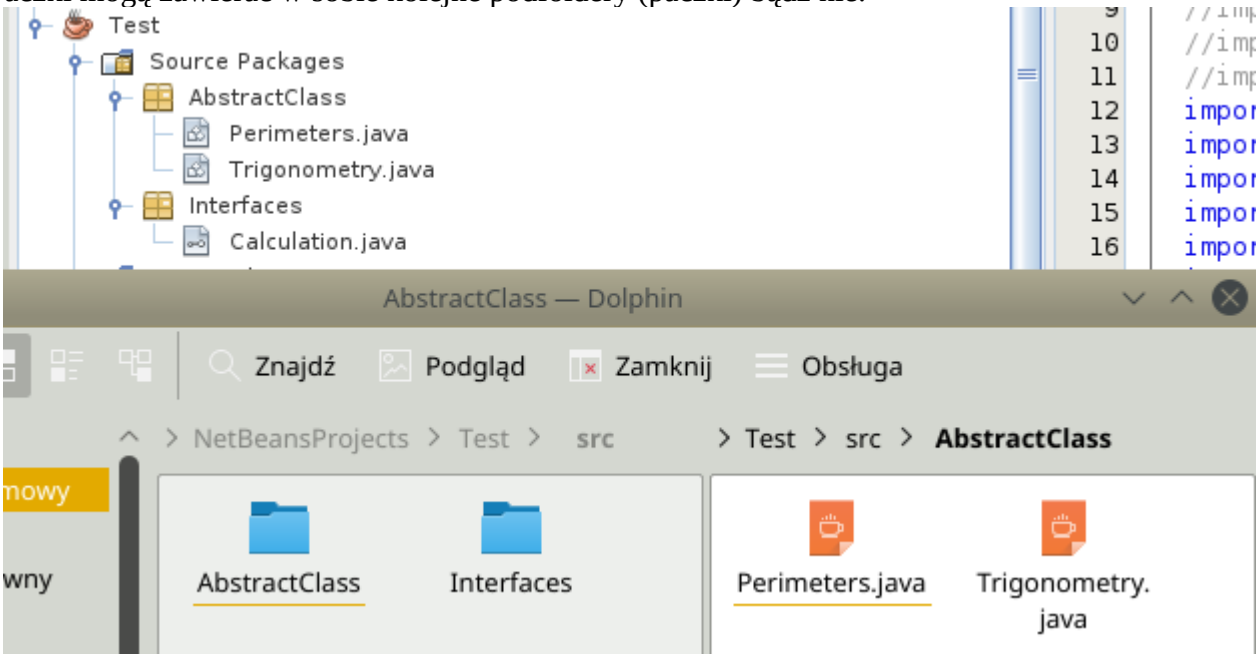
Podstawowe własności i właściwości języka Java.

Zadanie:

Otrzymaliśmy zadanie utworzenia narzędzia dla projektanta przestrzeni (dekorator wnętrz). Narzędzie ma za zadanie:
- reagować na login i hasło
- pozwalać na wyliczania powierzchni podstawowych figur geometrycznych jak i brył
- pozwalać wykonywać proste kalkulacje (dodawanie, odejmowanie, mnożenie, dzielenie)
- wyliczać kąty na podstawie podanych długości odcinków
- zapisywać odpowiednie konfiguracje (podane później)
- wynajdywać zapisane konfiguracje

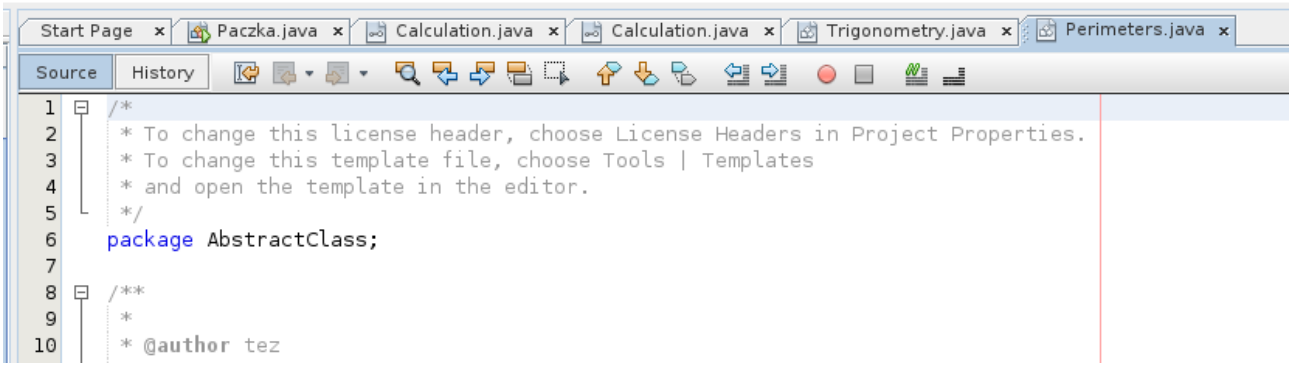
1. **Paczki (Packages)**

Paczki w języku Java to nic innego tylko foldery, w których składowane są pliki źródłowe kolejnych klas. Pojedynczy projekt może nie zawierać żadnej paczki (kod umieszczony bezpośrednio w katalogu projektu) bądź zawierać kilka paczek (kilka podfolderów z plikami .java). Paczki mogą zawierać w sobie kolejne podfoldery (paczki) bądź nie.



