

ROZDZIAŁ 1

Elementy języka

Spis treści

.....	1
cz. 1 - typy danych i zmienne	6
Typy danych	6
Typy proste	6
Typy złożone	7
Tablice (arrays)	8
Pętla po tablicy	9
Zmiana typu (casting)	9
Typ "var"	9
Enum	10
cz. 2 - klasy i obiekty	11
Tworzenie klasy	11
Konstruktor	12
Tworzenie obiektu	12
Akcesory (getter i setter)	12
Słowo „this” (tylko w kontekście pól)	13
Używanie metod i pól obiektu	13
Smutna prawda	14
cz. 3 - modyfikatory dostępu	15
Rodzaje dostępu	15
Omówienie dostępu elementów klasy	15
cz. 4 - inne modyfikatory	17
final - elementy stałe	17
Stale zmienne	17
Stale pola klasy	17
Stale metody	18
Stale klasy	18
static - elementy statyczne	19
Statyczne pola i metody klasy	19
Połączenie static i final	20
Inne modyfikatory	20
cz. 5 - przeładowywanie metod i konstruktor kopiujący	21
Przeładowywanie (overloading)	21
Konstruktor kopiujący	22
cz. 6.1 - techniki w klasach - value object i rekordy	23
Value objects	23
Przykładowy value object	23

Modyfikacja obiektu	24
Rekordy (record classes)	25
Przykładowy rekord	25
cz. 6.2 - techniki w klasach - cache'owanie	27
Cache'owanie	27
cz. 6.3 - techniki w klasach - singletony	29
Singleton	29
Przykładowy klasa typu singleton	29
cz. 7 – dziedziczenie.....	31
Przykład – klasy Employee i Client.....	31
Sposób 1 - protected.....	33
Sposób 2 - konstruktor rodzica.....	34
Nadpisywanie (override)	35
Modyfikator "final"	36
Blokujemy możliwość dziedziczenia klasy	36
Blokujemy możliwość nadpisywania metody	36
cz. 8 - abstrakcja	37
Modyfikator "abstract"	37
Klasa abstrakcyjna	37
Metoda abstrakcyjna	38
cz. 9 - interfejsy	39
Przykładowy interfejs	39
Metoda domyślna	40
cz. 10 - klasa w klasie	42
Przypadek 1 - zwykła klasa w zwykłej klasie	42
Przypadek 2 - zagnieżdżona klasa statyczna	43
cz. 11 - typ generyczny.....	45
Typy generyczne.....	45
Klasa generyczna	45
Metoda generyczna	46
Kilka typów generycznych na raz	47
cz. 12 - wyjątki, zarządzanie błędami.....	48
Try-catch.....	48
Throws	51
Wyrzucanie błędów.....	51
Tworzenie własnego rodzaju wyjątków	52
cz. 14.1 - techniki w klasach 2 - buildery	53
cz 14.2 - techniki w klasach 2 - dekoratory	58
cz. 15 - JDK i importowanie.....	62

JDK.....	62
Importowanie	63
cz. 16 - listy, mapy, sety	64
Interfejs "List"	64
ArrayList	64
Tworzenie listy	64
Przykładowe operacje na liście	65
LinkedList.....	65
Interfejs "Map"	66
HashMap.....	66
Interfejs "Set"	67
HashSet	67
cz. 17 - string i StringBuilder	68
String.....	68
Porównywanie napisów	68
Podstawowe metody Stringa	69
Wyrażenia regularne	69
Formatowanie napisu	70
StringBuilder	70
cz. 18.1 - pliki - File	72
Tworzenie obiektu File	72
Ważne metody w klasie File	72
cz. 18.2 - pliki - FileWriter i FileReader	73
FileWriter	73
FileReader.....	74
cz. 18.3 - pliki - BufferedWriter i BufferedReader	75
BufferedWriter.....	75
BufferedReader	76
cz. 18.4 - pliki - Scanner.....	78
Scanner.....	78
Najważniejsze metody w Scannerze	79
cz. 18.5 - pliki - FileOutputStream i FileInputStream.....	80
FileOutputStream	80
FileInputStream	81
cz. 18.6 - pliki - DataOutputStream i DataInputStream	82
DataOutputStream	82
DataInputStream	83
cz. 18.7 - pliki - ObjectOutputStream i ObjectInputStream	84
ObjectOutputStream	84

ObjectInputStream	85
cz. 18.8 - Files i Paths	86
Paths i Files	86
cz. 19 - Random	88
Najważniejsze metody	88
cz. 20 - data i czas	89
Aktualny czas i godzina	89
Formatowanie daty i czasu	89
Najważniejsze symbole do formatu czasu i daty	90
cz. 21 - lambda; interfejs funkcyjny	91
Interfejs funkcyjny	91
Przypisywanie istniejących metod do interfejsu funkcyjnego	91
Przekazywanie funkcji do metody	92
Przekazywanie istniejącej metody do funkcji	93
Typy interfejsów funkcyjnych	93
Function<argument, return>	93
Consumer<argument>	93
Predicate<argument>	94
Supplier<return>	94
Comparator	94
cz. 22 - strumienie	95
Strumienie	95
cz. 23 - sortowanie	97
Komparator	97
Sortowanie tablicy	97
Sortowanie strumieni	98

cz. 1 - typy danych i zmienne

Typy danych

Java ma dwa główne rodzaje typów danych:

- Proste
- Złożone

Typy proste

Typ	Rozmiar (bajty)	Opis
byte	1	Bardzo małe liczby naturalne (-128, 127).
char	2	Jeden znak (np. 'A').
short	2	Liczby naturalne od -32768 do 32767.
int	4	Liczby naturalne (duży zakres).
long	8	Zajebiście duże liczby naturalne.
float	4	Liczby z przecinkiem. Używamy tego typu tylko jak potrzebujemy szybkości działania. Liczby floatowe muszą być zapisywane z 'f', np. 21.37f .
double	8	Liczby z przecinkiem. Trochę wolniejsze niż float, ale dużo bardziej dokładne. Standardowo powinniśmy z tego korzystać. Możemy pisać liczby z 'd', np. 21.37d, ale nie jest to wymagane, bo jest to dodawane domyślnie.
boolean	1	Wartość prawda-fałsz. Java nie traktuje tego jako 0 i 1, tylko jako całkowicie osobny typ danych: false , true . Dlatego normalnie nie możemy ustawić booleana na np. 1.

```
int zmienna = 12;
```

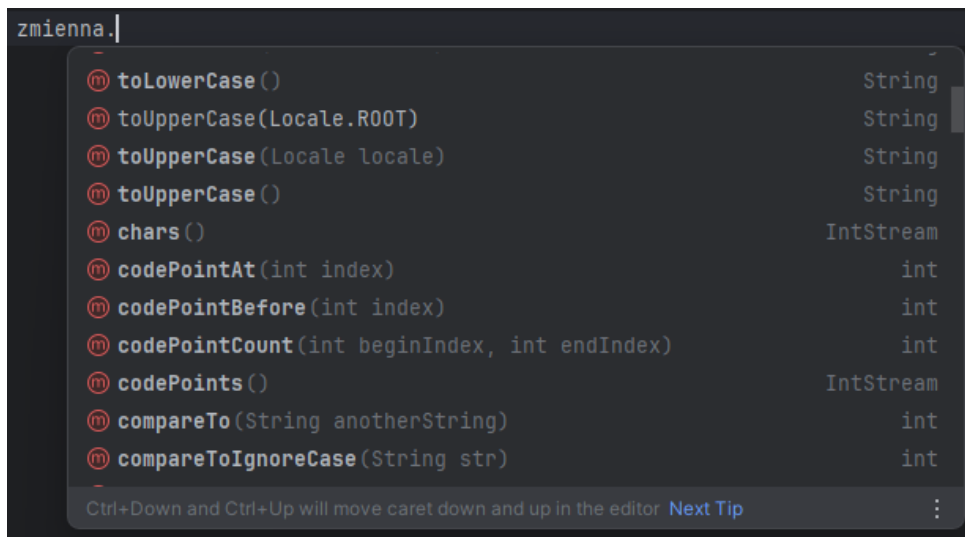
Typy złożone

Typ złożony to tak naprawdę klasa, która dodatkowo zawiera metody (czyli funkcje) i inne duperele. Typ złożony zwykle można odróżnić tym, że zaczyna się od wielkiej litery.

Typów złożonych jest dużo i będą omawiane później po kolei, ale jednym z podstawowych jest **String**:

```
String napis = "napis";
```

Poza tym, że String reprezentuje ciąg znaków, to posiada on różne dodatkowe metody, np.:



Możemy ustawić zmienną o typie prostym nie podając wartości, wtedy przyjmie ona wartość domyślną. Dla np. **inta** jest to 0:

```
int zmienna; // 0
```

Pracując na zmiennych o typie prostym pracujemy tak naprawdę na kopiach:

```
int zmienna = 5;
int zmienna2 = zmienna;
zmienna2 = 10;
zmienna1 // dalej 5
```

Typy złożone (np. obiekty) są robione tylko dynamicznie, ale nie musimy się tym przejmować bo Java odwala całą robotę za nas (w przeciwieństwie do skurwysyna C++) i sama je usuwa.

Tablice (arrays)

```
int[] tablica = new int[5]; // rozmiar 5
```

Tablice są typem złożonym, czyli są dynamiczne. To oznacza, że tworzymy je przy użyciu słowa "**new**". Potem podajemy ich typ i rozmiar, w naszym wypadku 5.

Nie musimy później niczego usuwać, bo Java jest językiem ludzi pracujących, którzy nie mają czasu na takie bzdety jak zarządzanie pamięcią.

Aby odnieść się do elementu tablicy wystarczy użyć indeksu:

```
tablica[0] = 5;  
tablica[1] = 3;  
// ...
```

Oczywiście możemy też najpierw napisać samą zmienną, a tablicę utworzyć kilka linii później:

```
int[] tablica;  
// ...  
tablica = new int[5];
```

Możemy też od razu podać elementy tablicy:

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

Aby utworzyć tablicę wielowymiarową wystarczy dodać kolejne nawiasy kwadratowe:

```
int[][] tablica = new int[5][4];
```


Pętla po tablicy

Oczywiście moglibyśmy użyć zwykłego fora i lecieć indeksami: `tablica[0]`, `tablica[1]`, ...; ale Java pozwala na iterowanie od razu po elementach tablicy bez zbędnego pierdolenia:

```
int[] tablica = {1, 2, 3, 4, 5};
for (int element : tablica) {
    // ...
}
```

Nie musimy pisać żadnych indeksów - od razu mamy dostęp do każdego elementu w tablicy.

Zmiana typu (casting)

Aby zmienić jeden typ na inny (mądrze: rzutowanie, casting) wystarczy napisać coś takiego:

```
int liczba1 = 12;
long liczba2 = (int) liczba1;
```

W przypadku liczb jest o tyle ciekawie, że Java domyślnie konwertuje mniejsze typy na większe, czyli tak naprawdę możemy nawet napisać tak:

```
int liczba1 = 12;
long liczba2 = liczba1;
```

I typ skonwertuje się sam.

Typ "var"

Java pozwala na automatyczne ustawienie typu zmiennej. Na przykład - zamiast pisania:

```
String text = "Tekst";
```

Możemy napisać:

```
var text = "Tekst";
```

Java sama domyśli się jaki to jest typ. Jeżeli nie będzie mogła się tego domyślić, to kod nie zadziała.

Jest to tylko i wyłącznie pomoc dla nas - podczas pisania kodu. Niektórzy wolą tego nie używać, bo ciężiej się połapać jakiego typu jest zmienna. Jeżeli nie wiesz co robisz, to lepiej daj sobie z tym spokój.

Enum

Enum to specjalny typ złożony, który reprezentuje **grupę stałych wartości**. Na przykład: chcemy zapisać w zmiennej jeden z trzech kolorów - Red, Green, Blue. Moglibyśmy to zrobić w stringu:

```
String color;
```

Ale męczące byłoby sprawdzanie, czy na pewno jest tam wpisany dobry kolor, a potem sprawdzanie tego po kilka razy.

Lepiej jest zastosować **Enuma**, w którym po prostu podamy te kolory:

```
enum Color { RED, GREEN, BLUE };
```

Z Enuma korzystamy w następujący sposób:

```
Color kolor = Color.RED;
```

```
if (kolor == Color.BLUE) {  
    // ...  
}
```

cz. 2 - klasy i obiekty

Poniżej znajduje się przykładowa klasa z różnymi elementami, które zostaną po kolei omówione. Często dla ułatwienia przed kodem będzie napisane w jakim pliku się on znajduje.

