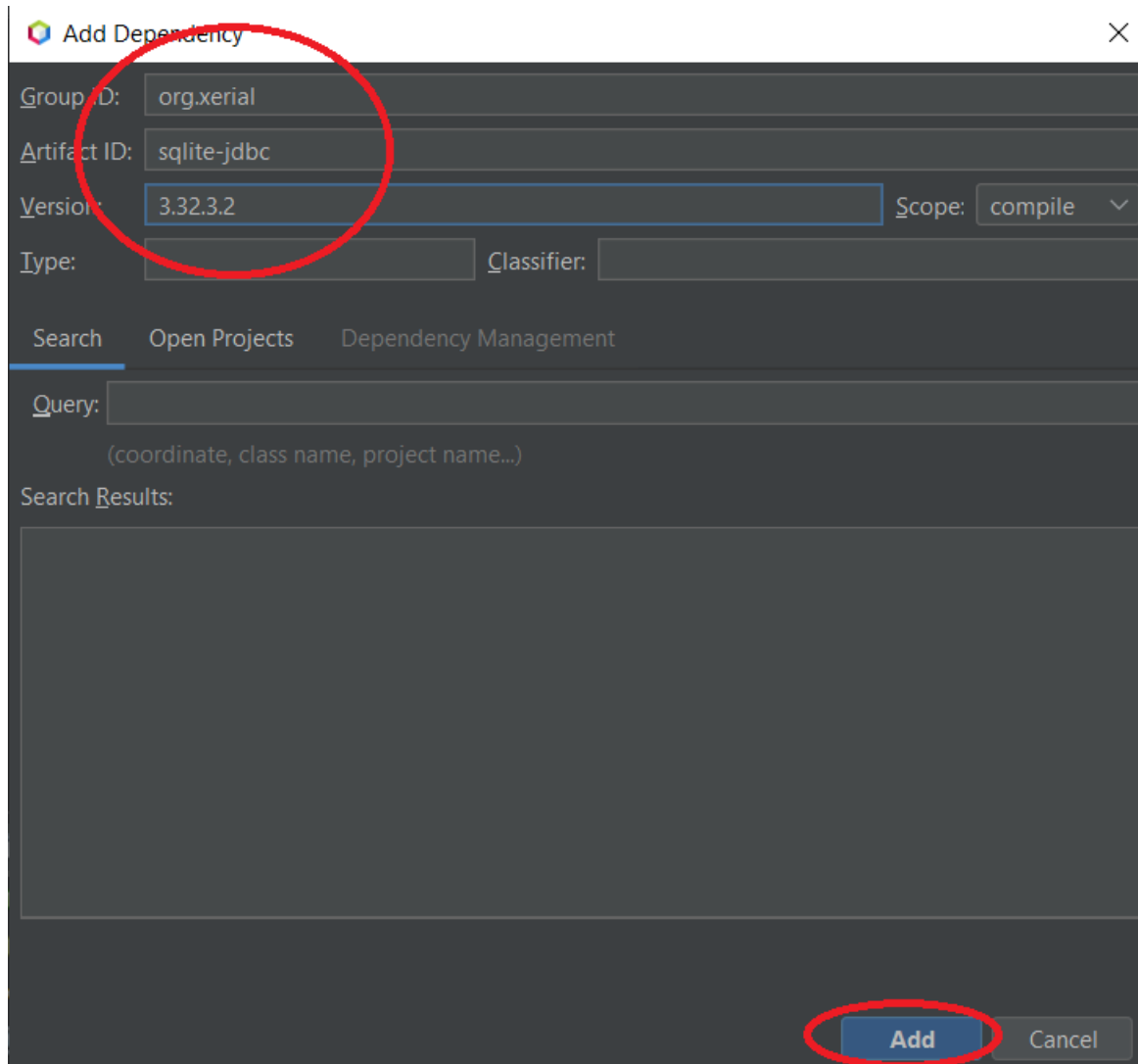


Rys.13.5. Dodanie ścieżki do bazy danych.