Powyższy zrzut najlepiej obrazuje pojęcie paczek. W projekcie Test posiadamy katalog źródeł (Source Packages, który strukturze katalogów nazywa się src), a w nim znajdują się dwa foldery, gdzie każdy z nich to jedna paczka. Przykładowo paczka AbstractClass zawiera dwie klasy – Perimeters oraz Trigonometry. Interfaces zaś zawiera (jeżeli byśmy spojrzeli do wnętrza folderu) jeden plik – Calculation.java.

Teraz należy spojrzeć od strony kodu źródłowego. Każdy z plików projektu znajdujący się w odpowiednim folderze MUSI posiadać specjalną linię na początku pliku kodu, która jednoznacznie będzie identyfikowała go ze wskazaną paczką.



Powyżej widać pierwsze linie kodu źródłowego Perimiters.java znajdującego się w paczce AbstractClass. Linia package AbstractClass MUSI wystąpić w tym pliku ponieważ ZNAJDUJE się on w tymże folderze (paczce). Bez tego polecenia wywołamy błąd kompilacji.

Po co stosować paczki? Ich najważniejszym celem, poza oczywistym segregowaniem plików źródłowych, jest zabezpieczanie projektu przed przypadkowym powieleniem nazw klas, metod bądź zmiennych. W obrębie pojedynczego zgrupowania (folderu) nazwy zmiennych nie powinny się powtarzać (wyjątek – polimorfizm, który zostanie objaśniony później). W przypadku małych aplikacji (takich jak nasza testowa) szansa na powtórzenie się tych samych nazw zmiennych czy funkcji/metod jest niewielka. Gdyby jednak wziąć pod uwagę większy kod, w tym kod obcej/innej osoby z drużyny, który chcemy podłączyć pod nasz projekt, taki konflikt nazw mógłby powstać. Dlatego ważnym jest by każdy moduł posiadał własną paczkę!

INFORMACJA: Rozwiązania podobne do wyżej opisywanego istnieją w niemal każdym języku II i III stopnia. Jednak znane są częściej jako przestrzenie nazw (namespaces). Działania i ich właściwości są jednak takie same.

Więcej informacji na temat paczek: https://www.tutorialspoint.com/java/java_packages.htm

2. **Klasy abstrakcyjne i interfejsy.**

a) klasy abstrakcyjne

Java, podobnie do innych języków, pozwala na tworzenie klas abstrakcyjnych. Pod pojęciem „abstracja” w języku programowania kryje się utworzenie typu, który sam w sobie stanowi niejako podkład i/lub model, wedle którego można tworzyć kolejne wersje innego elementu.

Najlepszym przykładem w codziennym życiu modelu abstrakcyjnego może być licencja wydawana np. na produkcje jakiegoś dobra (samochodu). Firma X posiada inżynierów, którzy od podstaw zaprojektowali model pojazdu XYZ. Prezesi nie chcą jednak dopuścić do produkcji tego modelu (np. polityka długofalowa firmy). Przychodzi jednak firma Z, która nie posiada inżynierów-projektantów, jednak chętnie rozpoczęła by w swoich halach produkcję jakiegoś samochodu. W tym momencie firma X udostępnia firmie Z na licencji model XYZ, który ta druga firma może produkować w postaci nie zmienionej lub zmienić jeden bądź więcej parametrów, przykładowo dodając głośniki, radio, szyberdach i inne, nie uwzględnione wyposażenie. Pozostaje jednak podwozie, nawozie, silnik i inne, niezbędne rzeczy w stanie nienaruszonym.

Jak można wywnioskować z powyższego przykładu, tego typu klasa nie może być inicjalizowana jako obiekt (a przynajmniej nie jawnie – znaczy bezpośrednio). Może ona stanowić rozszerzenie innej klasy (poprzez odpowiednie słowo-klauzulę extends). Klasa, której własności zostaną wzbogacone o elementy klasy abstrakcyjnej może przejąć jej metody i zmienne nie abstrakcyjne, wywołując je jako swoje. Standardowo można je również przepełnić (nadpisać własną definicją w

ciele klasy rozszerzanej). W klasie metody i własności mogą być typu publicznego, chronionego oraz prywatnego. Należy jednak pamiętać, że tylko typ publiczny/chroniony pozwala na nadpisanie funkcji/metody. W przypadku typu prywatnego NIE MA możliwości modyfikacji/nadpisania metody – tego typu możliwość istnieje jedynie w bazowej klasie.

