Java – podstawy wersja 23-11-2018

Spis treści

Słowo wstępu

Programy wykorzystane w poradniku znajdują się na moim githubie.

Aby zaimportować sobie wszystkie pliki należy w IntelliJ Idea wybrać kolejno:

File->New->Project from Version Control->GitHub->Git Repository URL->

i wkleić tutaj https://github.com/idzikdev/HighSchool.git

i od tej pory nie trzeba kopiować osobno plików. Macie wszystkie programy na dysku.

Informacje o języku

Język programowania Java powstał na początku lat 90tych ubiegłego stulecia. Jest on w tej chwili najpopularniejszym językiem w świecie komputerów. Aplikacje napisane przy jego użyciu znajdziemy we współczesnych urządzeniach takich jak smartphone, smatwatch, a nawet pralki, lodówki czy samochody. Aplikacje w Javie obsługują również strony internetowe. Pozwalają nam rozmawiać z innymi ludźmi przez czata.

Poniżej widzimy najprostszy program, który znajduje się w każdej książce o programowaniu. Poniższy program ma za zadanie wyświetlić napis „Witaj Świecie”.

```
public class Zadanie0 {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

Język Java jest językiem programowania wysokiego poziomu. Oznacza to, że nie trzeba już pisać kilkunastu instrukcji, aby wyświetlić napis tak jak to było na początku lat 90tych. Wystarczy napisać instrukcję println(). Wadą takiego rozwiązania jest fakt, że Java tak samo jak inne współczesne języki nie ma bezpośredniego dostępu do sprzętu komputera.

Dlaczego Java jest taka popularna? Bo można uruchamiać ją wszędzie tam gdzie jest zainstalowane JVM czyli wirtualna maszyna Javy. To ta maszyna odpowiada za interpretację skompilowanych kodów bajtowych Javy na język zrozumiały dla urządzenia.

Ale od początku. Do edytora wpisujemy kod źródłowy. Pliki źródłowe Javy to pliki o rozszerzeniu *.java. Kompilator javac tłumaczy te pliki do postaci plików z rozszerzeniem *.class. Te ostatnie są interpretowane i wykonywane wszędzie, gdzie jest dostępne JVM, czyli na wspomnianych już wszelkiej maści urządzeniach.

Należy tutaj wspomnieć, że język Java jest oficjalnym językiem programowania na system Android. Dobrze się domyślacie. Brawo. Każdy, kto potrafi programować w Java nie będzie miał problemów z szybką nauką pisania aplikacji na Androida.

Jak już wspomniano, aby uruchamiać programy napisane w Javie trzeba mieć JVM. Należy więc zainstalować JRE czyli Java Runtime Environment. Nie wystarczy to jednak, aby pisać własne programy. Wtedy należy zainstalować JDK, które już w sobie zawiera JRE. Po

zainstalowaniu JDK jesteśmy gotowi na…. pisanie programów w notatniku. Czego jeszcze brakuje? Jak myślicie?

Do wygodnej pracy należy zainstalować środowisko programistyczne np.Eclipse, NetBeans lub IntelliJ Idea. Przecież nie chcemy pisać własnych programów w notatniku, kompilować pisząc javac i uruchamiać pisząc java. Ze swojej strony polecam oprogramowanie firmy JetBrains o nazwie IntelliJ Idea. Jest do świetny program do pisanie różnorodnych aplikacji konsolowych, okienkowych i webowych w języku Java. Program ten używa inteligentnego podpowiadania, które korzysta z tzw. refleksji, o których będzie mowa później. Na silniku IntelliJ Idea powstało środowisko Android Studio – najlepszy program do programowania aplikacji na Androida.

Zmienne