Alternatywą dla pobierania i ręcznego dodawania sterownika JDBC jest wykorzystanie Maven'a – w pliku pom.xml należy dodać zależność jak pokazano poniżej (rys. 13.6 – 7). Numer wersji sterownika można sprawdzić w repozytorium Maven'a – warto korzystać z najnowszej. Po dodaniu zależności sterownik zostanie pobrany.



Rys.13.6. Dodanie zależności – sterownika JDBC.



The screenshot shows the 'Add Dependency' dialog box. The 'Group ID' field is filled with 'org.xerial', the 'Artifact ID' field with 'sqlite-jdbc', and the 'Version' field with '3.32.3.2'. The 'Scope' dropdown is set to 'compile'. The 'Add' button at the bottom right is circled in red.

Rys.13.7. Podanie danych niezbędnych do dodania zależności.

Tworzenie bazy danych

Dla lepszego zapoznania się z tematyką wykorzystywania baz danych w aplikacjach Java poniżej przedstawiono przykład. Projekt składa się z klasy zarządzającej bazą danych (BazaDanych), klasy pomocniczej (Student) oraz klasy głównej (Main). W klasie zarządzającej zdefiniowano cztery metody:

- tworzenie tabeli STUDENCI (tworzTabele()),
- wstawianie danych (wstawDane()),
- pobieranie danych (pobierzDane()),
- zamykanie połączenia (zamknijPolaczenie()).

Korzystając z klasy BazaDanych oraz klasy Student możliwe jest utworzenie tabeli STUDENCI oraz dodawanie i odczyt danych z bazy. W metodzie main() zrealizowano przykładowe wstawianie i odczyt danych.

```
package com.mycompany.bazadanych;
```