INFORMACJA: W Java istnieją trzy typy zakresu widoczności metod:
- public – widoczna dla wszystkich; każdy może ją wywołać z dowolnymi parametrami, możliwe jest także nadpisanie ciała metody
- protected – w przypadku obiektów, które nie są rozszerzone poprzez klasę zawierającą pola protected, pola te traktowane są jako prywatne. Oznacza to, że projektant systemu nie może wywołać tych funkcji. Może natomiast je przeciążyć (nadpisać) jeżeli funkcja należy do klasy (extends) bądź w chwili inicjalizacji obiektu (dynamiczne typowanie).
- private – metoda nie może być nadpisywana ani modyfikowana poza swoją macierzystą klasą. Nie jest widoczna dla programistów (w sensie nie mogą z niej skorzystać POZA zawierającą ją klasą). Tego typu metody i zmienne mogą być wywoływane jedynie przez inne metody tejże klasy. (które mogą być dowolnego zakresu – private, protected lub public)

Klasy abstrakcyjne są w niektórych językach nazywane virtual.

b) interfejsy

Drugim rodzajem elementów o zastosowaniu bazowym są interfejsy. Element ten jest charakterystyczny dla języka Java (oraz C#, który przejął to właśnie po Java). W prostym uogólnieniu interfejs można przyrównać do klas wirtualnych/abstrakcyjnych, których wszystkie metody są czysto wirtualne/abstrakcyjne (nie posiadają swojej definicji).

Próbując przyrównać cechy i zastosowanie interfejsów w programowaniu możemy sobie wyobrazić, że firma A produkuje zestawy do nagłośnienia. Posiadają w swojej ofercie wszystkie niezbędne akcesoria i urządzenia – od przejściówek, poprzez kable, aż do samego sprzętu. Klient tej firmy może przebierać dowolnie w ofercie i wybierać tylko to, co go interesuje. Ponadto sam może decydować jaką funkcję będzie miał wskazany wzmacniacz czy głośnik. Inny klient, nawet wybierając ten sam zestaw, może dokonać montażu i połączeń w inny sposób (uzyskując odpowiednio gorszy lub lepszy efekt). W ten sam sposób działają interfejsy – każdy może dodane w nich metody zdefiniować w dowolny sposób, jednak każdy, kto podłącza dany interfejs musi wstępnie dołączyć wszystkie jego funkcje i metody (chociażby były niewykorzystane).

W przypadku interfejsów mamy ten sam schemat co w przypadku klas abstrakcyjnych – pola w nich zawarte muszą być publiczne (w zasadzie do wersji 8 nie można im było nadać innego zakresu widoczności).

c) podsumowanie

Klasy abstrakcyjne i interfejsy są do siebie podobne – pozwalają ujednolicać kod, budować aplikacje o odpowiednie szablony, a twórcom bibliotek i modułów umożliwiają na wskazanie innym programistom, jakie metody muszą zdefiniować u siebie by mogli w pełni wykorzystać potencjał swojego produktu.

Klasy w Java nie można rozszerzać o więcej niż jedną klasę. Oznacza to, że w danym momencie możemy rozszerzyć możliwości naszej klasy o dokładnie jedną klasę (abstrakcyjną bądź nie). Ograniczenia takiego nie ma natomiast w przypadku interfejsów – pojedyncza klasa może mieć zaimplementowaną teoretycznie nieskończoną ich liczbę.

Jednak klasa abstrakcyjna może posiadać składowe (zmienne i metody), które będą już zdefiniowane przez jej twórcę. Oznacza to, że będziemy mogli z nich korzystać od razu i w taki sposób, w jaki chce tego twórca. Tego typu udogodnienia nie posiadają interfejsy – dostarczają nam one jedynie deklaracji metod, które sami musimy zdefiniować.

Należy pamiętać, że klasa abstrakcyjna może sama w sobie implementować interfejsy. W tym przypadku programista, który będzie rozszerzał swoją klasę o taką klasę abstrakcyjną będzie miał dostęp do zdefiniowanych już metod interfejsu bądź interfejsów.

Ponadto trzeba pamiętać, że obiekt z klasy abstrakcyjnej można utworzyć, jednak musi to zostać zrobione w określony sposób. Przykładowo posiadamy klasę abstrakcyjną Perimiters. Klasa ta posiada w sobie metody perimiter oraz field. Następnie na jej podstawie tworzymy klasy Square oraz Rectangle. Każda z nich będzie potrzebowała odmiennej definicji obwodu i pola. Jednak docelowo każda z figur posiada takie same metody. Dlatego też możemy deklarować obiekt klasy abstrakcyjnej. Lepij obrazuje to prosty projekt dostępny pod adresem https://gitlab.com/examples4java/kurs1 (folder AbstractTest).