Example.java

```
class Example {
    int number_1, number_2; // pola (zmienne)

    Example(int number_1, number_2) { // konstruktor
        this.number_1 = number_1;
        this.number_2 = number_2;
    }

    // akcesory (getter i setter)

    int getNumber_1() { // metoda, getter
        return number_1;
    }

    void setNumber_1(int number_1) { // metoda, setter
        this.number_1 = number_1;
    }
}
```

Tworzenie klasy

Każda klasa (publiczna, zewnętrzna) musi znajdować się w nowym pliku, o tej samej nazwie, co nazwa klasy. W naszym przypadku cała klasa **Example** znajduje się w pliku **Example.java**.

Nazwa klasy powinna zaczynać się z wielkiej litery i być pisana tzw. snake case, czyli: PrzykładowaNazwaKlasy.

Klasa może zawierać **pola** i **metody**. Pola to po prostu zmienne/stałe, które znajdują się w danej klasie, a metody to funkcje, które się w tej klasie znajdują.

Klasa to "przepis", szablon na coś. Ona tylko mówi, jak coś (obiekt) powinno zostać zrobione / zbudowane, ale sama w sobie nic nie robi (o ile nie jest statyczna, ale na razie chuj nas to obchodzi). **Obiekt** (instancja klasy) to element zbudowany na podstawie klasy. Zawiera pola i metody, które ustaliliśmy w klasie.

Dlatego zwykle nie ustawiamy wartości pól w samej klasie, tylko robimy to w konstruktorze, który ustawia je dla obiektu. Jedynym wyjątkiem są stałe statyczne, np. liczba Pi w klasie Math (ale to też chuj nas póki co obchodzi).

Konstruktor

Każda klasa posiada **konstruktor**, który tworzy się następująco:

```
NazwaKlasy() {  
    // zawartość  
}
```

Konstruktor nie ma zwracanego typu.

Jak nazwa wskazuje, konstruktor coś konstruuje. Tym czymś jest **obiekt**.

W konstruktorze najczęściej ustawiamy wartości zmiennych dla danego obiektu.

Każdy obiekt jest oddzielny i posiada własne pola i metody, które nie są dzielone z innymi obiektami (o ile nie są statyczne).

Jeżeli nie napiszemy konstruktora, to klasa będzie miała domyślny konstruktor, który nic nie robi.

Tworzenie obiektu

Aby utworzyć obiekt z naszej klasy możemy napisać:

```
Example nazwaObiektu = new Example(21, 37);
```

Co tu się kurwa dzieje?

Pierwsza część (przed znakiem =) to najzwyklejsza zmienna. Piszemy jej typ i nazwę. Naszym typem jest nasza klasa (bo to przecież wzór, schemat).

W drugiej części wywołujemy konstruktor z dwoma argumentami – czyli dwoma liczbami: 21, 37. Ten konstruktor robi to, co ustaliliśmy w klasie - czyli bierze te dwie liczby i ustawia pola **number_1** i **number_2** na ich wartości, a potem zwraca nowy obiekt na podstawie klasy Example. Przypisujemy potem ten obiekt do zmiennej **nazwaObiektu**.

Klasę można potraktować jako typ złożony, czyli aby utworzyć jej obiekt musi użyć „**new**”. To też oznacza, że obiekty są dynamiczne.

Akcesory (getter i setter)

Zwykle w naszych klasach będziemy ustawiać wszystkim polom dostęp prywatny (dla ułatwienia teraz jeszcze tego nie robimy). Abyśmy mogli w ogóle tych pól użyć poza klasą, musimy napisać metody, które będą nam je zwracać lub ustawiać.

Getter - to metody, które zwracają pola.

Setter - to metody, które ustawiają pola.

Podczas tworzenia metod musimy podać typ, jaki one zwracają. Może to być typ prosty (jak np. int) ale i typ złożony (jak np. inna klasa). Jeżeli metoda nic nie zwraca to piszemy **void**.

Po co nam to, skoro możemy po prostu odwoływać się bezpośrednio?, np.

```
nazwaObiektu.number_1;
```

Dlatego, że to w chuj niebezpieczne. Nie mamy żadnego zabezpieczenia, np. przed tym, aby nie ustawić tam czegoś pojebanego. W setterze za to możemy oprogramować cały system sprawdzania, czy to, co

podajemy jest poprawne.. Tak samo w getterach - zwracamy tylko kopię zmiennej, więc wszystko jest zabezpieczone.

Słowo „this” (tylko w kontekście pól)

Jeżeli spojrzymy na konstruktor to zauważymy, że pola w naszej klasie nazywają się tak samo jak parametry, które mamy w nagłówku konstruktora (dla niewtajemniczonych: argument to to, co przekazujemy w funkcji, a parametr to zmienna w nagłówku funkcji, która tą wartość otrzymuje; w praktyce to jedno i to samo – „coś” w nagłówku funkcji; ale niektóre osoby mają o to ból dupy). Mamy problem, bo Java nie wie która nazwa odnosi się do pola w klasie, a która do parametru w konstruktorze.

Dlatego możemy napisać tak:

```
this.number_1 = number_1;
```

Wartość po lewej odnosi się do pola **number_1** w naszej klasie, a wartość po prawej do parametru **number_1** w konstruktorze.

Oczywiście moglibyśmy też po prostu nazwać parametr inaczej, np. num_1, wtedy nie musielibyśmy używać "this".

Używanie metod i pól obiektu

Aby użyć zwykłej metody lub pola, musimy utworzyć obiekt danej klasy, a następnie podać nazwę obiektu, kropkę i nazwę pola lub metody, np.:

```
nazwaObiektu.setNumber_1(15);
```

Należy pamiętać, że to zadziała tylko wtedy, gdy mamy do nich dostęp, ale o tym za chwilę.

Main.java

```
public class Main {
    public static void main(String[] args) {
        Example nazwaObiektu = new Example(21, 37);
        nazwaObiektu.number_1; // 21
        nazwaObiektu.number_2; // 37

        int num = nazwaObiektu.getNumber_1(); // 21
        nazwaObiektu.setNumber_1(15);
        num = nazwaObiektu.getNumber_1(); // 15

        Example drugiObiekt = new Example(12, 34);
        int num2 = drugiObiekt.getNumber_1(); // 12

        // pierwszy obiekt nie ma wpływu na drugi, ich zmienne i
        metody są oddzielne
    }
}
```

Każdy program Javy wykonuje się od metody **main()** w klasie **Main**. To tam musimy zacząć pracę z utworzonymi przez nas klasami.

Smutna prawda

Tak naprawdę w prawdziwych programach klasy najczęściej nie przedstawiają żadnych obiektów z prawdziwego życia. To nie są żadne kotki, pieski i inne baśniowe stwory.

W rzeczywistości klasa **grupuje powiazaną ze sobą logikę**. Np. cały kod odpowiedzialny za system logowania możemy wrzucić do klasy **Login**.

Dla prostoty to zignorujemy bo nie ma chuja, że czegoś byśmy się nauczyli, gdybyśmy tego przestrzegali.



cz. 3 - modyfikatory dostępu

Klasa z części drugiej jest tak naprawdę chujowa, bo nie ustaliliśmy do żadnego jej elementu dostępu.

Rodzaje dostępu

- **public** - do takiego elementu można się odwoływać z każdego miejsca w kodzie.
- **private** - do takiego elementu można odwoływać się tylko z klasy (lub z obiektu z klasy), w której on powstał.
- **protected** - do takiego elementu można odwoływać się z tej samej klasy i z klas, które dziedziczą tę klasę (o tym później).
- **[domyślny]** - do elementu bez podanego modyfikatora można odwołać się tylko z tego samego pakietu (pakiety to coś a'la foldery, które grupują klasy; póki co nie jest to zbyt ważne). Chujowy rodzaj dostępu, lepiej go nie używać.

Skoro to już wiemy, to poprawmy naszą klasę z części 2.:

```
public class Example {  
    private int number_1, number_2;  
  
    public Example(int number_1, number_2) {  
        this.number_1 = number_1;  
        this.number_2 = number_2;  
    }  
  
    public int getNumber_1() {  
        return number_1;  
    }  
  
    public void setNumber_1(int number_1) {  
        this.number_1 = number_1;  
    }  
}
```

Omówienie dostępu elementów klasy

Klasy, po których nazwane są pliki MUSZĄ być publiczne, bo inaczej Java nie będzie miała do niej dostępu. A po co nam klasa, do której nie mamy dostępu?

Dostęp do **pól** w klasie zależy od nas, ale najlepiej, aby były prywatne - to najlepsza praktyka; o ile nie robimy czegoś, co wymaga pola publicznego.

Konstruktor powinien być publiczny, ponieważ w innym wypadku nie moglibyśmy stworzyć obiektu danej klasy poza nią. Konstruktor może być prywatny, ale to skomplikowany temat - prywatny konstruktor oznacza, że obiekt można stworzyć tylko w tej samej klasie, np. w innej metodzie, albo w ogóle go nie można stworzyć.

Gettery i settery powinny być publiczne, bo takie jest ich zadanie - mają nam "upubliczniać" pole.

Dostęp **innych metod** zależy od naszych potrzeb. Niektóre mogą być publiczne, niektóre prywatne.

cz. 4 - inne modyfikatory

final - elementy stałe

Możemy użyć modyfikatora **final**, aby oznaczyć element jako stały.

Stałe zmienne

```
final int zmienna = 5;
```

Ustawiając zmienną na stałą nie możemy jej zmienić w żaden sposób.

Uwaga! Jeżeli ustawimy zmienną, która przechowuje obiekt jako stałą, to Java tylko sprawdza, czy referencja do obiektu się nie zmienia - to oznacza, że sam obiekt może ulec zmianie.

```
final int tablica = {1, 2, 3};  
tablica[3] = 4; // git  
tablica = {4, 5, 6}; // nie git
```

Stałe pola klasy

Jeżeli ustawimy pole klasy jako stałą to MUSIMY ustawić mu wartość w konstruktorze albo od razu w klasie. Potem nie możemy tego pola zmienić w żaden sposób.

```
public class Example {  
    public final int number;  
  
    public Example(int number) {  
        this.number = number;  
    }  
}
```

Stałe metody

(Najlepiej wróć tu po części omawiającej dziedziczenie)

Jeżeli ustawimy metodę jako stałą, to nie będziemy mogli jej nadpisać w klasie-dziecku.

Example.java

```
public class Example {  
    public final int metoda() {  
        return 2;  
    }  
}
```

ExampleChild.java

```
public class ExampleChild extends Example {  
    @Override  
    public int metoda() { // błąd  
        return 4;  
    }  
}
```

Stałe klasy

(Najlepiej wróć tu po części omawiającej dziedziczenie)

Jeżeli ustawimy klasę jako stałą, to nie będziemy mogli używać jej do rozszerzania innych klas.

Example.java

```
public final class Example {  
  
}
```

ExampleChild.java

```
public class ExampleChild extends Example { // błąd  
  
}
```

static - elementy statyczne

Możemy użyć modyfikatora **static**, aby oznaczyć element jako statyczny - czyli taki niewymagający utworzenia obiektu, aby go używać.

Statyczne pola i metody klasy

Po ustawieniu pola lub metody jako statyczną, możemy się do niej odnieść nie tworząc obiektu. Po prostu podajemy nazwę klasy i nazwę pola/metody.

Przydaje się to wtedy, gdy chcemy mieć zapisane jakieś dane, które nie są powiązane z żadnym obiektem (np. kiedy nic z tym obiektem nie robią) lub mają być wspólne dla każdego obiektu (np. liczenie ile jest utworzonych obiektów w danej klasie).

Warto też wiedzieć, że w statycznych metodach nie mamy dostępu do nie-statycznych elementów. To całkiem logiczne, bo nie wiadomo by było z której wartości korzystać. W końcu tworząc kilka obiektów istniałoby w pamięci kilka pól o tych samych nazwach.

Example.java

```
public class Example {  
    public static int number = 5;  
  
    public static void metoda() {  
  
    }  
}
```

Main.java

```
int liczba = Example.number;  
Example.metoda();
```

Jak widać, aby używać metod / pól statycznych nie musimy tworzyć obiektu. Podajemy tylko nazwę klasy i nazwę elementu.

Static – powiązany z klasą.

Nie-static – powiązany z obiektem.