Fundusze Europejskie
Wiedza Edukacja Rozwój



**Rzeczpospolita
Polska**

Unia Europejska
Europejski Fundusz Społeczny



```
import java.sql.*;
import java.util.LinkedList;
import java.util.List;

public class BazaDanych {

    private static final String DRIVER = "org.sqlite.JDBC";
    private static final String DB_URL =
"jdbc:sqlite:mojaPierwszaBaza.db";

    private Connection connection;
    private Statement statement;

    public BazaDanych() {
        try {
            Class.forName(DRIVER);
        } catch (ClassNotFoundException ex) {
            System.err.println("Brak sterownika JDBC");
            ex.printStackTrace();
        }
        try {
            connection = DriverManager.getConnection(DB_URL);
            statement = connection.createStatement();
            tworzTabele();
        } catch (SQLException ex) {
            System.err.println("Problem z otwarciem
połączenia");
            ex.printStackTrace();
        }
    }

    public boolean tworzTabele() {

        String tworz = "CREATE TABLE IF NOT EXISTS STUDENCI(id
INTEGER PRIMARY KEY AUTOINCREMENT, nazwisko String, imie
String)";
        try {
            statement.execute(tworz);
        } catch (SQLException e) {
            System.err.println("Błąd przy tworzeniu tabeli");
            e.printStackTrace();
            return false;
        }

        return true;
    }

    public boolean wstawDane(String tabela, String nazwisko,
String imie) {
```




```
        try {
            PreparedStatement preparedStatement =
connection.prepareStatement("INSERT INTO " + tabela + " VALUES
(null,?,?)");

            preparedStatement.setString(1, nazwisko);
            preparedStatement.setString(2, imie);

            preparedStatement.execute();

        } catch (SQLException e) {
            System.err.println("Błąd przy wprowadzaniu danych
studenta: " + nazwisko + " " + imie);
            e.printStackTrace();
            return false;
        }

        return true;
    }

    public List<Student> pobierzDane(String tabela) {

        List<Student> wyjscie = new LinkedList<Student>();
        try {
            ResultSet resultSet =
statement.executeQuery("SELECT * FROM " + tabela);
            int id;
            String nazwisko, imie;
            while (resultSet.next()) {
                id = resultSet.getInt("id");
                nazwisko = resultSet.getString("nazwisko");
                imie = resultSet.getString("imie");

                wyjscie.add(new Student(id, nazwisko, imie));
            }
        } catch (SQLException e) {
            System.err.println("Problem z wczytaniem danych z
BD");
            e.printStackTrace();
            return null;
        }
        return wyjscie;
    }

    public void zamknijPolaczenie() {
        try {
            connection.close();
        }
```



```
        } catch (SQLException e) {
            System.err.println("Problem z zamknięciem
połączenia");
            e.printStackTrace();
        }
    }
}
```

```
package com.mycompany.bazadanych;

public class Student {

    private int id;
    private String nazwisko, imie;

    public Student(int id, String nazwisko, String imie) {
        this.id = id;
        this.nazwisko = nazwisko;
        this.imie = imie;
    }

    public int getId() {
        return id;
    }

    public void setId(int id) {
        this.id = id;
    }

    public String getNazwisko() {
        return nazwisko;
    }

    public void setNazwisko(String nazwisko) {
        this.nazwisko = nazwisko;
    }

    public String getImie() {
        return imie;
    }

    public void setImie(String imie) {
        this.imie = imie;
    }
}
```



```
package com.mycompany.bazadanych;

import java.util.List;

public class Main {

    public static void main(String[] args) {

        BazaDanych bazaDanych = new BazaDanych();

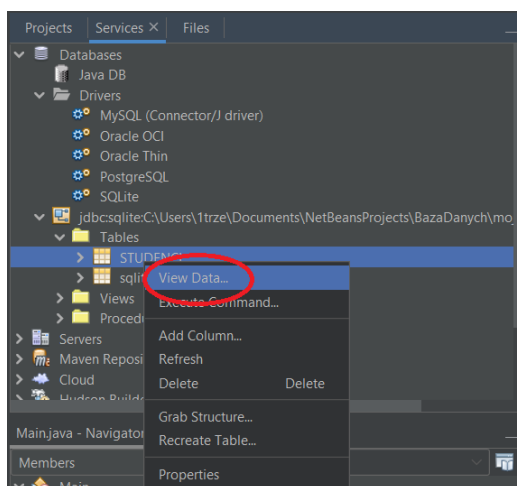
        bazaDanych.wstawDane("STUDENCI", "Kowalski", "Jan");
        bazaDanych.wstawDane("STUDENCI", "Wiśniewski",
"Piotr");
        bazaDanych.wstawDane("STUDENCI", "Nowak", "Michał");

        List<Student> lista =
bazaDanych.pobierzDane("STUDENCI");

        for (Student s : lista) {
            System.out.println(s.getId() + " " +
s.getNazwisko() + " " + s.getImie());
        }
        bazaDanych.zamknijPolaczenie();
    }
}
```

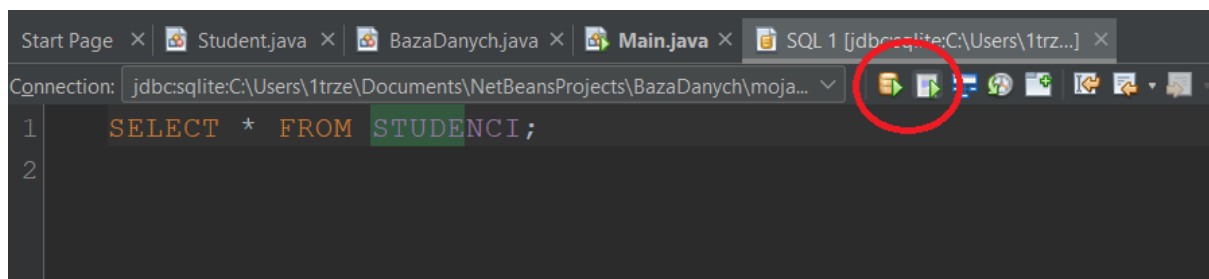
Po uruchomieniu aplikacji zostanie utworzona instancja klasy BazaDanych, a następnie zostaną wywołane na niej metody: trzykrotnie wstawDane() oraz jednokrotnie pobierzDane(). W pętli for – each pobrana lista zostanie wyświetlona.