Języki programowania nie powstałyby, gdyby ich celem było wyświetlanie napisu. Każdy programista prędzej czy później znudzi się pisaniem programów, w których wyświetlanie są napisy. Aby pisać lepsze programy trzeba się nauczyć używać zmiennych. Będziemy wtedy w stanie napisać program, który np. dodaje do siebie dwie liczby. Powstaje pytanie gdzie będziemy te liczby przechowywać?

Wartości liczbowe i znakowe przechowujemy w zmiennych. Każda zmienna ma swój ściśle określony typ. Ten typ obowiązuje od początku do zakończenia programu. Przez cały ten okres typ zmiennej i jej zakres się nie zmienia. Taką cechę nazywamy **statycznym typowanym**

Jakie są typy zmiennych? A jakie dane zwykle będziemy przechowywać w programie. Większość odpowie, że liczby i ciągi znaków. Macie rację. W Javie możemy przechowywać **liczby** (całkowite, rzeczywiste) oraz **ciągi znaków**.

Wiemy już co to jest typ zmiennej. Czym będzie zakres zmiennej? Zastanówmy się czy komputery mogą przechowywać liczby o dowolnej długości? Odpowiedz brzmi NIE. Nic nie jest nieskończone. Nie można w programie przechować nieskończonej liczby. Po pierwsze: przecież ona nie istnieje, ale nawet gdyby istniała, to ile czasu zajmie nam wpisanie jej cyfr do komputera. Zgadzamy się zatem, że komputer jest w stanie przechować liczby o określonej długości. Dobrze. W takim razie jakiej długości lub raczej wielkości są to liczby? Tę wielkość nazywamy rozmiarem zmiennej lub jej **zakresem**.

Java pozwala nam deklarować następujące typy całkowite: byte, short, int, long. Każdy typ jest **typem ze znakiem**. Oznacza to, że w każdym z nich jesteśmy w stanie zapisać nie tylko liczbę dodatnią i zero, ale także liczbę ujemną. Poniższa tabela przedstawia zakres tych typów.

Nazwa typu	Rozmiar w bajtach	zakres
byte	1	-2^7….2^7-1
short	2	-2^15….2^15-1
int	4	-2^31….2^31-1
long	8	-2^63….2^63-1

I od razu ciekawostka. Podany powyżej rozmiar w bajtach jest taki sam dla każdego urządzenia. Nie jest ważne czy jest to komórka czy samochód. Dlatego Java jest tak ciekawym, prostym i porządnym językiem.

Należy pamiętać, że **domyślnym typem całkowitym** dla operacji artytmetycznych jest typ int. Oznacza to, że typem wyniki dla tych operacji będzie int. Do operacji artytmetycznych zaliczamy między innymi: +,-,/,*,%. W Javie znak / oznacza dzielenie całkowite, a więc należy pamiętać, że dostaniemy resztę z takiego dzielenia.

Wśród **typów rzeczywistych** znajdziemy : float i double. Domyślnym typem jest double i zajmuje one więcej miejsca w pamięci niż float. Oznacza to, że jest on typem bardziej precyzyjnym.

Typ znakowy to typ char. Do tego typu możemy podstawiać znaki jako wartości. Należy pamiętać, że typ char może również przechowywać liczby całkowite typu int. Oznaczają one po prostu kod ASCII znaku.

Czy nie wydaje się wam, że brakuje tu jeszcze jednego typu? Brakuje! A gdzie będziemy przechowywali informacje o tym czy np.: liczba a jest większa od zera? Brakujący typ to **typ logiczny boolean**. Może on przyjmować jedną z dwóch możliwych wartości: true albo false.

Wszystkie typy przedstawione powyżej są nazywane typami prostymi lub prymitywnymi, w literaturze **primitive data types**. Są one przechowywane w specjalnej szybkiej pamięci.

Pierwszy program

Omówimy teraz części składowe programu napisanego w języku Java.