Połączenie static i final

Możemy **połączyć static i final**, wtedy wyjdą nam stałe pola, które nie są powiązane z żadnym obiektem. Przydaje nam się to, kiedy chcemy zrobić "prawdziwą" stałą, do której mamy dostęp cały czas - bez potrzeby zabawy w tworzenie obiektów i inne pierdoły.

Jaka jest kolejność przy pisaniu? Obojętnie, **static final** oraz **final static** działają tak samo, ale dużo osób się sra, że ładniej wygląda to pierwsze.

Example.java

```
public class Example {  
    public static final String VERSION = "1.0";  
}
```

Main.java

```
String wersja = Example.VERSION;  
Example.VERSION = "dupa"; // błąd
```

Inne modyfikatory

Istnieje jeszcze modyfikator "**abstract**" wykorzystywany przy dziedziczeniu, które zostanie opisane później.

cz. 5 - przeładowywanie metod i konstruktor kopiujący

Przeładowywanie (overloading)

Wyobraźmy sobie, że nie ufamy kalkulatorom, bo nie są open-source. Oczywiście nikt nie jest tak pojębany, ale założmy, że tak jest. Chcemy napisać własny, fantastyczny, kalkulator. No to robimy klasę:

Calculator.java

```
public class Calculator {  
    public static int add(int a, int b) {  
        return a + b;  
    }  
}
```

Main.java

```
int result = Calculator.add(2, 5);
```

Dlaczego używamy tu static? Bo nie musimy tworzyć obiektu klasy Calculator. Nie przechowujemy tu żadnych danych, a dodatkowo dzięki temu dużo łatwiej jest użyć metody add(), bo możemy to zrobić od razu - bez tworzenia obiektu.

Ale tutaj mamy pewien problem, bo chcemy też dodawać liczby z przecinkiem, a teraz tego nie możemy zrobić. Metoda wspiera tylko liczby całkowite (inty).

Możemy niby dodać drugą metodę, np. addDouble(), ale jak będziemy chcieli użyć naszego zajebistego kalkulatora to będziemy musieli marnować czas na szukanie metod dobranych do naszego typu danych.

Możemy zamiast tego przeładować metodę add() i pozwolić na użycie jej dla obu typów danych – zarówno inta jak i double’a:

Calculator.java

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

Main.java

```
int result = Calculator.add(2, 5);  
double result2 = Calculator.add(2.1, 3.7);
```

Dzięki przeładowaniu metody, możemy używać intów i double'ów pod tą samą nazwą metody. Przeładowane metody muszą mieć tylko tą samą nazwę - liczba argumentów, ich środek, zwracany typ mogą się różnić.

Możemy też przeładowywać konstruktor.

Konstruktor kopiujący

Brzmi to strasznie, ale tak naprawdę to jest zwykły konstruktor, który różni się tylko tym, że dane pobiera nie z argumentów, a z innego obiektu o tym samym typie. Przykład to wszystko rozjaśni.

Example.java

```
public class Example {  
    private int a, b;  
  
    public Example(int a, int b) {  
        this.a = a;  
        this.b = b;  
    }  
  
    public Example(Example obj) { // konstruktor kopiujący  
        this.a = obj.a;  
        this.b = obj.b;  
    }  
}
```

Main.java

```
Example obj = new Example(21, 37); // konstruktor zwykły  
Example obj_copy = new Example(obj); // konstruktor kopiujący
```

Konstruktor kopiujący to zwykły przeładowany konstruktor, który pobiera dane z obiektu tej samej klasy. Rzadko kiedy nam się on przydaje.

cz. 6.1 - techniki w klasach - value object i rekordy

„Techniki w klasach” to część bardziej teoretyczna, więc jak bardzo ci się śpieszy to ją olej. Poruszane tu rzeczy to praktycznie sama teoria, ale w tym pliku mam wyjebane teorię i ruszę te aspekty od strony bardziej praktycznej, co oznacza, że mogą tu wystąpić pewne duże uproszczenia lub niedociągnięcia.

Jeżeli cię to boli, to:



Value objects

Value objects to tak naprawdę pomocnicze obiekty, które służą do grupowania danych, aby późniejsze korzystanie z nich było dużo prostsze i przyjemniejsze.

Na przykład - kiedy pracujemy z pieniędzmi, mamy do zapisania dwie informacje - ilość i rodzaj waluty. Dodatkowo, raczej nie chcemy mieć w programie ujemnej waluty, więc trzeba się przed tym uchronić.

Po to jest właśnie **value object** - przechowuje te dane i będzie miał odpowiednie zabezpieczenia, dzięki czemu nie będziemy musieli się męczyć.

Zasady tworzenia:

- Tylko dane identyfikują obiekt.
- Po utworzeniu obiektu nie może się on zmieniać.

Przykładowy value object

Money.java

```
public class Money {
    public final double value;
    public final String currencyType;

    public Money(double value, String currencyType) {
        if (value <= 0) {
            // tu wyrzucamy jakiś błąd
        }

        this.value = value;
        this.currencyType = currencyType;
    }
}
```

Main.java

```
Money m = new Money(2.50, "PLN");  
m.value; // 2.50  
m.currencyType; // "PLN"
```

Stworzyliśmy tutaj klasę reprezentującą walutę - mamy wartość i typ. Pola są oznaczone jako **final**, czyli nie możemy ich zmienić po ustawieniu ich wartości w konstruktorze. W konstruktorze sprawdzamy poprawność danych i w razie problemów wyrzucamy błąd (o tym później).

Ciekawą rzeczą jest to, że nasze pola są publiczne. Możemy to zrobić i jest to bezpieczne, bo są one stałe. Gettery/settery tworzymy po to, aby zabezpieczyć obiekt przed niepożądanymi zmianami, ale w naszym przypadku te zmiany nie mogą wystąpić. Oczywiście jeżeli nam się to nie podoba, to możemy zrobić gettery i settery, a pola ustawić jako prywatne - chuja to zmienia.

Modyfikacja obiektu

Ale co jeśli chcielibyśmy dodać w obiekcie możliwość zmiany typu waluty na np. USD? W takim wypadku musielibyśmy zmienić wartości w naszych polach, a teraz tego zrobić nie możemy, bo są one finalne. Problem w tym, że nie możemy ustawić tych pól na nie-finalne, ponieważ wtedy nie będzie to value object. Czyli co, gg, mamy przesrane?

Na szczęście nie. W takim wypadku robimy pewien magiczny zabieg - tworzymy nowy obiekt na podstawie istniejących danych. Dodajemy do klasy metodę **toUSD()**:

Money.java

```
public class Money {  
    public final double value;  
    public final String currencyType;  
  
    public Money(double value, String currencyType) {  
        if (value <= 0) {  
            // tu wyrzucamy jakiś błąd  
        }  
  
        this.value = value;  
        this.currencyType = currencyType;  
    }  
  
    public static Money toUSD(Money money) {  
        return new Money(money.value * 0.25, "USD");  
    }  
}
```


Main.java

```
Money pln = new Money(2.50, "PLN");
Money usd = Money.toUSD(pln); // nowy obiekt na podstawie obiektu pln
```

Pojawia się metoda **toUSD()**, która zwraca obiekt **Money** i jest statyczna. Niektórzy mogą zapytać co tu się odpięrdala.

Po kolei - nie możemy zmienić pól w obiekcie, ale możemy utworzyć nowy obiekt i go zwrócić. To robi ta nowa metoda.

Podajemy w jej parametrze istniejący obiekt i na podstawie jego danych tworzymy nowy, który tym razem będzie reprezentował pieniądze w USD.

Metoda jest statyczna, ponieważ nie musi nic zapisywać. Dane pobiera z obiektu, który jej przesyłamy. Ona sama nie musi mieć własnych pól value i currencyType.

Rekordy (record classes)

To skomplikowany temat, więc upraszczając - rekordy to to samo co value objects, ale z dużo prostszą budową.

Tak naprawdę to rekord jest zupełnie czymś innym, ale nie interesujemy się tym, bo kocięj mordy dostaniemy. Ważne, że wiemy jak działa i co robi.

Przykładowy rekord

Money.java

```
public record Money(double value, String currencyType) {
    public Money {
        if (value <= 0) {
            // tu wyrzucamy jakiś błąd
        }
    }

    public static Money toUSD(Money money) {
        return new Money(money.value * 0.25, "USD");
    }
}
```

Main.java

```
Money pln = new Money(2.50, "PLN");
Money usd = Money.toUSD(pln);

usd.value(); // 2.50
usd.currencyType(); // "USD"
```

Pierwsza różnica - zamiast słowa **"class"** używamy **record**.

Druga różnica - nie mamy pól ani konstruktora. Wszystko robi się automatycznie, kiedy w nawiasach przy nazwie rekordu wypiszemy nasze pola. Jeżeli chcemy dodać jakieś sprawdzajki błędów, to możemy stworzyć pseudokonstruktor (bez nawiasów), który zawierał będzie tylko nasze ify.

Trzecia różnica - rekord sam tworzy gettery w formacie: `nazwaPola()`; nie da się uzyskać danych w inny sposób.

To tyle, reszta jest taka sama. Jak widać, fajna sprawa - robi to samo, a budowa jest znacznie prostsza, szczególnie, kiedy nasze value objecty byłyby bardziej skomplikowane.

cz. 6.2 - techniki w klasach - cache'owanie

Cache'owanie

Wyobraźmy sobie, że mamy metodę, która zwraca nam wynik, który zająłoby nam dużo czasu na obliczenie. Tu mamy problem - nie chcemy za każdym razem tak długo na ten wynik czekać.

Rozwiązanie jest proste - skoro wynik został już obliczony, a dane wejściowe się nie zmieniły, to możemy go zapisać i nie liczyć po raz kolejny. To robi właśnie **cache**.

Money.java

```
public class Money {
    public final Double value;
    public final String currencyType;

    private Double exchangeRate;

    public Money(double value, String currencyType) {
        this.value = value;
        this.currencyType = currencyType;
    }

    public double getUSD() {
        if (exchangeRate == null) {
            exchangeRate = 0.25; // wyobraźmy sobie, że pobieramy to z
                                // internetu i musimy czekać 10 sek.
        }
        return value * exchangeRate;
    }
}
```

Main.java

```
Money pln = new Money(2.50, "PLN");
double usd = pln.getUSD(); // pierwszy raz - czekamy 10 sekund
double usd2 = pln.getUSD(); // kolejny raz - natychmiastowy wynik
```

Mamy tutaj klasę z pieniędzmi (prawie taką samą, jak w przykładzie z value objects). Różnica jest następująca:

- Zamieniliśmy "**double**" na "**Double**" (wielka litera). To pierwsze to typ prosty, a to drugie to obiekt - czyli typ złożony. Oba te typy działają tak samo z tą różnicą, że Double ma dużo fajnych metod (np. `toString()`) i, co najważniejsze, **może być NULlem**. Typ prosty double nie może być równy NULL (bo domyślnie jest równy 0). W metodzie `getUSD()` potrzebujemy double'a z NULlem, dlatego to zrobiliśmy.
- Dodaliśmy metodę `getUSD()`, która zwraca wartość pieniądza po przeliczeniu na USD. Wyobrażamy sobie, że ten przelicznik pobieramy z internetu, a to zwykle trochę trwa.

Cache'em w naszej klasie jest nowe pole **exchangeRate**, a zarządza nim if z metody `toUSD()`:

```
private Double exchangeRate;
```

```
if (exchangeRate == null) { ... }
```

Kiedy pierwszy raz wywołamy tę metodę, **exchangeRate** nie był nigdy pobrany z internetu, więc to robi. Za kolejnym wywołaniem będziemy już mieli zapisaną wartość, więc nie będziemy musieli pobierać go po raz następny.

W prawdziwym świecie nasz cache by się czyścił np. po 30 sekundach, aby dane były aktualne. Na szczęście nie żyjemy w prawdziwym świecie.

cz. 6.3 - techniki w klasach - singletony

Singleton

Singleton to klasa (dokładniej to wzorzec projektowy, według którego robi się klasę), która pozwala na istnienie **tylko jednej instancji (obiektu)**. Nie możemy tworzyć więcej niż jednego obiektu singletonu.

Przydaje się to kiedy mamy klasy pokroju systemu logów. W takim przypadku możemy potrzebować istnienia jakiegoś obiektu, aby zapisać jakieś dane, ale nie potrzebujemy więcej niż jednego obiektu.

Przykładowy klasa typu singleton

```
public class Logger {
    private static Logger instance;
    private List<String> logs = new ArrayList<>();

    public static Logger getInstance() {
        if (instance == null) {
            instance = new Logger();
        }
        return instance;
    }

    public void addLog(String message) {
        logs.add(message);
    }

    public void printLogs() {
        for (String log : logs) {
            System.out.println(log);
        }
    }
}
```

Jedną, jedyną instancję (obiekt) naszej klasy przechowujemy w polu **instance**.