Więcej o klasach abstrakcyjnych oraz interfejsach:
https://www.javatpoint.com/abstract-class-in-java
https://www.javatpoint.com/interface-in-java
https://www.javatpoint.com/difference-between-abstract-class-and-interface

3. Konwersja typów w Java

Komputer rozpoznaje jedynie niskie i wysokie stany napięcia, które interpretuje jako stany logiczne 0 i 1. Pojedynczy stan przechowywany jest w jednostce pamięci zwanej bitem. Bity organizowane są w bajty, bajty w kilobajty itd. W przypadku komputerów 1 bajt złożony jest z 8 bitów. Każdy bit w takim zestawieniu posiada swoją tzw. wagę. Wagą jest wielokrotność (potęga) dwójki. Wagi rozpoczynają się od wartości 0. Przykład jednego bajta:

1	0	0	1	1	0	1	0
2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0

$1 * 2^7 + 0 * 2^6 + 0 * 2^5 + 1 * 2^4 + 1 * 2^3 + 0 * 2^2 + 1 * 2^1 + 0 * 2^0 = 128 + 16 + 8 + 2 = 154$

Jak widać z przykładu, na bajcie, w którym poszczególne bity będą miały identyczne ustawienie, komputer zapisze wartość 154. Jak łatwo się domyślić, na jednym bajdzie jesteśmy w stanie maksymalnie zapisać 256 stanów, zaś zakres liczbowy będzie wynosił 0-255 (bit maksymalnie w prawo może posiadać wartość 0 lub 1 – zerowa potęga). W matematyce występują także liczby ujemne. Celem poradzenia sobie z tym problemem w komputerach najstarszy bit (w przykładzie posiadający najwyższą potęgę) traktowany jest jako wartość ze znakiem – jeżeli posiada ustawiony bit to uzyskujemy wartość -128 (a nie 128). To oznacza, że ze znakiem możemy maksymalnie zapisać liczby od -128 do 127 (na wskazanym zakresie). Im większą liczbą bitów dysponujemy tym większe liczby możemy zapisywać.

CIEKAWOSTKA: Grupy bitów mogą być czytane i przesyłane na różne sposoby. W codziennym użytku są urządzenia, które czytają grupy od najstarszej części (big endian), od najmłodszej części (little endian – przedstawiony powyżej przykład odnosi się do tego typu rozwiązania) oraz takie, dla których odczyt danych może następować z dowolnej strony (endianness).

Przykład dobitnie pokazuje, że zapisane informacje, w postaci surowej, są dla nas, ludzi, zupełnie nieprzydatne. Dopiero po określonym nadaniu znaczenia pojedynczemu zapisowi, jesteśmy wykorzystać zapisane dane w odpowiedni sposób. Niestety (lub na szczęście) jedyną formą interpretacji danych jest zapis liczbowy (w dowolnym formacie – ósemkowym, szesnastkowym, dziesiętnym itd.). Jak więc otrzymujemy znaki? Bardzo prosto – poprzez odpowiednie ponumerowanie alfabetu (oraz wszystkich znaków pisanych i niepisanych), a następnie odpowiednie rysowanie (zapełnianie fragmentu ekranu odpowiednimi wartościami każdego reprezentującego go piksela).

Oznacza to, że każda wartość zapisywana w komputerze MUSI posiadać swój typ.

Dla komputera najbardziej naturalny jest typ bitowy (0 – fałsz, 1 – prawda) czyli typ boolean. Należy pamiętać, że w przypadku Java bool nie ma określonej wielkości.

Następnym typem, dostępnym w Java, jest Integer (int, Int). Typ ten to liczby całkowite ze znakiem. Rozróżniamy następujące grupy liczb całkowitych:
- byte – zawiera dokładnie 8 bit (ze znakiem)
- short – typ 16 bitowy
- int – typ 32 bitowy
- long – typ 64 bitowy

Liczby ze znakiem można zapisywać w typach zmiennoprzecinkowych (po angielsku floating point). Istnieją dwa typy podstawowe dla tego typu liczb:
- float – liczba zmiennoprzecinkowa pojedynczej precyzji (32 bitowa), gdzie przedział bitów przeznaczonych na ułamek wynosi 8-24 bity
- double – liczba zmiennoprzecinkowa podwójnej precyzji (64 bitowa), gdzie przedział bitów przeznaczonych na ułamek wynosi 11-53 bity

Ostatnim typem prostym, pozwalającym na zapis znaków jest znak (charakter, w Java zapisywany char). Posiada on wielkość 16 bit. Pozwala zapisać do 65 535 znaków.