```
public class Zadanie0 {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

Od teraz zaczynamy tłumaczenie kolejnych zagadnień. Od tej pory będzie wyjaśniane tylko to, co jest potrzebnei w taki sposób jaki jest niezbędny . Zaawansowani programiści mogą się nie zgadzać do końca z pewnym rzeczami. Zabieg ten jednak jest celowy i prowadzi do stopniowego i lepszego zrozumienia programowania przez początkujących programistów.

Siedziecie wygodnie? To zaczynamy!

Każdy program napisany w Java musi się **zmieścić w klasie** a więc między nawiasami klamrowymi…o tutaj

```
public class Zadanie0 {
    TUTAJ JESTEM
}
```

Klasa w której umieszczamy nasz program musi mieć taką samą nazwę jak nazwa pliku. To nie wystarczy, żeby uruchomić program. Aby uruchomić program, musimy **zaimplementować** (napisać) metodę główną w programie. Nazywa się ona main i wygląda następująco:

```
public static void main(String[] args) {
    TUTAJ WPISUEJMY NASZ KOD
}
```

Aby nasz program działał wystarczy wpisać kod między klamry.

Wyświetlmy więc napis Hello World.

```
public class Zadanie0 {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

Co robi instrukcja System.*out*.println() ? Wyświetla napis podany w cudzysłowie. Teraz aby zobaczyć efekt należy skompilować program i uruchomić.

Wskazówka. Aby nie pisać ostatniej instrukcji można skorzystać z podpowiedzi. Jeśli kursor znajduje się w main, napisz sout, poczekaj chwilkę i naciśniej enter. Dostaniesz całą instrukcję ;) Wystarczy teraz wpisać Hello Word.

Jeśli nie chce ci się pisać, program znajdziesz na moim githubie. Jeśli skopiujesz sobie ten program, to zastanów się w jakim celu tu jesteś? Aby kopiować kod czy uczyć się pisząc program i wyłapywać błędy. Mam nadzieję, że nie skopiujesz już żadnego programu i będziesz sam programował.

Deklaracja i definicja

Zajmiemy się teraz deklarowaniem zmiennych w naszym programie.

Załóżmy, że chcemy napisać program, który dodaje do siebie dwie liczby całkowite. Musimy wtedy zadeklarować dwie liczby całkowite.

```
int firstNumber;
```

Tak wygląda **deklaracja zmiennych całkowitej typu int o nazwie firstNumber**. Ale co ona oznacza? Kompilator ma zarezerwować sobie w pamięci 4 bajty. Zarezerwować to znaczy, że żadna inna zmienna nie będzie tutaj przechowywała swoich wartości poza zmienną firstNumber. Kiedy już zarezerwujemy sobie miejsce w pamięci, to możemy do naszej zmiennej przypisać jakąś wartość.

```
firstNumber=5;
```

Tak wygląda **definicja zmiennej całkowitej tpu int o nazwie firstNumber**. Od teraz w zarezerwowanych komórkach pamięci operacyjnej RAM będzie znajdowała się wartość 5.

Ustaloną wartość można oczywiście zmieniać. Robimy to poprzez nadpisanie definicji czyli

```
firstNumber=15;
```

Od tej chwili zmienna firstNumber będzie miała wartość 15.

Zadeklarujemy drugą zmienną. Nasz program będzie teraz wyglądał następująco:

```
public class Zadanie1 {
    public static void main(String[] args) {
        int firstNumber=15;
        int secondNumber=13;
       }
}
```

Czego nam jeszcze brakuje? Co mamy zrobić? Przypominam, że powinniśmy obliczyć sumę dwóch liczb. W tym celu potrzebujemy jeszcze jednej zmiennej, która będzie przechowywała nasz wynik. Zróbmy to:

```
int suma=firstNumber+secondNumber;
```

I to wszystko. Trzeba jeszcze wyświetlić wynik…Po wszystkich zmianach nasz program będzie wyglądał następująco:

```
public class Zadanie1 {
    public static void main(String[] args) {
        int firstNumber=5;
        int secondNumber=13;
        int suma=firstNumber+secondNumber;
        System.out.println("The sum is "+suma);
    }
}
```

Zastanówmy się teraz, czy można zoptymalizować nasz program? Pewnie, że można. Może po prostu dodawanie wykonamy podczas wyświetlania wyniku?

```
System.out.println("The sum is "+firstNumber+secondNumber);
```

Po skompilowaniu i uruchomieniu programu dostajemy

The sum is 513

Czy wiemy dlaczego? Nie wiemy. Dlatego wyjaśnię.

Mamy tutaj pierwszy raz do czynienia z bardzo ważną strukturą danych, którą jest klasa **String**. Tej struktury będziemy używali bardzo często. String reprezentuje ciąg znaków. Nie jest on typem prymitywnym. Świadczy o tym np.: to, że piszemy go wielką.

Co oznacza taki zapis ?

```
"The sum is "+5+13
```

Zwróćmy uwagę, że po lewej stronie stoi String . Bardzo silnie przyciąga on elementy stojące po jego prawej stronie. Tak silnie, że dokleja do siebie kolejne elementy, a więc na początku dosteniemy

```
"The sum is 5"+13
```

I znowu nastąpi doklejenie. Tym razem wynikiem będzie

```
"The sum is 513”
```

I wszystko jasne? Ktoś zada pytanie czy da się dokleić 18 ? Da się tylko, że trzeba zapisać dodawanie w nawiasie okrągłym.

```
System.out.println("The sum is "+(firstNumber+secondNumber));
```

Tutaj widzimy

```
"The sum is "+(firstNumber+secondNumber)
```

Teraz String znowu próbuje dokleić do siebie wyrażenie z prawej strony, ale natrafia na opór. Nawias mówi, aby poczekał chwilę, aż wyrażenie w nawiasie zostanie obliczone. Zostanie wykonane:

```
"The sum is "+(18)
```

Liczba 18 jako wynik może już zostać doklejona. I to cała tajemnica!

Wiemy już więc jak deklarować i definiować typy całkowite. Wiemy również, że trzeba uważać na klasę String, z którym już niedługo ponownie się spotkamy.

Podobnie deklarujemy inne typy całkowite, rzeczywiste, logiczne i znakowe.

Jako uzupełnienie wiedzy polecam filmy mojego autorstwa:

Wyświetlanie napisu

Deklaracja i definicja zmiennych typów całkowitych

Deklaracja i definicja zmiennych typów rzeczywistych

Deklaracja i definicja znaków

Deklaracja i definicja typu logicznego

Dodawanie dwóch liczb

Rada: Nie trzeba pisać osobno deklaracji i definicji. Można jedno i drugie napisać w jednej instrukcji i w jednym wierszu.