Lista (opisana dużo później, jeżeli nie wiesz co to jest to to zignoruj) przechowuje nasze logi - czyli stringi.

Klasa posiada prywatny (!) konstruktor. To oznacza, że **nie da się stworzyć jej obiektu** w normalny sposób. Nie możemy napisać "new Logger()" w Mainie.

Obiekt tworzymy / pobieramy za pomocą statycznej metody `getInstance()`. Jeżeli instancja nie istnieje to ją tworzymy. Jeżeli istnieje, to ją zwracamy.

Dlaczego ta metoda jest statyczna? Ponieważ musi taka być. Początkowo, żaden obiekt (instancja) klasy `Logger` nie istnieje. Tworzymy go dopiero używając tej metody:

```
Logger instance = Logger.getInstance();
```

Metoda nie-statyczna wymaga istnienia obiektu, z którego się ona wykona, a u nas takiego po prostu by nie było.

cz. 7 – dziedziczenie

Dziedziczenie pozwala nam na uzyskanie dostępu do pól i metod jednej klasy (klasy-rodzica) z innej klasy (klasy-dziecka).

Przydaje nam się to, kiedy chcemy utworzyć jakąś klasę na podstawie innej klasy, wyciągnąć elementy wspólne kilku klas (np. powtarzające się pola) do innej klasy i tym podobne.

Przykład – klasy Employee i Client

Employee.java

```
public class Employee {
    private String firstName, lastName;
    private double height, weight;
    private int age;

    private double salary;
    private String companyName;

    // ...

    public String getFullName() {
        return firstName + " " + lastName;
    }

    public String getCompanyName() {
        return companyName;
    }
}
```

Client.java

```
public class Client {
    private String firstName, lastName;
    private double height, weight;
    private int age;

    private int[] orders;

    // ...

    public String getFullName() {
        return firstName + " " + lastName;
    }

    public int[] getOrders() {
        return orders;
    }
}
```

Widzimy, że dużo elementów się powtarza - imię, wzrost, waga itd. Kiedy mamy dużą ilość takich przypadków może to się zrobić męczące. Dlatego możemy wykorzystać dziedziczenie i elementy wspólne przenieść do klasy-rodzica, z której te dwie klasy będą je dziedziczyć:

Person.java (klasa-rodzic)

```
public class Person {
    private String firstName, lastName;
    private double height, weight;
    private int age;

    // ...

    public String getFullName() {
        return firstName + " " + lastName;
    }
}
```

Employee.java (klasa-dziecko)

```
public class Employee extends Person {
    private double salary;
    private String companyName;

    // ...

    public String getCompanyName() {
        return companyName;
    }
}
```

Client.java (klasa-dziecko)

```
public class Client extends Person {
    private int[] orders;

    // ...

    public int[] getOrders() {
        return orders;
    }
}
```

Wspólne pola i metody przenieśliśmy do klasy Person, a w klasach-kidosach dodaliśmy klauzulę **extends Person**. Dalej możemy korzystać z tych pól i metod, ale nie musimy się już męczyć z pisaniem tego samego kilka razy w różnych klasach.

Main.java

```
Employee emp = new Employee("Jan", "Paweł");
emp.getFullName(); // "Jan Paweł"
```


Żartowałem. W powyższym przypadku wciąż nie mamy dostępu do pól i metod.

Przypomnijmy sobie informacje o wszystkich modyfikatorach dostępu (cz. 3). **private** mówi nam, że: "do takiego elementu można odwoływać się tylko z klasy (lub z obiektu z klasy), w której on powstał".

To oznacza, że tak naprawdę w klasach `Employee` i `Client` **nie mamy dostępu** do pól `firstName`, `lastName` itp.

W takim wypadku jak ustawimy dane w naszym konstruktorze? Chcemy ustawić imię i nazwisko. Są dwa główne rozwiązania:

- Użycie konstruktora rodzica.
- Ustawienie pól na "protected".

Ustawiając pola na **protected** mamy do nich dostęp w klasach-dzieciach. Sprawa załatwiona. Oczywiście jest to trochę niebezpieczne, ale takie jest życie. Używając konstruktora rodzica (funkcja **super()**) dalej nie mamy dostępu do tych pól, ale przynajmniej się ustawią i będziemy mogli użyć metody `getFullName()`.

Sposób 1 - protected

Person.java

```
public class Person {
    protected String firstName, lastName;
    protected double height, weight;
    protected int age;

    // ...

    public String getFullName() {
        return firstName + " " + lastName;
    }
}
```

Employee.java

```
public class Employee extends Person {
    private double salary;
    private String companyName;

    public Employee(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;

        // ...
    }

    // ...

    public String getCompanyName() {
        return companyName;
    }
}
```

Sposób 2 - konstruktor rodzica

Person.java

```
public class Person {
    private String firstName, lastName;
    private double height, weight;
    private int age;

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;

        // ...
    }

    public String getFullName() {
        return firstName + " " + lastName;
    }
}
```

Employee.java

```
public class Employee extends Person {
    private double salary;
    private String companyName;

    public Employee(String firstName, String lastName) {
        super(firstName, lastName);

        // ...
    }

    // ...

    public String getCompanyName() {
        return companyName;
    }
}
```

W konstruktorze klasy „Employee” wywołaliśmy konstruktor klasy „Person” metodą **super()** i podaliśmy argumenty `firstName`, `lastName`.

Nadpisywanie (override)

Chcemy, aby pracownik i klient przy wyświetlaniu imienia mieli dopisek "Pracownik:" lub "Klient:". Klasa `Person` posiada już metodę `getFullName()`, której nie chcemy usuwać. W tym wypadku możemy ją **nadpisać**:

Person.java

(nic się nie zmienia; zostaje kod ze sposobu 1.)

Employee.java

```
public class Employee extends Person {
    private double salary;
    private String companyName;

    public Employee(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    public String getCompanyName() {
        return companyName;
    }

    @Override
    public String getFullName() {
        return "Pracownik: " + firstName + " " + lastName;
    }
}
```

Każda nadpisana metoda musi mieć przed nią dopisek "**@Override**", który oznacza, że jest to metoda nadpisana od rodzica. Nie jest to wymagane, bez tego nadal wszystko zadziała, ale lepiej to pisać bo wtedy środowisko będzie lepiej to przechwytywało i szukało problemów.

Teraz używając `getFullName()` używamy metody z klasy `Employee`, która dopisuje do imienia przedrostek **"Pracownik: "**.

Uwaga! W przypadku, gdy rodzic ma ustawione pola `firstName` i `lastName` na `private`, to nie mamy do nich dostępu nawet w nadpisanej metodzie. Co wtedy? Ano trzeba zrobić coś takiego:

```
@Override
public String getFullName() {
    return "Pracownik: " + super.getFullName();
}
```

`super.getFullName()` oznacza, że odnosimy się do metody `getFullName()` z klasy-rodzica (ta metoda jest publiczna, więc mamy do niej dostęp). Ogólnie, słówko **"super"** oznacza rodzica. Metoda `getFullName()` od klasy-rodzica zwraca nam imię i nazwisko w formie stringa, więc możemy to złączyć z napisem **"Pracownik: "** i wyjdzie nam np. **"Pracownik: " + "Jan Paweł"**.

Modyfikator "final"

Warto sobie przypomnieć jak działał modyfikator "**final**" opisany w cz. 4. Blokuje on możliwość dziedziczenia.

Blokujemy możliwość dziedziczenia klasy

Example.java

```
public final class Example { ... }
```

ExampleChild.java

```
public class ExampleChild extends Example { // błąd  
  
}
```

Blokujemy możliwość nadpisywania metody

Person.java

```
public class Person {  
    protected String firstName, lastName;  
    protected double height, weight;  
    protected int age;  
  
    public final String getFullName() {  
        return firstName + " " + lastName;  
    }  
}
```

Employee.java

```
public class Employee extends Person {  
    private double salary;  
    private String companyName;  
  
    public Employee(String firstName, String lastName) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
  
    @Override  
    public String getFullName() { // błąd, nie można nadpisać  
        return "Pracownik: " + firstName + " " + lastName;  
    }  
}
```

cz. 8 - abstrakcja

Modyfikator "abstract"

Klasa abstrakcyjna

Wróćmy do przykładu z części 7.:

Person.java

```
public class Person {  
    protected String firstName, lastName;  
    protected double height, weight;  
    protected int age;  
  
    public String getFullName() {  
        return firstName + " " + lastName;  
    }  
}
```

Employee.java

```
public class Employee extends Person {  
    private double salary;  
    private String companyName;  
  
    public Employee(String firstName, String lastName) {  
        this.firstName = firstName;  
        this.lastName = lastName;  
    }  
  
    public String getCompanyName() {  
        return companyName;  
    }  
}
```

Mamy pewien problem - okazuje się, że da się stworzyć obiekt klasy Person:

Main.java

```
Person p = new Person();
```

Nawet jak nie podaliśmy tam żadnego konstruktora, to Java stworzyła domyślny - pusty. Wyobraźmy sobie, że bardzo tego nie chcemy. Chcemy, aby dało się tworzyć tylko dzieci, a rodzica nie.

W takim wypadku możemy ustawić klasę "Person" jako **abstrakcyjną**:

```
public abstract class Person {
    protected String firstName, lastName;
    protected double height, weight;
    protected int age;

    public String getFullName() {
        return firstName + " " + lastName;
    }
}
```

Problem rozwiązany. Od teraz nie można utworzyć obiektu klasy Person. Do tej klasy mamy dostęp tylko od strony dzieci. Ustawienie klasy na abstrakcyjną powoduje, że nie możemy stworzyć jej obiektu. Może być ona wykorzystywana tylko do dziedziczenia.

Metoda abstrakcyjna

Ustawiając metodę na abstrakcyjną mówimy dzieciom, że MUSZĄ ją nadpisać (lub nerdowsko: zaimplementować we własnym zakresie). Dodatkowo, u rodzica nie ma ona ciała - posiada tylko sam nagłówek.

Person.java

```
public abstract class Person {
    protected String firstName, lastName;
    protected double height, weight;
    protected int age;

    public abstract String getFullName(); // mówimy dzieciom, że coś
    takiego istnieje i mają to zaimplementować
}
```

Employee.java

```
public class Employee extends Person {
    private double salary;
    private String companyName;

    public Employee(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }

    @Override // a dziecko to implementuje
    public String getFullName() {
        return "Pracownik: " + firstName + " " + lastName;
    }
}
```

cz. 9 - interfejsy

Drugi sposób na abstrakcję to **interfejsy**.

Działają prawie tak samo jak klasy abstrakcyjne z tą różnicą, że nie przechowują w ogóle stanów – to są same nagłówki metod i stałe. Nie mają ciał metod, nie mają zmiennych. Przydają się, kiedy chcemy mieć coś, co mówi dzieciom jak mają być zbudowane.

Przykładowy interfejs

Calculator.java (rodzic)

```
public interface Calculator {
    public static final double PI = 3.14;
    public static final double E = 2.71;

    public abstract void clearMemory();
    public abstract void saveMemory(double memory);
}
```

SimpleCalculator.java (dziecko – implementuje interfejs)

```
public class SimpleCalculator implements Calculator {
    private Double memory;

    @Override
    public void clearMemory() {
        memory = null;
    }

    @Override
    public void saveMemory(double memory) {
        this.memory = memory;
    }

    double add(double a, double b) {
        return a + b;
    }
}
```

Przypomnijmy sobie, jak wygląda klasa abstrakcyjna i tu wróćmy.

Jak widać, interfejs robi prawie to samo. Różnice są następujące:

- W interfejsie nie można tworzyć zwykłych pól. Mogą to być tylko stałe statyczne (`static final`).
- W klasie abstrakcyjnej mogliśmy oznaczyć część metod jako abstrakcyjne, wtedy musieliśmy implementować tylko te metody, a pozostałe, nie-abstrakcyjne, były zapewnione przez rodzica. W interfejsie wszystkie metody są domyślnie abstrakcyjne i wszystkie musimy zaimplementować w klasie-dziecku.