Prócz powyżej wymienionych podstawowych typów Java posiada typy złożone. Jednym z popularniejszych jest typ String pozwalający na zapis danych tekstowych większych niż jeden znak. Ponadto istnieją zmienne grupowe (tablice, listy, słowniki itp.), które pozwalają ze sobą łączyć poszczególne typy.

Co jednak, jeżeli potrzebne będzie zamienić wartość „124” (ciąg znakowy) na typ liczbowy? Java posiada odpowiednie metody zamieniające wartości jedne na drugie. Wspomniany powyżej przypadek można rozwiązać poprzez komendę:

```
String test = "124";
int val = Integer.parseInt(test);
```

Poprzez klasy odpowiadające za dany typ (Integer, Long, String, Boolean itd.) możemy dowolnie modyfikować zapisane bajty z jednego typu w drugi. Należy jednak pamiętać, że niekiedy zajmuje to cenny czas procesora – dlatego nie należy tego przesadnie nadużywać. Prócz konwersji obiektowych można też wykorzystywać konwersje szybkie, takie jak np.

```
int val = 34;
float val2 = val;
```

W tym przypadku zmienna zostanie przemieniona NIEJAWNIE.

Przykład jawnej zamiany:

```
float val = 34.2f;
int val2 = (int)val;
```

W powyższym przypadku JAWNIE informujemy kompilator o zmianie wartości na całkowitą (przy utracie wartości po przecinku).

Materiały uzupełniające:

- informacje na temat typów w Java - https://docs.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html
- konwersje typów w Java - http://imagejdocu.tudor.lu/doku.php?id=howto:java:how_to_convert_data_type_x_into_type_y_in_java

4. Złożone typy zmiennych.

Prócz wcześniej przestawionych typów, Java posiada ich odpowiedniki w postaci obiektowej. Obiekty dostarczają przede wszystkim niezbędnych funkcjonalności, jak między innymi możliwość konwersji pomiędzy typami. Ponadto obiektowe wersje danych prostych mogą być stosowane z innymi typami złożonych zmiennych – kolekcjami danych.

Kolekcje danych to szereg udogodnień. Obecnie Java posiada sporą liczbę interfejsów gromadzących (np. List, Set, Deque) jak i klas (np. ArrayList, LinkedList, HashMap, HashTable, TreeMap, EnumSet). Programista może używać naprzemiennie bardziej potrzebnych w danym momencie rozwiązań. Każde z nich cechuje inna funkcjonalność; interfejsy nadają się do prostego składowania danych, nie pozwalają na swobodny dostęp do dowolnego elementu kolekcji, natomiast klasy jak najbardziej umożliwiają takie operacje. Ponadto, w zależności od interfejsu oraz wykorzystanego algorytmu/framework otrzymujemy mniej/bardziej wydajane sortowanie, mniej/bardziej wydajne przeszukiwanie danych, mniej/bardziej wydajne zamieniane elementów itd. Generalnie używanie kolekcji można sprowadzić do jednego czy dwóch mechanizmów, jednak nie zawsze takie wariant będzie niezawodny.

Poniżej najważniejsze cechy poszczególnych interfejsów:
- Set – nie pozwala na duplikowanie wartości w zestawie; wykorzystuje algorytmy mieszające (hash) celem szybkiego wynajdywania elementów w kolekcji (algorytm sprawdzający czy oba wyniki mieszania są równe).
- List – kolekcja umożliwiająca najlepsze sortowanie zawartości; pozwala na duplikowanie wartości, pozwala na efektywne przeszukiwanie, przemieszczanie się po liście oraz tworzenie pod-list (zakresy widoku na zebrane dane)
- Queue – kolejka jest dobrym rozwiązaniem do przechowywania danych przygotowanych do późniejszego przetwarzania. Domyślnie działa wedle zasady FIFO (First In – First Out, czyli Pierwszy wszedł, pierwszy wyjdzie)
- Stack – implementacja stosu znanego z działania komputera. Działa wedle zasady LIFO (Last In First Out, czyli Ostatni Wszedł, Pierwszy Wyjdzie).
- Deque – podobny do kolejki z tym, że operacje można przeprowadzać zarówno na początku jak i jego końcu. Implementuje zarówno interfejs Queue jak i Stack (i przejmuje ich najlepsze cechy)
- Map – kolekcja, w której każdy klucz odpowiada wskazanej wartości. Klucze i wartości mogą być różnego typu (np. klucz będzie typu String, a wartości typu Integer). Klucze nie mogą się duplikować. Wartości oraz klucze są najczęściej mieszane, przez co przeszukiwanie tego typu kolekcji jest dość szybkie i efektywne.
- SortedSet – zestaw, który domyślnie ustawia zwarte w nim elementy narastająco (ascend).