```
int firstNumber=15;
```

Klasa String

Nadeszła pora, aby opowiedzieć o wyjątkowej strukturze danych którą jest **String**. Przechowuje ona łańcuchy znaków. Jego deklaracja wygląda następująco:

```
String word;
```

Jak widać nie różni się ona od deklaracji typów prostych. Nie tylko to sprawia, że String jest podobny do prymitywnych typów danych. Dzieje się tak również dlatego, że jego definicja jest także bardzo prosta.

```
word="Napis";
```

Co jak już wiemy możemy zapisać równoważnie jako:

```
String word="Napis";
```

Spróbujmy teraz porównać dwa Stringi. Zdefiniujemy je następująco:

Pierwszy sposób definiowania Stringa

```
String napis1="Java";
String napis2="Java";
if (napis1==napis2) System.out.println("Takie same");
else System.out.println("Różne");
```

Jak myślicie jaki będzie wynik porównania Stringów o takich samych zawartościach? Założę się, że podacie wynik TAKIE SAME. Macie rację chociaż nie wiecie że dzieje się tak przez przypadek. Zaraz dowiecie się dlaczego.

Zdefiniujmy teraz Stringi tak jak na Stringi przystało. String jest klasą a klasy definiuje się z użyciem słowa kluczowego new. Będzie o tym jeszcze mowa na późniejszym etapie.

Drugi sposób definiowania Stringa

```
String napis1=new String("Java");
String napis2=new String("Java";)
if (napis1==napis2) System.out.println("Takie same");
else System.out.println("Różne");
```

Jaki będzie wynik? Pewnie znowu powiecie TAKIE SAME. Teraz nie macie racji mimo, że zawartość każdego Stringa jest cały czas identyczna!

Co się tutaj dzieje?

Przeanalizujmy jeszcze raz pierwszy przypadek. Przecież napisaliśmy ==, Stringi są takie same, instrukcja IF jest ok? Coś jest nie tak? Okazuje się, że wszystko działa jak powinno. A przyczyną jest niewłaściwe użycie znaku == oraz sposób definicji Stringów.

Po pierwsze należy zapamiętać, że znaku porównania == (podwójny znak równości) używamy jedynie w przypadku porównywania typów prostych, a String nim nie jest! Co wobec tego jest porównywane? Porównywane są zmienne napis1 i napis 2 a nie Stringi! Napis1 i napis2 to odwołania do pamięci w której jest obiekt String zapisany! To dlaczego nie wyszło RÓŻNE!! A dlatego, że w tym przypadku zmienna napis1 i napis2 wskazuje na to samo miejsce pamięci. Dzieje się tak dlatego, że

1. String napis1 dostaje obszar w pamięci i zostaje tam zapisany napis Java
2. kompilator zauważa, że znowu definiujemy Stringa o takiej samej zawartości i nie ma zamiaru rezerwować kolejnego miejsca w pamięci, w której będzie siedziała taka sama zawartość i dlatego zmienna napis2 będzie wskazywała na to samo co zmienna napis1

Dlaczego kompilator nie chce dawać nowego miejsca w pamięci? Ze względu na oszczędność. A co będzie jeśli zawartość Stringa się zmieni? A właśnie, że nie. Figa z makiem! String jest klasą niezmienniczą więc się nie zmieni. O tym napiszę później.

Taki system zapisu Stringów o tej samej zawartości nazywamy **String Common Pool**.

Przeanalizujmy jeszcze raz drugi przypadek

Tutaj mamy do czynienia z definiowaniem Stringa jak na klasę przystało, a więc przy użyciu słowa kluczowego new. Taki sposób definiowania Stringa sprawia, że jednej zmiennej

i drugiej zmiennej czyli napis1 i napis2 będą przydzielone osobne miejsca w pamięci mimo tego, że będą w nich zapisane te same wartości.

Pozostaje jeszcze do rozwiązania jeden ważny problem. Dlaczego jeśli we właściwy sposób definiujemy String (czyli używając new) to jednak nie są one takie same mimo, że mają taką samą zawartość?

Musimy zapamiętać, że operator == porównuje typy proste, a do nich String nie należy. String jest klasą i jak na klasę przystało powinniśmy używać metody equals().

Zapamiętajmy : ze względu na to, że **String jest klasą**, to do porównywania obiektów klasy String **należy używać equals().** Poniższy program działa już prawidłowo.