Tak naprawdę powyższy zapis interfejsu jest przekombinowany, bo wszystkie pola są domyślnie oznaczone jako "public static final", a wszystkie metody jako "public abstract". Możemy napisać nasz interfejs w taki sposób i wydzie na to samo:

```
public interface Calculator {  
    double PI = 3.14;  
    double E = 2.71;  
  
    void clearMemory();  
    void saveMemory(double memory);  
}
```

No dobra, ale po co nam to jest, skoro to wszystko możemy zrobić w klasach abstrakcyjnych? Dla ułatwienia. Zapis interfejsu jest dużo krótszy i wiadomo co robi. Jeżeli nie potrzebujemy bajerów z klas abstrakcyjnych, tylko prosty szablon budowy do klas to możemy użyć interfejsu.

Dodatkowo, **możemy zaimplementować kilka interfejsów na raz**. W przypadku klas możemy dziedziczyć po tylko jednej.

```
public class Child implements Interface1, Interface2 { ... }
```

A ponadto, możemy jednocześnie implementować interfejs i dziedziczyć po klasie:

```
public class Child extends Parent implements Interface1 { ... }
```

Jak wspominałem, ważną cechą interfejsu jest to, że nie zachowuje stanu - tworząc obiekt klasy, która implementuje interfejs, sam interfejs nie jest tworzony.

Kiedy używamy konstruktora klasy, która dziedziczy po rodzicu, to konstruktor rodzica też jest wywoływany – automatycznie.

Metoda domyślna

Interfejs pozwala na zdefiniowane **metody domyślnej**. Jest to metoda, która zawiera ciało:

```
public interface Calculator {  
    public static final double PI = 3.14;  
    public static final double E = 2.71;  
  
    public abstract void clearMemory();  
    public abstract void saveMemory(double memory);  
  
    default void getVersion() {  
        System.out.println("Wersja kalkulatora: 1.0.0");  
    }  
}
```


Tu pojawia się pytanie – czy ty, drogi skurwysynu, nas okłamałeś? Powiedziałeś, że interfejsy nie mogą mieć ciała metod.

To prawda, przepraszam. Metody domyślne zostały dodane w ramach kompatybilności wstecznej i najlepiej z nich nie korzystać, jeżeli nie musimy. Docelowo służą do tego, aby dodawać do istniejących interfejsów nowe metody bez rozpierdalania ich działania.

Czym to się różni zwykłej metody z ciałem w klasie abstrakcyjnej? W praktyce - prawie niczym. Szczerze, można zapomnieć o istnieniu czegoś takiego.

cz. 10 - klasa w klasie

Przypadek 1 - zwykła klasa w zwykłej klasie

Car.java

```
public class Car {
    private String model;

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

    public class Engine {
        public String engineType;

        public void setEngine() {
            if (model.equals("BMW")) {
                engineType = "N74";
            }
            // ...
        }
    }
}
```

Main.java

```
Car bmw = new Car("BMW");
Car.Engine engine = bmw.new Engine();
engine.setEngine();

engine.engineType; // N74
```

Tworzymy klasę w klasie. Obie te klasy są powiązane – w metodzie **setEngine()** wykorzystujemy pole **model** z klasy głównej. Moglibyśmy zrobić to w dwóch osobnych klasach, przekazując w tej metodzie po prostu obiekt klasy Car. Po co to więc istnieje? Głównie do grupowania. Jeżeli dana klasa jest powiązana TYLKO z jakąś jedną klasą to możemy ją w niej umieścić, połączyć i mieć z głowy latanie po kilku plikach lub bawienie się w przekazywanie obiektów.

Pojawia się pojebany zapis:

```
Car.Engine engine = bmw.new Engine();
```

Wy tłumaczmy, jak on działa.

Klasa Engine znajduje się w klasie Car. Aby mieć do niej dostęp musimy napisać Car.Engine.

Ta klasa NIE JEST STATYCZNA. To oznacza, że JEST POWIĄZANA Z OBIEKTEM, więc musimy użyć jej konstruktora.

W normalnej klasie, konstruktor wyglądałby tak:

```
Engine engine = new Engine();
```

My za to mamy zagnieżdżoną klasę, ale nie możemy napisać tak:

```
Car.Engine engine = new Car.Engine();
```

Powyższy zapis byłby dobry wtedy, gdyby klasa Engine była statyczna. Ale u nas ta klasa nie jest statyczna – jest powiązana z obiektem „bmw”. Więc musimy napisać:

```
Car.Engine engine = bmw.new Engine();
```

Pojebane, prawda?

Przypadek 2 - zagnieżdżona klasa statyczna

Waiter.java

```
public class Waiter {
    public static class Money {
        private double value;
        private String type;

        public Money(double value, String type) {
            this.value = value;
            this.type = type;
        }
    }

    private String firstName, lastName;
    private Money salary;

    public Waiter(String firstName, String lastName, Money salary) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.salary = salary;
    }

    public String getSalary() {
        return salary.value + " " + salary.type;
    }
}
```

Main.java

```
Waiter.Money salary = new Waiter.Money(2137, "PLN");  
Waiter w = new Waiter("Jan", "Paweł", salary);
```

Klasa jest statyczna - to oznacza, że możemy ją tworzyć bez obiektu. To, podobnie jak w przypadku pierwszym, służy głównie do grupowania klas, które nigdy nie będą wykorzystywane w innych miejscach. Można to bezproblemowo rozdzielić na kilka klas, jeżeli mamy na to ochotę (a pewnie mamy).

W klasie `Waiter` po prostu wykorzystujemy podklasę `Money` jako typ pola **salary**.

cz. 11 - typ generyczny

(PRZECZYTAJ TEN ROZDZIAŁ DOPIERO PO DOTARCIU DO LIST)

Typy generyczne

Chcemy napisać metodę, która wspiera różnego rodzaju typy danych. W naszym wypadku ambitnie - wszystkie typy danych. Jak to zrobić? Znamy już przeładowanie metod, ale musielibyśmy ich zrobić z dobre 100, aby nasza klasa działała na każdym typie danych w programie.

Typy generyczne (generics) nam w tym pomogą. Typ generyczny dodaje do naszej metody / klasy parametr - typ, który możemy w niej wykorzystywać. Np. zamiast pisać wszędzie "int", piszemy "T", które później samo się podmieni dla każdego rodzaju typu danych.

Klasa generyczna

Do zobrazowania działania użyjmy przykładu. Chcemy napisać nowy rodzaj listy - nazwijmy go **KotyraList**. Nasza lista musi wspierać KAŻDY element, więc aby to zrobić musi mieć ona typ generyczny:

```
public class KotyraList<T> {  
    private T[] elements;  
}
```

Typ generyczny w klasie ustalamy dodając <T> po nazwie klasy. Potem możemy korzystać z tej nazwy w każdym miejscu w klasie, w którym chcielibyśmy użyć normalnego typu.

Tak naprawdę to możemy nasz typ nazwać inaczej, to bez znaczenia:

```
public class KotyraList<Dupa> {  
    private Dupa[] elements;  
}
```

Ale nie jesteśmy chamami, wróćmy do T. Następnie dodajemy konstruktor - wykorzystujemy nasz typ generyczny tak, jak każdy inny typ:

```
public class KotyraList<T> {  
    private T[] elements;  
  
    public KotyraList(T[] elements) {  
        this.elements = elements;  
    }  
}
```

Aby utworzyć obiekt takiej klasy musimy zrobić następująco:

```
String[] elements = {"jeden", "dwa", "trzy"};
KotyraList<String> lista = new KotyraList<String>(elements);
```

Jak widać, to jest zwykłe tworzenie obiektu z tą różnicą, że po nazwie klasy podajemy typ, jaki przekazujemy temu obiektowi - w tym wypadku **String**. Obiekt w swoim wnętrzu wszystko ładnie podmieni, a my niczym się nie przejmujemy.

Tutaj pojawia się problem typów generycznych - nie wspierają one typów prostych (np. int, boolean, double). Aby to rozwiązać używamy typów złożonych: **Double**, **Integer** itd. Każdy typ prosty ma taki odpowiednik.

Metoda generyczna

Chcemy napisać metodę **get()**, która zwróci element tablicy o podanym indeksie:

```
public class KotyraList<T> {
    private T[] elements;

    public KotyraList(T[] elements) {
        this.elements = elements;
    }

    public T get(int index) {
        return elements[index];
    }
}
```

Metoda **get()** zwraca nam typ T, który podmieni się w obiekcie na podany przez nas (podczas tworzenia obiektu) typ. Ale, plot twist, to nie jest metoda generyczna. To zwykła metoda wykorzystująca typ generyczny z klasy.

Nie szkodzi - czujemy się super, stoi nam fujara, ale pojawia się kolejny problem - chcemy napisać metodę **printArray()**, która przyjmuje tablicę elementów dowolnego typu i ją wyświetla. To oznacza, że ta metoda **nie jest powiązana z istniejącym obiektem**, tylko z samą klasą. Tutaj nie możemy użyć typu T z naszej klasy, bo ten typ podajemy PRZY TWORZENIU OBIEKTU.

Aby to rozwiązać możemy ustawić typ generyczny tylko dla metody:

```
public static <E> void printArray(E[] array) {
    for (E element : array) {
        System.out.println(element);
    }
}
```

Typ "E" jest dostępny tylko w metodzie `printArray()`. Piszemy go przed typem zwracanym. Dodatkowo - to oznacza, że możemy tworzyć metody generyczne w klasach, które nie są generyczne. Taka przyjemna ciekawostka.

Kilka typów generycznych na raz

Możemy ustawić kilka typów generycznych jednocześnie i z nich korzystać:

```
public class KotyraList<T1, T2> {
    private T1[] elements_t1;
    private T2[] elements_t2;

    public KotyraList(T1[] elements_t1, T2[] elements_t2) {
        this.elements_t1 = elements_t1;
        this.elements_t2 = elements_t2;
    }

    public static <E> void printArray(E[] array) {
        for (E element : array) {
            System.out.println(element);
        }
    }
}
```

W tym przykładzie nasza niesamowita lista, nazwana całkowicie przypadkowym nazwiskiem, posiada dwie tablice elementów o różnych (może się zdarzyć, że nawet tych samych) typach. Aby utworzyć obiekt takiej klasy robimy to samo, co wcześniej, ale po prostu podajemy kilka typów:

```
String[] e1 = {"jeden", "dwa", "trzy"};
Integer[] e2 = {1, 2, 3};
KotyraList<String, Integer> lista = new KotyraList<String, Integer>(e1, e2);
```

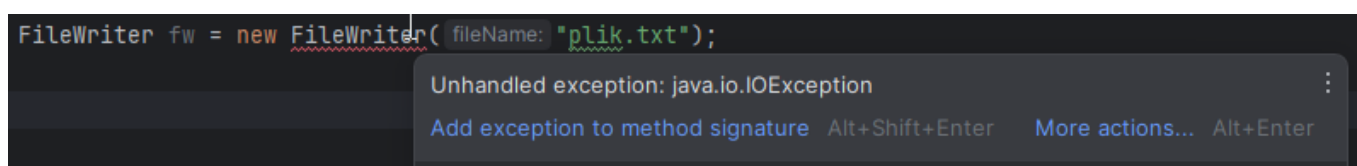
cz. 12 - wyjątki, zarządzanie błędami

Niektóre metody mogą wyrzucić **błąd** (inaczej **wyjątek**). Jeżeli to olejemy, to kiedy on nastąpi cały program się wyjeebie, zesra i zakończy działanie. Raczej takiej sytuacji nie chcemy - lepiej, żeby program dalej działał, ale pokazał np. wiadomość z błędem.

Mamy dwa główne sposoby na poradzenie sobie z wyjątkami:

- Owiniecie problematycznego kodu w **try-catch**.
- Dodanie do nagłówka metody **"throws NazwaBłędu"**.

Przykładowo, wyobraźmy sobie sytuację, gdzie pracujemy z plikami (o nich później). Jak wiadomo, plik może się spierdolić: może nie istnieć, może być uszkodzony itp. W takim wypadku np. otworzenie go będzie niemożliwe. Java wtedy wyrzuci błąd o nazwie "IOException". Naszą odpowiedzialnością jest to, aby coś z nim zrobić.



Try-catch

Oto nasz początkowy kod:

```
FileWriter fw = new FileWriter("plik.txt");
fw.write("test");
fw.close();
```

FileWriter zwraca błąd "IOException". Użyjemy **try-catch**, aby go "złapać" i wyświetlić komunikat błędu. Program dalej będzie działał poprawnie, nawet jeśli nie uda się otworzyć pliku:

```
try {
    FileWriter fw = new FileWriter("plik.txt");
    fw.write("test");
    fw.close();
} catch (IOException e) {
    System.out.println(e.getMessage());
}
```

Pojawia się tu sporo nowych rzeczy, więc je omówmy.