- SortedMap – podobnie do SortedList, jednak ustawia dane narastająco po wartości klucza.

Najważniejsze klasy kolekcji oraz ich cechy:
- HashSet – najszybsza implementacja interfejsu Set; nie gwarantuje kolejności iteracji (przemieszczania się po danych)
- TreeSet – porządkuje dane wedle algorytmu drzewa czerwono-czarnego; gwarantuje ciągłość iteracji jednak jest wolniejsza
- LinkedHashSet – dane porządkowane są wedle ich dołączania do kolekcji; tym samym jest tylko nieco wolniejsza od HashSet, jednak mniej chaotyczna
- ArrayList – lista oparta o tradycyjne tabele, posiada dobrą wydajność w przypadku magazynowania danych
- LinkedList – jest lepsza do zarządzania danymi (przesuwanie, podmiana, usuwanie) ze względu na fakt implementacji listy podwójnie podłączanej (doubly linked list).
- HashMap, TreeMap, LinkedHashMap – zachowują się analogicznie do HashSet, TreeSet oraz LinkedHashSet.

Więcej informacji na temat kolekcji i typów złożonych:
- https://www.javatpoint.com/collections-in-java
- https://docs.oracle.com/javase/tutorial/collections/interfaces/collection.html
- https://docs.oracle.com/javase/8/docs/api/?java/lang/Integer.html
- https://docs.oracle.com/javase/8/docs/api/java/lang/String.html
- https://docs.oracle.com/javase/8/docs/api/java/lang/Double.html
- https://docs.oracle.com/javase/8/docs/api/java/lang/Float.html
- https://docs.oracle.com/javase/8/docs/api/java/lang/Boolean.html
- https://docs.oracle.com/javase/8/docs/api/java/lang/Byte.html
- https://docs.oracle.com/javase/8/docs/api/java/lang/Character.html

5. Polimorfizm i rekurencja

a) przeciążanie funkcji (polimorfizm metod)

Każda metoda w Java ma ściśle sprecyzowane zadanie. Może zwracać określoną wartość (bądź nie zwracać żadnej wartości) a także wymagać określonej ilości parametrów (bądź nie wymagać żadnego parametru). Dodatkowo zawsze wykonuje dokładnie te same zadania (w założeniu przynajmniej jedno).

Niekiedy zdarza się, że chcemy dać możliwość użytkownikowi naszej klasy (lub sobie samym) wywołania tej samej metody, robiącej dokładnie to samo, jednak z innymi parametrami początkowymi. Przeanalizujmy przykład.

Do tej pory mieliśmy funkcję writeText(), której zadaniem było wypisywanie tekstu:

```
public void writeText() {
	System.out.println("Witaj w Java");
}
```

Chcemy jednak by prócz standardowego tekstu wypisała nam także tekst dodany jako parametr:

```
public void writeText(String p) {
	System.out.println("Witaj w Java " + p);
}
```

Problem w tym, że dodatkowy tekst nie zawsze będzie pożądany, a programista będzie zmuszony do wywołania tej funkcji zawsze z parametrem. Zamiast

writeText();

będzie musiał każdorazowo napisać:

public void writeText("");

Dodatkowo na końcu tekstu zawsze będzie widniała biała przestrzeń, która, przy kopiowaniu bądź obróbce danych, może mieć negatywny wpływ na działanie całego programu.

Powyższy problem można rozwiązać w następujący sposób:

```
public void writeText() {
        System.out.println("Witaj w Java");
}

public void writeText(String p) {
        System.out.println("Witaj w Java " + p);
}
```

Teraz programista może wywołać funkcję w dwa następujące sposoby:

writeText();

lub

writeText("Nowy tekst");

Powyższy przykład będzie dobry do czasu, gdy nie zajdzie potrzeba zmiany stałego tekstu (Witaj w Java). Nasz przykład, ze względu na swoją prostotę, przy takie zmianie wymagać będzie interwencji jedynie w dwóch miejscach. Co jednak, jeżeli nasza metoda miała by więcej form? Przykładowo:

```
public void writeText(int a) {
      System.out.println("Witaj w Java " + a);
   }

public void writeText() {
        System.out.println("Witaj w Java");
}

public void writeText(String p) {
        System.out.println("Witaj w Java " + p);
}
```

W tym momencie programista może przekazać jako parametr liczbę, która ostatecznie będzie miała zostać wyświetlona za tekstem stałym. Tego typu przeciążeń może być znacznie więcej Im ich

więcej – tym więcej ingerencji przy potencjalnej zmianie. Stąd dobrym rozwiązaniem jest przebudowanie ciał metod w taki sposób, by ewentualne zmiany odbywały się tylko w jednej z nich. Przykład:

```
public void writeText(int a) {
      writeText(String.valueOf(a), new String());
   }

   public void writeText() {
         writeText("", "");
   }

   public void writeText(String p) {
      writeText(p, " ");
   }

   public void writeText(String p, String sep) {
      if (sep.isEmpty()) sep = ": ";
      System.out.println("Komunikat z programu" + sep + p);
   }
```

Jak widać docelowa część do wykonania przeniesiona została do ostatniej funkcji. Dodatkowo pierwotna funkcja została wyposażona z drugi parametr, odpowiadający za separator pomiędzy tekstem stałym a wprowadzanym jako parametr.