```
String napis1="Java";
String napis2="Java";
if (napis1.equals(napis2)) System.out.println("Takie same");
else System.out.println("Różne");
```

Jak zauważyliśmy pojawia się coś nowego (przed equals), a dokładniej operator . czyli operator **kropka**. Oznacza on dostęp do obiektu. Jest to **operator dostępu do obiektu**. Pisząc kropkę wyświetlają się nam wszystkie czynności, które możemy na obiekcie klasy String wykonać. Wśród popularnych metod (czyli czynności) znajdziemy:

1. equals() – porównywanie dwóch stringów

2. lenght() – długość stringa

3.startsWith(ciag) - czy string zaczyna się określonym ciągiem znaków

4. endsWith(ciag) - czy string kończy się określonym ciągiem znaków

5.charAt(indeks) – zwraca element a więc znak, który znajduje się na pozycji indeks w stringu

Jak widać te metody zapisuje się w taki sposób, że na końcu znajdują się nawiasy okrągłe. Zawsze będzie to oznaczało, że mamy do czynienia z metodą czyli czynnością, którą można wykonać na obiekcie. Nawiasy nie zawsze będą puste. Jak widać powyżej czasami potrzebne są argumenty, aby wykonać jakąś metodę. Przykładowo charAt() nie może być pusty, bo nie będzie wiadomo, który znak zwrócić.

Jeszcze jedna ważna rzecz. Ostatnio napisałem, że String jest klasą niezmienniczą. Oznacza to, że nie można zmienić jego elementów czyli znaków. Jak zatem działał program z pierwszych rozdziałów. Dla przypomnienia

```
"The sum is "+18
```

Co tu się działo? Pamiętacie? Wynikiem było

```
"The sum is 18"
```

```
//tutaj poprawka StringBuilder
```

Deklaracja łańcucha

Porównywanie dwóch liczb

Porównywanie dwóch łańcuchów

Programowanie funkcyjne

Zasady

1. Nasze klasy powinny być immutable czyli dajemy **final** (nie będzie więc dziedziczenia).
2. Pola składowe klasy są **inicjalizowane w konstruktorze.**
3. **Nie** tworzymy **seterów**.
4. Aby przeczytać wartości pól składowych tworzymy **getery.**
5. Metody są czyste. Nie modyfikują stanów innych obiektów.
6. Jeśli modyfikujemy jednak obiekt, to zwracamy już inny obiekt, jego kopię.
7. Unikamy nulli. Lepiej zwrócić obiekt klasy Optional<Klasa> niż null.
8. Metody są tak samo ważne jak klasy. Mogą być parametrami innych metod, a także mogą być przypisywane do zmiennych kokalnych.

Co to jest interfejs funkcyjny?

To taki interfejs, który posiada **tylko jedną metodę**

Przykłady:

1. Metoda run() z interfejsu Runnable
2. Metoda compareTo() z interfejsu Comparable

Interfejsy funkcyjne najlepiej oznaczać adnotacją **@FunctionalInterface.**

Java 1.8 to kilkanaście nowych interfejsów, właśnie funkcyjnych

Wyrażenie lambda – od Java 1.8

Są to anonimowe metody, bo nie mają nazwy.

Mogą zastąpić każdy interfejs funkcyjny

(String s)->{sout (c);};

Runnable r=()-> sout (a);

Jeśli metoda coś zwraca, to i tak tego nie piszemy z prawej strony, bo domyślnie jest return.

Nie trzeba nawet pisać jak jest typ zmiennej z lewej. Java powinna się domyślić ;)

Klasa Student i klasa Indeks

```
final public class Indeks {
private String indexNumber;

    public Indeks(String indexNumber) {
this.indexNumber = indexNumber;
}

public String getIndexNumber() {
return indexNumber;
}
}
```

```
final public class Student {
private String name;
    private int age;
    private Indeks index;

    public Student(String name, int age,String indexNumber) {
this.name = name;
        this.age = age;
        this.index = new Indeks(indexNumber);
}

public String getName() {
return name;
}

public int getAge() {
return age;
}
```

```
public Indeks getIndex() {
return index;
}

@Override
public String toString() {
return "Student{" +
"name='" + name + '\'' +
", age=" + age +
", index=" + index +
'}';
}
public Student changeIndexNumber(String indexNumber){
return new Student("Paweł",39,indexNumber);
}
}
```

Interfejs Predicate i metoda test(T)

1. Pomaga przy sekwencyjnym przetwarzaniu danych.
2. Na podstawie obiektu zwraca nam Booleana.

Klasa App