W **try { ... }** umieszczamy kod, który może powodować błędy. Jeżeli taki błąd wystąpi, to kod p rzestanie być wykonywany i przejdziemy do **catch**.

W **catch (...)** { ... } podajemy w nawiasie typ błędu (InteliJ pomoże nam go znaleźć), a w nawiasach klamrowych umieszczamy operacje, jakie w takiej sytuacji wykonamy. W naszym przypadku wyświetlimy tylko wiadomość z błędu.

Kod w **try** wykona się w całości tylko wtedy, gdy nie wystąpi żaden błąd. Kod w **catch** wykona się tylko wtedy, gdy błąd wystąpi.

Możemy dodać **catch** kilka razy, jeżeli nasz kod wywala różne rodzaje błędów:

```
try {
    FileWriter fw = new FileWriter("plik.txt");
    fw.write("test");
    fw.close();
} catch (FileNotFoundException e) {
    System.out.println(e.getMessage());
} catch (FileSystemNotFoundException e) {
    System.out.println(e.getMessage());
}
```

Oba **catch** robią to samo (oba wyświetlają wiadomość) - możemy je złączyć w jeden:

```
try {
    FileWriter fw = new FileWriter("plik.txt");
    fw.write("test");
    fw.close();
} catch (FileNotFoundException | FileSystemNotFoundException e) {
    System.out.println(e.getMessage());
}
```

Możemy do tego wszystkiego dodać jeszcze klauzulę **finally**, które wykona się ZAWSZE, niezależnie od tego, czy błąd wystąpi, czy nie:

```
try {
    FileWriter fw = new FileWriter("plik.txt");
    fw.write("test");
    fw.close();
} catch (IOException e) {
    System.out.println(e.getMessage());
} finally {
    System.out.println("Praca z plikiem zakończona");
}
```

Najczęściej w **finally** kończy się praca, np. zamyka pliki. Gdybyśmy mieli w **try** kilka metod, które zwracają błędy i np. otwarcie pliku się udało, ale zapisanie do niego danych nie, to plik się nie zamknie w ogóle, bo wykonywanie kodu się przerwie.

Dlatego poprawmy nasz kod:

```
FileWriter fw = null;
try {
    fw = new FileWriter("plik.txt");
    fw.write("test");
} catch (IOException e) {
    System.out.println(e.getMessage());
} finally {
    try {
        if (fw != null) {
            fw.close();
        }
    } catch (IOException e) {
        System.out.println(e.getMessage());
    }
}
```

Wygląda to zajębiście skomplikowanie, nie? To dlatego, że `fw.close()` także może wyrzucić błąd, który także musimy owinać w try-catch. Zanim powiesz „chuj z tym, nie będę zabezpieczać kodu”, spójrz niżej.

Java pozwala na coś fajniejszego - **try-with-resources**:

```
try (FileWriter fw = new FileWriter("plik.txt")) {
    fw.write("test");
} catch (IOException e) {
    System.out.println(e.getMessage());
}
```

To zwykłe try-catch tyle, że do try dodajemy nawiasy okrągłe, w których tworzymy nasz obiekt `FileWriter`. Nie musimy bawić się w zamykanie, bo Java robi to za nas. Jeżeli tworzymy kilka obiektów, to w nawiasach okrągłych tworzymy je po kolei po średnikach.

Throws

Możemy stwierdzić, że to pierdolimy i nie będziemy się tym błędem zajmować. Zajmiemy się nim kiedy indziej, w innym miejscu (np. w Mainie albo innej metodzie).

W takim wypadku możemy dodać do nagłówka metody **"throws"**. W taki sposób oznaczamy, że nasza metoda może zwrócić jakiś błąd, ale martwić się tym błędem będziemy podczas jej używania, a nie pisania.

Test.java

```
public class Test {  
    public static void openFile() throws IOException {  
        FileWriter fw = new FileWriter("plik.txt");  
        fw.write("test");  
        fw.close();  
    }  
}
```

Main.java

```
Test.openFile();  
  
Unhandled exception: java.io.IOException
```

Wyrzucanie błędów

Czasem zdarzy się tak, że będziemy chcieli aby w naszej metodzie (np. jakimś setterze) była możliwość ustawienia tylko i wyłącznie liczb dodatnich. No to co? Prosta sprawa:

```
public class Test {  
    private int number;  
  
    public void setNumber(int number) {  
        if (number > 0)  
            this.number = number;  
    }  
}
```

Co się stanie, jeżeli będziemy mieli liczbę ujemną? Nic.

Nie jest to fajne, bo tak właściwie nie wiemy nawet, że coś jest nie tak. Nie wyświetli się przecież żaden komunikat.

Moglibyśmy dodać tutaj **println()** z wiadomością typu "Użyj liczb dodatnich", ale to nie spowoduje przerwania programu. Reszta programu się wykona tak, jakby metoda **setNumber()** coś ustawiła.

Nie podoba nam się to. W tym celu możemy samemu wyrzucić błąd:

```
public void setNumber(int number) {  
    if (number > 0)  
        this.number = number;  
    else  
        throw new IllegalStateException("Number cannot be negative");  
}
```

Wyjątek (błąd) to zwykły obiekt - musimy go utworzyć za pomocą "new". Aby go wyrzucić używamy klauzuli "throw". Rodzajów wyjątków jest wiele. Jeżeli chcielibyśmy je poznać, to najlepiej będzie jak wpiszemy "throw new " w IntelliJ i poszukamy czegoś na wyświetlonej liście.

Teraz wykonanie takiego kodu w Main.java:

```
Test t = new Test();  
t.setNumber(-5);
```

Spowoduje błąd i zatrzyma program:

```
Exception in thread "main" java.lang.IllegalStateException Create breakpoint : Number cannot be negative  
    at Test.setNumber(Test.java:8)  
    at Main.main(Main.java:8)
```

Tworzenie własnego rodzaju wyjątków

Jeżeli chcemy stworzyć własny obiekt-wyjątek to wystarczy stworzyć klasę, która dziedziczy po "Exception", a w konstruktorze użyć konstruktora rodzica z daną wiadomością, np.:

```
public class NegativeNumberException extends Exception {  
    public NegativeNumberException() {  
        super("Number cannot be negative");  
    }  
}
```

Aby użyć naszego nowego wyjątku robimy to samo, co w przypadku zwykłych wyjątków:

```
public void setNumber(int number) throws NegativeNumberException {  
    if (number > 0)  
        this.number = number;  
    else  
        throw new NegativeNumberException();  
}
```

cz. 14.1 - techniki w klasach 2 - buildery

Wyobraźmy sobie, że mamy pokurwioną klasę, która ma dużo różnych pól i kombinacji. Np. klasę House, w której konstruktorze możemy ustawić wszystko - kolor ścian, dodatkowe budowle (np. garaże, strych, piwnica) i tak dalej. Tu mamy pewien problem - aby te dane przekazać musielibyśmy pisać w konstruktorze ogromną ilość argumentów, np.:

House.java

```
public class House {
    public enum Color { BLUE, RED, GREEN, WHITE, BLACK };
    private Color insideWallColor, outsideWallColor;
    private double squareFootage;
    private boolean onSale;

    public House(Color wallColor, double squareFootage, boolean
hasGarage, boolean onSale) {
        this.insideWallColor = wallColor;
        this.outsideWallColor = wallColor;
        this.squareFootage = squareFootage;
        this.onSale = onSale;
        this.hasGarage = hasGarage;
        if (hasGarage) {
            this.squareFootage += 20;
        }
    }
}
```

Main.java

```
House.Color colorOutside = House.Color.RED;
House.Color colorInside = House.Color.BLACK;
House house = new House(colorOutside, colorInside, 50, false, true);
```

Żeby utworzyć obiekt z takiej klasy musimy się namęczyć niemiłosiernie. To tylko 5 pól, a i tak zajęło to nam to aż 3 linijki. Dodatkowo, nagłówek konstruktora już nie mieści się w jednej linijce i ciężko się z niego korzysta, bo jest po prostu nieczytelny. Ewidentnie da się to zrobić lepiej.

Do tego służą **buildery** - są to klasy (wzorzec projektowy), które budują nasz obiekt i go zwracają. Kiedy chcemy utworzyć obiekt klasy House, to używamy po prostu Buildera i mówimy mu kolejno, np.: ustaw kolor ściany na czerwony, dodaj garaż, wystaw na sprzedaż.

W aktualnym przypadku kiedy nie chcemy mieć garażu to musimy to dokładnie określić w argumencie w konstruktorze. Co jeśli żaden z domów nie ma garażu? I tak musimy podawać w konstruktorze, że nie chcemy garażu. Aż szkoda na to klawiatury.

Przeróbmy naszą klasę i dodajmy budowniczego (uwaga, to będzie najbardziej skomplikowany kod dotychczas):

House.java

```
public class House {
    public enum Color { BLUE, RED, GREEN, WHITE, BLACK };

    private Color insideWallColor, outsideWallColor;
    private Double squareFootage;
    private Boolean onSale, hasGarage;

    public static class Builder {
        private Double squareFootage = 0.0;
        private Color wallColor;
        private Boolean onSale = false, hasGarage = false;

        public Builder addSquareFootage(Double squareFootage) {
            this.squareFootage += squareFootage;
            return this;
        }

        public Builder wallColor(Color wallColor) {
            this.wallColor = wallColor;
            return this;
        }

        public Builder setOnSale() {
            this.onSale = true;
            return this;
        }

        public Builder addGarage() {
            this.hasGarage = true;
            this.squareFootage += 10;
            return this;
        }

        public House build() {
            House house = new House();

            house.insideWallColor = this.wallColor;
            house.outsideWallColor = this.wallColor;
            house.squareFootage = this.squareFootage;
            house.onSale = this.onSale;
            house.hasGarage = this.hasGarage;

            return house;
        }
    }

    private House() {}
}
```

To, co tu się dzieje jest pojebane, ale jednocześnie fajne. Zaczniemy po kolei.

W klasie House mamy prywatny konstruktor. To sprawia, że nie możemy utworzyć obiektu House w zwykły sposób, np. w Mainie:

```
House house = new House(); // błąd
```

Dalej - utworzyliśmy klasę w klasie, którą nazwaliśmy **Builder** - to właśnie ona służy do budowania naszej klasy głównej. Dlatego konstruktor w klasie House tylko by przeszkadzał.

Jeżeli nam się to podoba, to klasa Builder mogłaby być umieszczona w innym pliku, nie ma to większego znaczenia.

W klasie Builder przepisaaliśmy prawie wszystkie pola z klasy głównej z kilkoma ważnymi zmianami:

- `outsideWallColor` i `insideWallColor` to tak naprawdę jeden kolor - tak było to ustawione w konstruktorze klasy House. Żeby zapisać ich kolor wystarczy nam jedno pole.
- Ustawiliśmy wartości domyślne pól, np. domyślnie ustawiamy, że dom nie ma garażu i że nie jest wystawiony na sprzedaż.

Dlaczego to zrobiliśmy? Dlaczego nie używamy po prostu pól z klasy wyżej? Ponieważ klasa Builder buduje nasz dom stopniowo - na początek ustawiamy wszystkie wartości poszczególnymi metodami, a kiedy skończymy to dopiero wtedy wywołujemy metodę, która zbiera to w kupę i tworzy obiekt House. Potrzebujemy "lokalnych" pól, aby te wartości przechować przed ich wpierdoleniem do obiektu House.

Następnie tworzymy metody, które coś nam będą ustawiać lub zmieniać. Tutaj mamy pełną dowolność z jednym ważnym wyjątkiem: te metody muszą zwracać obiekt Builder. Przyjrzyjmy się tej metodzie:

```
public Builder wallColor(Color wallColor) {  
    this.wallColor = wallColor;  
    return this;  
}
```

W nagłówku metody widzimy "Builder", czyli zwraca ona obiekt Builder. Słowo "**this**" w returnie to właśnie jest nasz obiekt. O tym słowie było mówione na samym początku, kiedy mieliśmy takie same nazwy pól w np. setterze:

```
void setNumber_1(int number_1) {  
    this.number_1 = number_1;  
}
```

Aby ustawić `number_1` z naszej klasy musieliśmy dopisać `this`. Tak naprawdę zapis `this.number_1` oznacza, że jest to pole z naszego obiektu klasy. Dzięki temu Java wie, które `number_1` jest które.

Wracamy do Buildera. Metody robią różne rzeczy - to już nie są zwykłe settery. Jedna metoda może ustawiać pole na true, druga metoda może dodawać do innego pola jakąś liczbę, trzecia robić jeszcze coś innego. To wszystko zależy od nas.