Sama funkcja wzbogacona została o dodatkową linię, dzięki której bada wprowadzony separator. Jeżeli programista nie podał żadnego separatora to ustawia mu wartość domyślną (dwukropek ze spacją). W przypadku obecnego tekstu tego typu rozwiązanie jest jak najbardziej wskazane.

UWAGA: Wywołane powyżej new String() ma dokładnie taki sam efekt jak podany niżej "" (pusty znak). Obie wersje zostaną rozpoznane przez if(sep.isEmpty()) jak prawda (ponieważ oba są puste!)

INFORMACJA: Należy pamiętać, że przeciążane funkcje mogą posiadać także inne parametry zwrotu wartości. Przykładowo jedna (lub kilka) funkcji może zwracać wartość (różnej postaci) inne zaś mogą nic nie zwracać. Jeżeli funkcje będą zwracać różne wartości możemy przechwycić ich wartości w typ Object:

```
public String writeText(int a) {
      writeText(String.valueOf(a), new String());
      return "Liczba";
   }

   public int writeText() {
         writeText("", "");
      return 0;
   }

//dalsza część kodu

Object zwrot1 = new Paczka().writeText();
zwrot1 = new Paczka().writeText(12);
```

System.out.println(zwrot1);

Przykład wyświetli nam tekst „Liczba”. Gdybyśmy nie podmienili zawartości danej otrzymalibyśmy komunikat „0”.

b) polimorfizm

Ostatni przykład z przeciążania obrazuje dodatkową cechę obiektów języku Java – polimorfizm obiektów. Każdy z obiektów ma swój typ, jednak wszystkie bazują na głównym obiekcie Object.

Object w ogólnym zarysie w języku Java nie jest bezpośrednio dostępny dla programisty. Programista ma dostęp do obiektów poprzez referencję. W danej chwili referencja może wskazywać na obiekt JEDNEGO TYPU (String, Boolean, Integer itd. bądź Object). Jednak, jeżeli referencja nie posiada modyfikatora final możliwe jest przepisanie jej wartości (przewartościowanie) na inny typ (zmiana referencji). Dodatkowo zmienne referencyjne mogą odwoływać się zarówno do klas jak i interfejsów.

Najprostszym przykładem polimorfizmu może być kolekcja. Domyślnie wywołana zmienna:

```
List<String> str = new ArrayList<String>();
str.add("Pierwszy");
str.add("Drugi");
str.add("Trzeci");
```

Powyżej mamy utworzoną zmienną typu List, która sama w sobie jest jedynie interfejsem. Ponieważ jednak interfejs ten jest wykorzystywany między innymi jako baza dla ArrayList tworzymy na tej podstawie zmienną, która ma cechy klasy-kontruktora obiektu. W poprzednim (czwartym) punkcie wspomniane było, że List jest interfejsem także dla LinkedList. Nic nie stoi na przeszkodzie by zrobić coś takiego:

```
str = new LinkedList<>();
str.add("Pierwszy linked");
str.add("Drugi linked");
str.add("Trzeci linked");
```

Referencja pozostała tego samego typu, zmienił się typ obiektu z ArrayList na LinkedList. Poprzednia lista została utracona, otrzymaliśmy nową.

Jednak jeżeli próbowalibyśmy wykonać taką operację:

```
str = new Hashtable<>();
```

To spowoduje to błąd kompilacji. Dzieje się tak dlatego, że HashTable nie należy do naszego typu (w typ wypadku nie implementuje interfejsu List).

Taki sam problem pojawi się w poniższym przypadku:

```
LinkedList<String> str2 = new LinkedList<>();
str2 = new ArrayList<>();
```

Ponieważ od samego początku założyliśmy, że zmienna referencyjna ma być typem kolekcji LinkedList straciliśmy możliwość przepisania jej na typ ArrayList.

INFORMACJA: Od wersji 7 w Java można, zamiast powielać znany typ kolekcji (w naszym przypadku String) podczas inicjalizacji zmiennej, użyć tzw. typu diamentowego (diamentowy operator). Zapis jest krótszy a działa tak samo. W pierwszym przykładzie wykonano inicjalizację standardową, w kolejnych skorzystano z diamentu (działanie identyczne).

c) rekurencja

Tworząc programy nie zawsze możemy mieć możliwość wykonania wszystkich operacji porzez standardowe mechanizmy decydujące i pętle. Niekiedy nie będziemy mieli możliwości ocenić ile pętli potrzebujemy wykonać oraz w którym momencie je przerwać. Czasami też napisanie pętli for będzie po prostu za długie/będzie wymagać więcej zasobów maszyny niż jest tego warte.