```
public class App {
public static List<Student>createData(){
        List<Student> result=new ArrayList<>(
                Arrays.asList(
new Student("Paweł",38,"123"),
                    new Student("Jacek",34,"345"),
                    new Student("Kasia",26,"341"),
                    new Student("Tomasz",39,"3145")
                )
        );
        return result;
}
}
```

```
public static void main(String[] args) {
    List<Student> students=createData();
Predicate<Student> kasiaPredicate=new Predicate<Student>() {
@Override
public boolean test(Student student) {
return student.getName().equals("Kasia");
}
    };
System.out.println(students);
}
```

W metodzie main tworzymy listę studentów i predykat kasiaPredicate, który pomoże nam wyszukać studentów o imieniu Kasia. Ten zapis można zastąpić następującym wyrażeniem lambda

```
Predicate<Student> kasiaPredicate=student ->
student.getName().equals("Kasia");
```

A więc będzie

```
public static void main(String[] args) {
    List<Student> students=createData();
Predicate<Student> kasiaPredicate=student ->
student.getName().equals("Kasia");
System.out.println(students);
}
```

Napiszemy metodę, która dla listy studentów do niej przekazanej zwraca listę studentów spełniających test zawarty w predykacie.

```
public static List<Student>getStudents(List<Student>
list,Predicate<Student> predicate){
    List<Student> result=new ArrayList<>();
    for (Student student:list
         ) {
if (predicate.test(student))result.add(student);
}
return result;
}
```

Obiekt spełnia test predykatu jeśli zostanie zaakceptowany przez metodę test(T)

Teraz wystarczy uruchomić tę metodę, aby otrzymać listę studentów, mających na imię Kasia.

```
System.out.println(getStudents(students,kasiaPredicate));
```

Jakie operacje można wykonywać na predykatach?

And, Or, negate

```
Predicate<Student> equals38andKasia=kasiaPredicate.and(equals38Predicate);
System.out.println(getStudents(students,equals38andKasia));
```

Wybierze nam studentów, którzy spełniają jednocześnie dwa warunki

Interfejs Consumer i metoda accept()

Na podstawie obiektu wykonuje jakąś operację dzięki metodzie accept() ale nic nie zwraca. Np. coś wyświetla.

Napiszemy teraz consumera, który na podstawie obiektu Student nic nie zwraca, ale wyświetla imię studenta.

```
Consumer<Student> studentNameConsumer=new Consumer<Student>() {
@Override
public void accept(Student student) {
        System.out.println(student.getName());
```

```
}
};
```

Można uprościć do wyrażenia lambda

```
Consumer<Student> print= student -> System.out.println(student.getName());
```

I wyświetlić (czyli skonsumować studentów)

```
public static void
consumeStudents(List<Student>list,Consumer<Student>consumer){
for (Student student:list
        ) {
        consumer.accept(student);
}
}
```

```
consumeStudents(students,print);
```

A jak zrobić skonsumowanie (wyświetlenie studentów po przefiltrowaniu?)

```
consumeStudents(getStudents(students,kasiaPredicate),print);
```

i wszystko jasne.

Jak się łaczy consumerów? Metodą andThen(C)

```
Consumer<Student> printName= student ->
System.out.println(student.getName());
Consumer<Student> printAge=student -> System.out.println(student.getAge());
Consumer<Student> two=printName.andThen(printAge);
consumeStudents(getStudents(students,kasiaPredicate),two);
```

Należy pamiętać, że jeśli nie powiedzie się wykonywanie pierwszego consumera, to i drugi będzie zatrzymany.

Interfejs Supplier

Nie bierze żadnych argumentów, a zwraca obiekt czyli odwrotnie jak w przypadku Consumera.

Napiszemy teraz suppliera, aby zwracał nam całą listę studentów.

```
Supplier<List<Student>> supplyStudentList=()->createData();
```

Albo jako zapis funkcyjny

```
Supplier<List<Student>> supplyStudentList= App::createData;
```

Teraz użyjemy suppliera razem z consumerem i predicate.

```
consumeStudents(getStudents(supplyStudentList,kasiaPredicate),two);
```

oczywiście aby to zadziałało należy trochę zmienić metodę getStudents()

```
public static List<Student>getStudents(Supplier<List<Student>>
supplier,Predicate<Student> predicate){
    List<Student> result=new ArrayList<>();
List<Student> students=supplier.get();
    for (Student student:students
         ) {
if (predicate.test(student))result.add(student);
}
return result;
}
```

Interfejs Function i metoda apply()

Bierze jeden typ obiektu, modyfikuje i zwraca inny obiekt.

Zrobimy teraz Function, które będzie brało studenta i zwracało jego imię.