Na samym końcu mamy metodę **build()**, która tworzy nowy obiekt klasy House, zapisuje do niego nasze dane (tak, jak to było w konstruktorze w oryginalnej klasie House) i go zwraca.

Teraz najważniejsza część zabawy, użycie buildera. Zaczniemy od rozpoczęcia budowy. Stworzymy nowy obiekt `House.Builder`:

Main.java

```
House.Builder builder = new House.Builder();
```

Samo w sobie to nic nie robi - nie napisaliśmy konstruktora w klasie `Builder`, więc Java dodała pusty, który nic nie ustawia ani nic nie zmienia. Póki co mamy w środku tego obiektu zapisane wartości domyślne, tak jak ustaliliśmy to w klasie.

Teraz możemy podać Builderowi po kolei co chcemy zrobić, np. w tym domu chcemy, aby kolor ścian był czerwony i chcemy dodać 20 metrów kwadratowych. Dopisujemy:

```
House.Builder builder = new House.Builder();
builder.addSquareFootage(20.0);
builder.wallColor(House.Color.RED);
```

Nie chcemy mieć garażu. Ale nie musimy nic dopisywać, bo domyślnie go nie mamy. Dopiszmy jeszcze, że ten dom ma być wystawiony na sprzedaż:

```
House.Builder builder = new House.Builder();
builder.addSquareFootage(20.0);
builder.wallColor(House.Color.RED);
builder.setOnSale();
```

Fenomenalnie, powiedzmy, że patodeweloperka została skończona i dom nam się podoba. Ale uwaga: to jest dopiero `House.Builder`. Teraz musimy zbudować na podstawie wprowadzonych przez nas informacji obiekt `House`. Aby to zrobić używamy metody **`build()`**, która zwraca nasz nowy obiekt:

```
House.Builder builder = new House.Builder();
builder.addSquareFootage(20.0);
builder.wallColor(House.Color.RED);
builder.setOnSale();
House house = builder.build();
```

Gotowe - utworzyliśmy obiekt `House` w sposób czytelny - każda instrukcja jest jasno opisana, a rzeczy, których nie potrzebowaliśmy po prostu nie podawaliśmy. Używa się tego nieźle, ale ten zapis jest długi.

Przypominamy sobie, że nasze metody (np.. `wallColor`) zwracały obiekt `Builder`. Pozwala nam to na zrobienie tego samego w znacznie krótszy sposób:

```
House house = new House.Builder()
    .addSquareFootage(20.0)
    .wallColor(House.Color.BLUE)
    .setOnSale()
    .build();
```

Tworzy się to właśnie dzięki zwracaniu w metodach tego samego obiektu. Gdybyśmy go nie zwracali, to nie moglibyśmy zrobić takiego zapisu.

Możemy sobie wyobrazić to tak:

```
new House.Builder() // zwraca obiekt Builder
```

```
(new House.Builder()).addSquareFootage(20.0) // używamy metody na obiekcie Builder; to wszystko  
zwraca obiekt Builder
```

```
((new House.Builder()).addSquareFootage(20.0)).wallColor(house.Color.BLUE) // i tak dalej...
```

cz 14.2 - techniki w klasach 2 - dekoratory

Wyobraźmy sobie, że mamy klasę **Tea**. Do herbatki można dodawać cukier, cytrynę, niektóre pojeby dodają też mleko. Wszystkich kombinacji jest dużo - można zrobić herbatę z cukrem i cytryną, z samą cytryną, ze wszystkimi dodatkami itp.

To co, Builder? Uspokój się kurwa, uczymy się innej metody. Dałoby to radę zrobić z builderem, ale zrobimy to **dekoratorem**.

Buildery przygotowały kolejno wszystkie pola, a na koniec budowały klasę. Dekoratory dodają lub zmieniają w klasie różne elementy na bieżąco i na bieżąco zwracają „udekorowany” obiekt.

Najlepiej to zrozumieć na przykładzie. Stwórzmy na początek samą klasę podstawową. Powiedzmy, że jedyne co ma, to ilość kalorii, którą będziemy zmieniać zależnie od dodatków:

Tea.java

```
public class Tea {  
    private static final int kcal = 0;  
  
    public int getKcal() {  
        return kcal;  
    }  
}
```

Main.java

```
Tea herbatka = new Tea();
```

Pole `kcal` to stała statyczna. Nie ma możliwości, żeby herbata bez dodatków miała inną ilość kalorii niż 0, a przynajmniej nie w tym uniwersum, więc możemy to w taki sposób ustawić - obiekty i tak nie będą tej wartości zmieniać.

Tworzymy pierwszego dekoratora - herbatę z dodatkiem cukru:

```
public class TeaWithSugar extends Tea {  
    private static final int kcal = 20;  
    private Tea baseTea;  
  
    public TeaWithSugar(Tea baseTea) {  
        this.baseTea = baseTea;  
    }  
  
    @Override  
    public int getKcal() {  
        return baseTea.getKcal() + kcal;  
    }  
}
```

Dziedziczymy tę klasę po Tea - logiczne, musimy mieć dostęp do jej pól i metod.
Dodatkowo edytujemy istniejące pole - ilość kcal zmieniamy na 20; tyle u nas ma sama herbata z cukrem.

Zapisujemy obiekt, z którym będziemy pracować w polu baseTea.

I teraz zaczynamy dekorowanie. Musimy coś dodać do poprzedniej wersji herbaty - w tym przypadku bazowej (dlatego zapisaliśmy jej obiekt). Dodatek samego cukru ma 20 kalorii, czyli do poprzedniej ilości kalorii musimy dodać 20. To właśnie robi metoda getKcal().

Spróbujmy tego użyć:

```
Tea herbatka = new Tea();  
TeaWithSugar herbatkaZCukrem = new TeaWithSugar(herbatka);
```

Jak widać - do zwykłej herbaty dodajemy cukier przez naszą nową klasę i powstaje nowy obiekt - herbata z dodatkiem cukru. "Udekorowaliśmy" nasz poprzedni obiekt, zwiększając liczbę kalorii.

Możemy w ten sposób dodać resztę dodatków, np. przepyszne mleko:

```
public class TeaWithMilk extends Tea {  
    private static final int kcal = 30;  
    private Tea baseTea;  
  
    public TeaWithMilk(Tea baseTea) {  
        this.baseTea = baseTea;  
    }  
  
    @Override  
    public int getKcal() {  
        return baseTea.getKcal() + kcal;  
    }  
}
```

Ta klasa jest zbudowana identycznie jak TeaWithSugar. Teraz możemy połączyć kilka dodatków:

```
Tea herbatka = new Tea();  
Tea herbatkaZCukrem = new TeaWithSugar(herbatka);  
Tea herbatkaZCukremIMlekiem = new TeaWithMilk(herbatkaZCukrem);
```

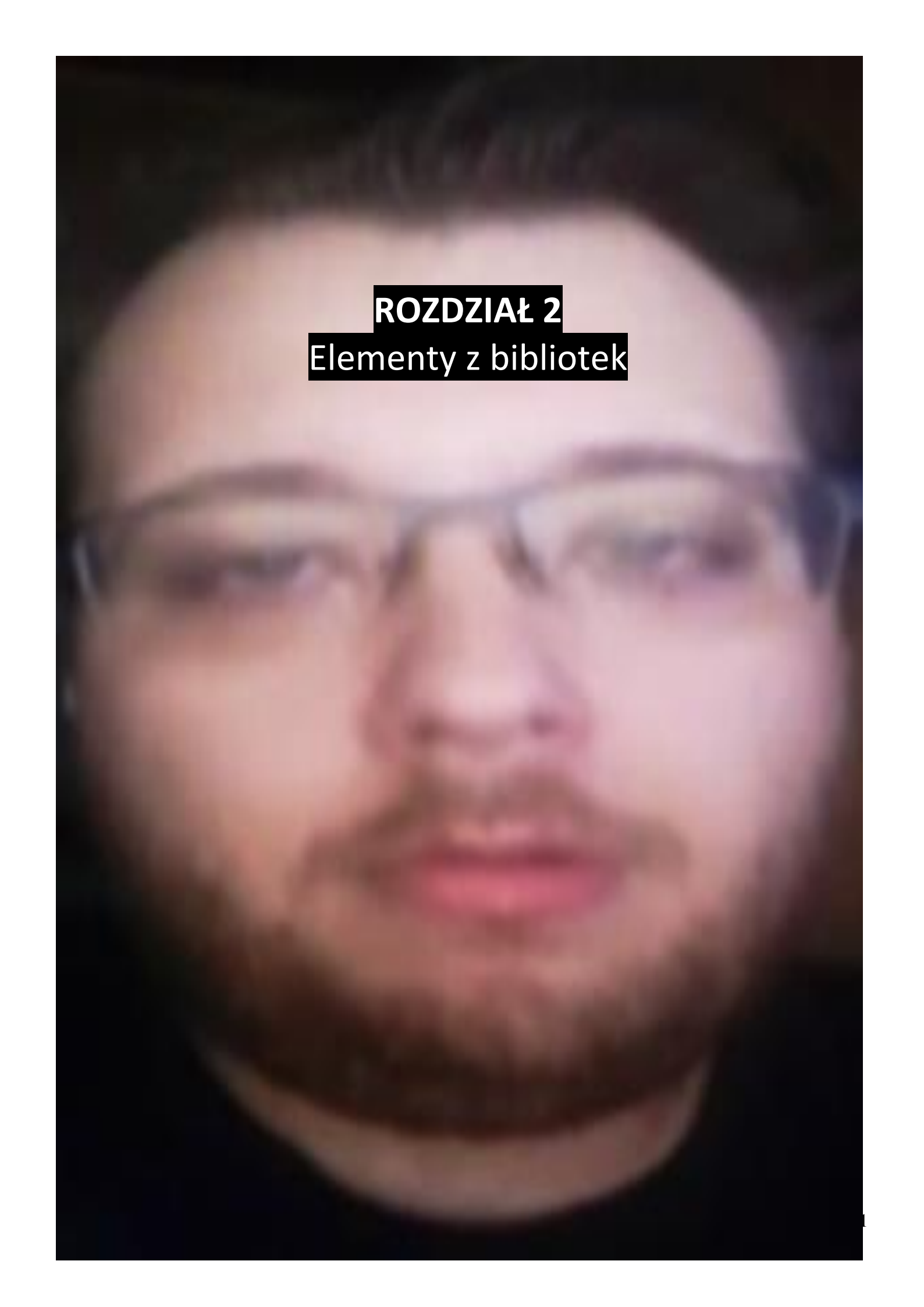
"Udekorowaliśmy" herbatę z cukrem, dodając do niej mleko i zwiększając liczbę kalorii o 30. Teraz możemy sprawdzić jak bardzo przytyjemy:

```
int kcal = herbatkaZCukremIMlekiem.getKcal();  
System.out.println(kcal); // 50, bo 0 + 20 + 30
```

Spostrzegawcze osoby zauważą, że coś tu nie gra:

```
Tea herbatkaZCukrem = new TeaWithSugar(herbatka);
```

Do zmiennej o typie "Tea" przypisaliśmy obiekt o typie "TeaWithSugar". To jest poprawny zapis. Jeżeli jakaś klasa jest potomkiem innej, to możemy ją przypisywać do typu rodzica. Nic to nie zmienia, ale czasami jest krócej.



ROZDZIAŁ 2

Elementy z bibliotek

cz. 15 - JDK i importowanie

JDK

Poznaliśmy już wszystkie najważniejsze elementy Javy. W normalnych okolicznościach byłoby zajęcie i mielibyśmy wolne, ale okazuje się, że Java posiada coś, co nazywa się "Java Development Kit" (JDK). W wielkim skrócie - to całe środowisko Javy, które zawiera w sobie ogromną ilość wbudowanych bajerów.