Środowisko NetBeans umożliwia podgląd danych znajdujących się w bazie – w zakładce Services (rys. 13.8) należy wybrać podłączoną bazę i na konkretnej tabeli kliknąć PPM → View Data, a następnie wybrać Run SQL lub Run Statement (rys.13.9). Otrzymane wyniki przedstawiono na rysunku 13.10.

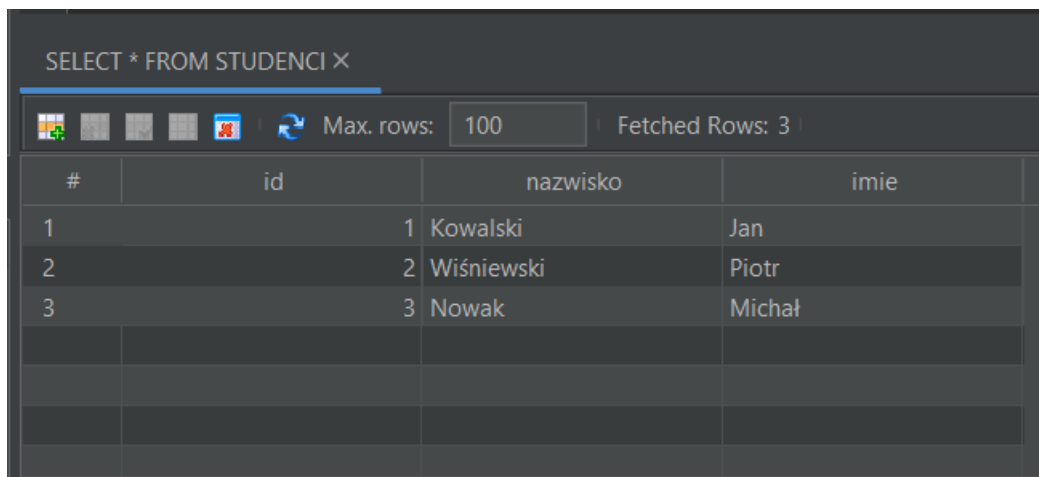


Rys.13.8. Podgląd danych z tabeli.





Rys.13.9. Utworzenie zapytania SQL.



#	id	nazwisko	imie
1	1	Kowalski	Jan
2	2	Wiśniewski	Piotr
3	3	Nowak	Michał

Rys.13.10. Wynik zapytania.

Dane mogą być również widoczne z poziomu dowolnego programu obsługującego bazy danych (np. SQLite Studio).

Zadanie 13.1. Baza danych - rozwinięcie

Wykorzystując kod dostępny w instrukcji dokonaj niezbędnych modyfikacji. Zmieniony kod wykorzystaj do stworzenia aplikacji, która będzie mogła z poziomu konsoli:

- dodawać dane studentów do bazy
- odczytywać dane konkretnych studentów z bazy
- modyfikować dane studentów w bazie (w tym usuwać rekordy)

Zadbaj o obsługę wyjątków, upewnij się, że nie można dodawać danych pustych lub niepoprawnych (podstawowa walidacja), np. nie powinien powstać rekord z wartością w kolumnie imię studenta: Piotr2020. W przypadku pustej bazy danych wyświetl informacje o braku rekordów.

Zadanie 13.2. Baza danych - GUI

Wykorzystaj kod z zadania 13.1 do stworzenia aplikacji GUI. Dodaj niezbędne komponenty (JTextField, JButton, JTextArea...). Aplikacja ma posiadać funkcjonalności wymienione w zadaniu 13.1 oraz przyjemny dla oka graficzny interfejs użytkownika.



Materiały zostały opracowane w ramach projektu
„Zintegrowany Program Rozwoju Politechniki Lubelskiej – część druga”,
umowa nr **POWR.03.05.00-00-Z060/18-00**
w ramach Programu Operacyjnego Wiedza Edukacja Rozwój 2014-2020
współfinansowanego ze środków Europejskiego Funduszu Społecznego