```
Function<Student,String> functionStudentName=student -> student.getName();
```

Albo prościej

```
Function<Student,String> functionStudentName= Student::getName;
```

Dodamy teraz funkcję do trzech pozostałych interfejsów.

Wcześniej musimy jednak zmienić consumera

```
Consumer<String> printString= string-> System.out.println(string);
```

```
consumeStudents(getStudents(supplyStudentList,kasiaPredicate),functionStude
ntName,printString);
```

```
public static void
consumeStudents(List<Student>list,Function<Student,String>
function,Consumer<String>consumer){
for (Student student:list
         ) {
        consumer.accept(function.apply(student));
}
}
```

i teraz będzie działać

Interfejs BiFunction

Bierze dwa obiekty, a zwraca trzy obiekty.

Interfejs BinaryOperator

Na podstawie dwóch obiektów tworzy jeden.

Warianty prymitywne interfejsów funkcyjnych

IntPredicate, na podstawie int zwraca Booleana

DoublePredicate, LongPredicate itd.

Podobnie jest z Consumerami i Supplier.

Jest też np. Klasa ToIntFunction = pobiera jakiś obiekt i zwraca inta. Są odpowiedniki dla innych typów prymitywnych.

Method references

To zapis np.:

```
Function<Student,String> functionStudentName= Student::getName;
```

Gdzie App jest naszą klasą.

Można więc powiedzieć, że jest to skrócenie zapisu lambdy.

Optional i metoda get()

Opakowuje inny obiekt i mówi o nim, że może być lub nie być.

Chodzi o to, by uniknąć zwracania nulli.

1 sposób, gdy nie jesteśmy pewni, czy nie będzie nullem

```
public Optional<Indeks>getIndex() {
return Optional.ofNullable(index);
}
```

2 sposób, gdy jesteśmy pewni, że nie będzie nullem

```
public Optional<Indeks>getIndex() {
return Optional.of(index);
}
```

3 sposób, gdy zwracamy pustą wartość

```
public Optional<Indeks>getIndex() {
return Optional.empty();
}
```

Jak korzystać z Optional?

```
Optional<Indeks> indeks=supplyStudentList.get().get(0).getIndex();
```

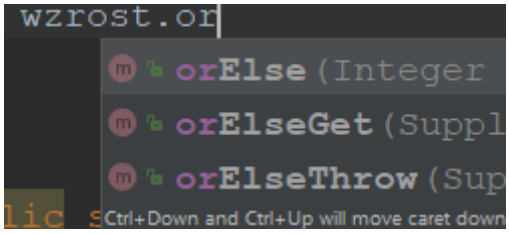
1 sposób

```
if(indeks.isPresent()){
System.out.println(indeks.get().getIndexNumber());
}
```

jeśli wywołamy geta na pustym Optional dostaniemy wyjątek

2 sposób (tutaj będzie consumer)

```
indeks.ifPresent(i ->System.out.println(i));
```



3 sposób (filtrowanie)

```
indeks.filter(i->i.getIndexNumber().equals("123")).ifPresent(i-
>System.out.println(i.getIndexNumber()));
```

4 sposób (mapowanie)

```
indeks.map(i->i.getIndexNumber()).filter(number-
>number.equals("123")).ifPresent(index-> System.out.println(index));
```

Stream API

To zestaw metod do strumieniowego przetwarzania danych.

Po streamie możemy przejechać się tylko raz czyli inaczej niż w kolekcjach.

Przykład

```
supplyStudentList.get().stream().filter(kasiaPredicate).map(Student::getNam
e).forEach(printString);
```

Generowanie wartości dla strumieni

1 sposób (predefiniowane wartości)

```
Stream.of("A","B","C").forEach(printString);
```

2 sposób (kolekcje)

```
List<Student> studentList1=createData();
studentList1.stream().filter(kasiaPredicate).forEach(printName);
```

3 sposób (kolekcje)

```
Stream.generate(supplyStudentList).forEach(print);
```

Tylko, ze trzeba napisać jeszcze consumera dla listy studentów

4 sposób (Supplier)

```
Stream.generate(()->Math.random()).limit(10).forEach(System.out::println);
```

5 sposób (iterate, wyświetlenie 20 liczb parzystych, począwszy od 0)

```
Stream.iterate(0,i->i+2).limit(20).forEach(System.out::println);
```

6 sposób (liczby podzielne przez 2)

```
IntStream.rangeClosed(5,100).filter(i-
>i%2==0).forEach(System.out::println);
```

Filtr

Filter to operacja pośrednia na strumieniach. Filter przyjmuje jako argument Predicate.