Większość języków, w tym i Java, pozwala na stosowanie rekurencji. Rekurencja, jak sama nazwa wskazuje, to wywoływanie np. metody przez samą siebie. Dość prostym, aczkolwiek charakterystycznym przykładem na rekurencję jest obliczanie silni (factorial).

INFORMACJA: Silnia to operacja matematyczna (arytmetyczna) pozwalają na przemnażanie liczby przez każą jej składową. Symbolem silni jest wykrzyknik. Przykładowo silnia:

5! = 1*2*3*4*5=120

Silnia zawsze ma wartość dodatnią naturalną. Oznacza to, że 0! wynosi 1 (tak samo jak 1! wynosi 1).

Jak łatwo się domyślić, silnię w Java można zaimplementować w następujący sposób:

```
public long silnia(long a) {
        long ret = 1;
        for (long i = 1; i <= a; i++)
                ret *= i;
        return ret;
}
```

Jak można zauważyć kod będzie działał poprawnie. Jednak w jego przypadku musieliśmy stworzyć następujące zmienne:

ret – do zachowania wyniku
i – do przemieszczania się po kolejnych wartościach naszej silni

Dodatkowo trzeba było stworzyć pętlę celem obliczenia wartości silni.

Rekurencja znacznie upraszcza zapis kodu oraz powoduje, że jej obliczenie będzie znacznie efektywniejsze:

```
public long silnia(long a) {
if (a > 0)
    return a*silnia(--a);
return 1;
}
```

lub jeszcze bardziej zwięźle (przy wykorzystaniu operatora trójstanowego):

```
public long silnia(long a) {
	return (a > 0) ? a*factorial(--a) : 1;
}
```

Jak to działa? Pozostańmy przy wartości 5. Taka wartość przypisana zostanie do zmiennej a.

Funkcja od razu przechodzi do instrukcji powrotu z funkcji (return). W tym jednak momencie pojawia się operator trójstanowy.

Warunek sprawdza czy a jest większe od zera. Ponieważ jest jest (mamy do czynienia z wartością 5) wykonuje się kod po znaku zapytania. W nim to a przemnażane jest PRZEZ WYNIK METODY SILNIA, jednak z wartością o jeden mniejszą. W tym wypadku –a można zamienić na a-1 – efekt będzie ten sam (--a wykonuje się w chwili osadzenia a jako parametru i nie ma wpływu na wartość sprzed operatora mnożenia!).

Rozpoczyna się wykonywanie TEJ SAMEJ METODY jednak z wartością o jeden mniejszą (4). Ponieważ jest ona większa od 0 instrukcje wykonają się tak samo jak akapit wyżej.

Metoda będzie tak schodzić coraz niżej do chwili gdy nie napotka wartości 0. Wtedy to zwróci ona po prostu 1. Od tego momentu następować będzie powrót do programu głównego, a wraz z nim tworzenie właściwego wyniku. Ogólnie operacja powrotu będzie wyglądała następująco:

- zwróć wartość 1
- przemnóż wartość 1 przez zwróconą zwróconą 1
- przemnóż wartość 2 przez zwróconą wartość 1
- przemnóż wartość 3 przez zwróconą wartość 2
- przemnóż wartość 4 przez zwróconą wartość 6 (nastąpiło już mnożenie 3*2)
- przemnóż wartość 5 przez zwróconą wartość 24
- zwróć do punktu wywołania wartość 120

Najczęstszym zastosowaniem rekurencji może być:
- działania arytmetyczne, geometryczne, analityczne, które nie mają jasno ustalonego elementu końca LUB ich implementacja jest zbyt kosztowna (ilość zmiennych oraz pętli)
- działanie na zbiorach, które nie mają jasno określonych granic (np. przeszukiwanie katalogów w systemie plików)
- wyszukiwanie informacji w często zmieniających się zbiorach (przykładowo najszybsza trasa dla przejazdu samochodem – problem komiwojażera)
- tworzenie modeli fraktali (problemy matematyczne, które posiadają swoje odpowiedniki w świecie rzeczywistym – np. liść paproci, płatek śniegu)

Więcej informacji:
- przeciążanie i polimorfizm
https://beginnersbook.com/2013/03/polymorphism-in-java/
https://www.tutorialspoint.com/java/java_polymorphism.htm
https://docs.oracle.com/javase/tutorial/java/IandI/polymorphism.html
- rekurencja
https://introcs.cs.princeton.edu/java/23recursion/
http://www.toves.org/books/java/ch18-recurex/