```
public staticStream<Student>createDataStream(){
    Student student1=newStudent("Paweł",38,"123");
Student student2=newStudent("Jacek",34,"345");
Student student3=newStudent("Kasia",38,"341");
Student student4=newStudent("Tomasz",39,"3145");
    return Stream.of(student1,student2,student3,student4);
}
```

oraz metoda main()

```
public staticvoidmain(String[] args) {
Predicate<Student> over30Predicate=student ->student.getAge()>30;
Consumer<String>println= System.out::println;
Function<Student,String>getStudentName=Student::getName;

createDataStream().filter(over30Predicate).map(getStudentName).forEach(prin
tln);
}
```

Wyświetlamy więc imiona studentów powyżej 30 lat.

Map

Map to operacja pośrednia na strumieniach. Map przyjmuje jako argument Function.

Map i Filter można stosować wiele razy i na przemian.

ForEach

ForEach to metoda terminalna, która kończy używanie strumienia.

FindFirst, AnyMatch,AllMatch,NoneMatch

Wszystkie trzy to metody terminalne, które po wykonaniu swoich zadań zamykają strumień.

Wyświetlimy imię pierwszego studenta, którego wiek przekracza 30 lat

```
createDataStream().filter(over30Predicate).map(getStudentName).findFirst().
ifPresent(System.out::println);
```

AnyMatch– czy dowolny obiekt w strumieniu spełnia Predicate

Czy jest jakiś student o imieniu Paweł?

```
System.out.println(createDataStream().map(getStudentName).anyMatch(s -
>s.equals("Paweł")));
```

AllMatch – czy wszystkie obiekty spełniają określony warunek

Czy wszystkie imiona są palindromami

```
System.out.println(createDataStream().map(getStudentName).allMatch(s->new
StringBuilder(s).reverse().equals(s)));
```

Czy wszystkie wyrazy znajdujące się w pliku są palindromami?

```
Files.readAllLines(Paths.get("palindrom.txt")).stream().map(String::toLower
Case).allMatch(s->new StringBuilder(s).reverse().equals(s));
```

Reduce

Metoda terminalna. Redukuje strumień do jednej wartości.

Można użyć do wyszukiwania min,max czy do łączenia stringów.

Sumowanie dziesięciu liczb losowych (użycie jako drugiego parametru newBinaryOperator i przekształcić na lambdę)

```
System.out.println(Stream.generate(Math::random).limit(10).reduce(0.0,
(aDouble, aDouble2) -> aDouble+aDouble2));
```

Jak znaleźć najstarszego studenta?

```
createDataStream().map(s-
>s.getAge()).max(Comparator.naturalOrder()).ifPresent(System.out::println);
```

lub

```
createDataStream().map(Student::getAge).reduce(Integer::max).ifPresent(Syst
em.out::println);
```

Collect

Metoda terminalna, jest to specjalny typ reduce, który pozwala nam np. na zebranie wszystkich elementów w listę.

Jak uzyskać listę wieku studentów?

```
System.out.println(createDataStream().map(Student::getAge).collect(Collecto
rs.toList()));
```

Counting – zwraca liczbę elementów.

Jak połączyć wszystkie elementy ze strumienia do jednego stringa z separatorem „ , ” ?

```
createDataStream().map(Student::getAge).map(s-
>s.toString()).collect(Collectors.joining(","))
```

Jak utworzyć mapę, klucz = age, wartość = liczba studentów w danym wieku

```
Map<Integer,List<Student>>
list=createDataStream().collect(Collectors.groupingBy(Student::getAge));
System.out.println(list);
```

Limit, skip, distinct,sorted, count

Limit – ograniczenie do liczby elementów

Skip – pomija określoną liczbę elementów

Distnict – bierzemy pod uwagę tylko różne obiekty (hashcode i equals)

Sorted() – sortowanie wg naturalnego porządku lub newComparator w nawiasie

Count – oblicza ilość elementów w strumieniu (reduktor)

Strumienie typów prymitywnych

Zaleta?

Szybciej pracuje się na strumieniach prymitywnych

```
IntStreamintStream=createDataStream().map(Student::getAge).mapToInt(value ->value.intValue());
```

Może być jeszcze tylko Long i Double

Wada?

Od teraz wszędzie w parametrach trzeba podawać Integerowe odmiany interfejsów funkcyjnych.