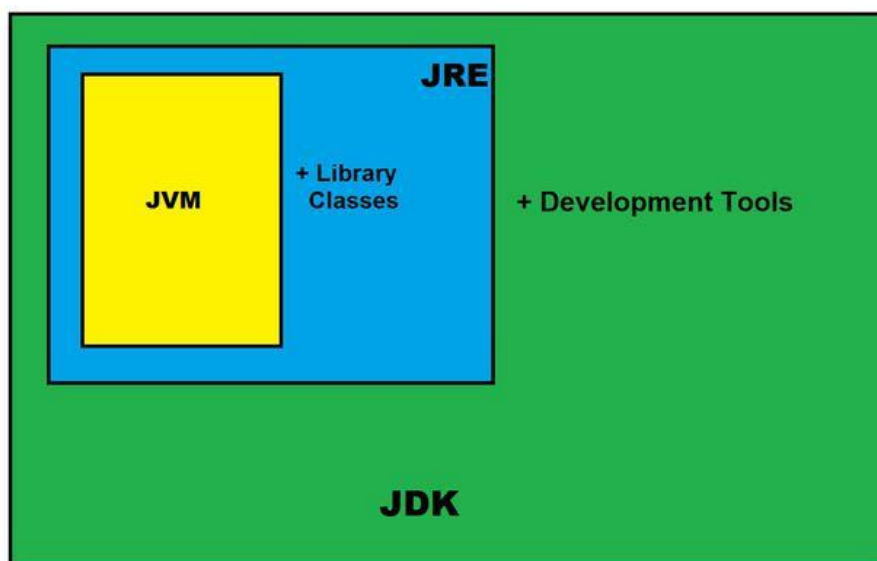
JDK to jest tak naprawdę maszyna wirtualna, kompilator, interpreter i zbiór bibliotek w jednym. JDK zawiera "Java Runtime Environment" (JRE), który zawiera maszynę wirtualną Javy (JVM) i różnego rodzaju biblioteki, które będziemy w tym rozdziale poznawać.

Znudziły ci się tablice? To masz 20 nowych rodzajów list.

Chcesz otworzyć plik? Dajemy ci na to 10 różnych sposobów!

Może chcesz zapisać datę w zmiennej? Spoko mordeczko - wybierz sobie jeden z 30 typów, który dla ciebie przygotowaliśmy.

W tym rozdziale opisane będą poszczególne elementy z tych bibliotek. To oznacza, że będziemy mieli kilka(naście) sposobów na zrobienie tego samego, a wybór tego jednego sposobu zależy od nas.

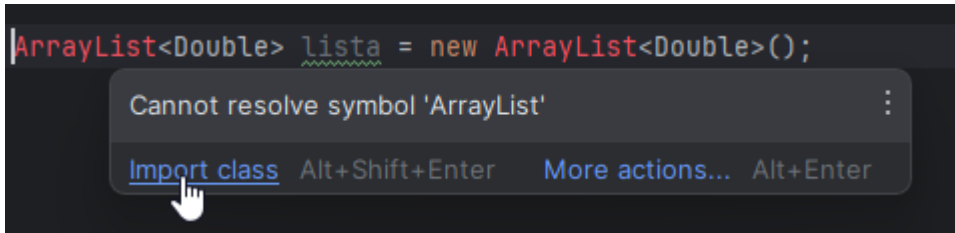


Importowanie

Aby użyć elementów z jakiejś biblioteki / pakietu, musimy je zaimportować. Aby to zrobić na początku pliku należy dodać klauzulę **import** z daną ścieżką, np.:

```
import java.util.ArrayList;  
import java.util.List;
```

Jakbyśmy to robili ręcznie to byśmy się zajebali. Na szczęście IntelliJ robi to za nas i jak użyjemy jakiegoś elementu to zaimportuje się sam. A nawet jeśli nie, to będziemy mogli go zaimportować automatycznie:



cz. 16 - listy, mapy, sety

Przypomnijmy sobie tablice. Aby je utworzyć musieliśmy znać ich rozmiar. Dodatkowo, chuja mogliśmy na nich zrobić. Jak dodać na nich następny element? Jak usunąć z nich pierwszy? Ciężka sprawa.

Java posiada coś, co nazywa się **listą**. W skrócie - to ulepszone tablice. Nie musimy podawać ich rozmiaru i posiadają wbudowane metody, które ułatwiają robotę.

Problem w tym, że rodzajów list jest całkiem dużo i każda ma swoje zastosowanie, plusy i minusy.

Interfejs "List"

Każda lista w Javie jest implementacją interfejsu **"List"**. Jak pamiętamy - interfejsy mówiły dzieciom jakie metody mają mieć. Prawdę mówiąc nie musimy nawet wiedzieć, że on istnieje, ale jest jedna ciekawa rzecz, o której zaraz powiemy.

Każda metoda z tego interfejsu jest wspólna dla każdego rodzaju list, między innymi mamy:

```
lista.size() // rozmiar listy
lista.isEmpty() // czy lista jest pusta
lista.contains(obiekt) // czy lista zawiera jakiś obiekt
lista.toArray() // zamienia listę na zwykłą tablicę
lista.add(obiekt) // dodaje obiekt
lista.remove(obiekt) // usuwa obiekt (pierwsze jego wystąpienie, jeżeli się powtarza)
lista.clear() // czyści listę
lista.get(index) // zwraca element o podanym indeksie
lista.set(index, obiekt) // ustawia element na dany indeks
lista.subList(i_od, i_do) // zwraca część listy od podanych zakresów
lista.sort(komparator) // sortuje listę
```

W normalnych tablicach nie ma takich rzeczy.

ArrayList

Standardowym typem listy jest **"ArrayList"**. Stosujemy ją, kiedy potrzebujemy zalet list, czyli m.in: brak stałego rozmiaru, łatwe dodawanie / usuwanie elementów, dodatkowe metody pomocnicze.

Tworzenie listy

```
ArrayList<String> napisy = new ArrayList<String>();
```

Pojawia się zapis: **<String>**. Nazywa się to "typem generycznym". Pełen opis typu generycznego znajduje się w części 11 - najlepiej dobrze się z tym zapoznać, bo będziemy teraz z nich korzystać cały czas.

Pamiętajmy, że gdybyśmy chcieli zrobić listę liczb całkowitych, to musimy użyć typu Integer, a nie int.

Wróćmy do interfejsu `List`. Jak wspomniałem, `ArrayList` go implementuje. To oznacza, że tworząc obiekt `ArrayList` możemy zrobić taki zapis:

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

To jest to samo, co zapis wyżej. Możemy sobie wybrać, jak będziemy pisać. Ja dla wygody będę pisał sposobem drugim, bo jest krócej.

Dodatkowo, możemy ominąć podawanie typu po prawej stronie, bo ustaliliśmy go już po lewej, a po co się powtarzać?:

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

Przykładowe operacje na liście

```
List<String> lista = new ArrayList<>();  
lista.add("test");  
lista.add("napis");  
System.out.println(lista.get(1));
```

LinkedList

Istnieje coś takiego jak „`LinkedList`”. Korzysta się z tego prawie identycznie jak z `ArrayList`. Ten rodzaj listy jest zoptymalizowany pod dodawanie / usuwanie danych z jej środka. Minusem jest to, że ta lista jest beznadziejna do odczytywania danych.

To przydaje się wtedy, gdy potrzebujemy bardzo szybkiej modyfikacji danych na liście i nie obchodzi nas wolniejszy czas dostępu do tych danych.

Czyli w naszym przypadku - nigdy się nie przyda.

Dla chętnych - utworzenie `LinkedList`:

```
LinkedList<Integer> lista = new LinkedList<>();
```

Interfejs "Map"

Dla ułatwienia uznajmy, że mapy to kolejny rodzaj list. W teorii różnią się od listy wszystkim - to całkowicie inny interfejs, ale w praktyce korzysta się z nich bardzo podobnie.

Mapy od list różnią się tym, że w listach wartości zapisywane są po **indeksach**, a w mapach po **kluczach**.

Indeksy mogą być tylko numeryczne i muszą być kolejno numerowane: 1, 2, 3, ...

Klucze mogą być wszystkim - numerami, stringami, obiektami i nie muszą być ze sobą powiązane. Tak jak indeksy, nie mogą się powtarzać.

Mapy przydają nam się, kiedy chcemy zapisać dane, które mają własne identyfikatory, np. numery chorób: F71 - upośledzenie umysłowe w stopniu umiarkowanym

Nie moglibyśmy tego zapisać w normalnej `ArrayList` w prosty sposób. Tutaj z pomocą przychodzą mapy, które pozwalają nam na ustawienie "F71" jako klucza, a opisu choroby jako wartości.

Najważniejsze metody w interfejsie Map:

```
mapa.put(klucz, wartość) // dodaje element pod dany klucz
mapa.get(klucz) // pobiera wartość z danego klucza
mapa.remove(klucz) // usuwa element o podanym kluczu
mapa.isEmpty() // sprawdza czy mapa jest pusta
mapa.size() // zwraca ilość elementów
mapa.keySet() // zwraca zbiór kluczy
mapa.values() // zwraca zbiór wartości
```

HashMap

Jest to podstawowy rodzaj mapy, który implementuje interfejs Map. Nie gwarantuje on kolejności danych, dlatego musimy na to uważać (ale w sumie po co nam kolejność w mapach?).

Tworzenie obiektu:

```
HashMap<String, String> mapa = new HashMap<>();
```

Mamy tu dwa typy generyczne. Pierwszy to **typ klucza**, drugi to **typ wartości**. Jeżeli chcemy, aby naszymi kluczami były liczby, a wartościami jakieś opisy, to moglibyśmy napisać: `<Integer, String>`

Podobnie jak w listach, podczas tworzenia obiektu możemy użyć nazwy interfejsu (ale nic to nie zmienia):

```
Map<String, String> mapa = new HashMap<>();
```

Interfejs "Set"

Problemem map i list jest to, że wartości mogą się powtarzać. Jeżeli chcemy przechować unikatowe wartości, możemy użyć **setów**. Sety działają identycznie jak listy z tą różnicą, że nie pozwalają na zduplikowane wartości.

Najważniejsze metody w setach:

```
set.add(wartość) // dodaje wartość
set.remove(wartość) // usuwa wartość
set.size() // ilość elementów
set.contains(wartość) // czy zawiera wartość
set.toArray() // zwraca set w formie tablicy
set.clear() // czyści set
```

HashSet

Jest to podstawowa implementacja interfejsu Set - bazuje na HashMapach i także nie zachowuje kolejności.

Tworzenie obiektu:

```
HashSet<String> set = new HashSet<>();
```

Mamy tu jeden typ generyczny, który oznacza typ wartości, jakie są przechowywane w secie.

Możemy też przy tworzeniu obiektu używać nazwy interfejsu:

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

cz. 17 - string i StringBuilder

String

Na początku było mówione, że **String** to także jest klasa i zawiera kilka metod wbudowanych wartych omówienia.

Na początek wyjaśnijmy jedną rzecz - te dwa zapisy:

```
String str = "przykładowy napis";  
String str = new String("przykładowy napis");
```

Oznaczają one prawie to samo. Oba te zapisy zwracają jakiś obiekt. W przypadku pierwszego – „string literal”, w przypadku drugiego – zwykły obiekt String. Literał i obiekt String niczym się nie różnią. Różnica polega na tym, że podczas tworzenia literału, Java sprawdza, czy inny literał o tej samej treści istnieje. Jeżeli tak – to nie tworzy nowego, tylko używa istniejącego. Jeżeli nie – tworzy nowy. W przypadku obiektów zawsze tworzymy coś nowego.

Porównywanie napisów

Na pewno się teraz zastanawiasz: „Kto pytał?”. Wspominamy o tym dlatego, że ta wiedza jest ważna przy porównywaniu napisów.

Nie powinniśmy porównywać napisów za pomocą operatora „==”, ponieważ on tylko sprawdza, czy referencje na obiekty są takie same. O ile normalnie tworząc napisy nie zauważymy nic dziwnego, bo będziemy mieli tylko jeden literał, tak tworząc napisy używając sposobu nr 2 będziemy mieli problem:

```
String str1 = new String("przykładowy napis");  
String str2 = new String("przykładowy napis");  
str1 == str2 // false
```

Dzieje się tak, bo str1 i str2 to dwa różne obiekty.

Aby zrobić to poprawnie używamy metody **equals()**:

```
String str1 = new String("przykładowy napis");  
String str2 = new String("przykładowy napis");  
str1.equals(str2); // true
```

I najlepiej ją stosować nawet w przypadku normalnych stringów. IntelliJ sam powinien to podpowiedzieć.

Podstawowe metody Stringa

```
String str = "przykładowy napis str";
```

```
str.toLowerCase(); // "przykładowy napis str"
str.toUpperCase(); // "PRZYKŁADOWY NAPIS STR"
str.contains("napis"); // true
str.charAt(1); // 'r' (char)
str.startsWith("p"); // true
str.endsWith("s"); // false
str.length(); // 21
str.split(" "); // {"przykładowy", "napis", "str"}
str.split(" ", 1); // {"przykładowy", "napis str"} // dzieli tylko 1 raz
str.trim(); // usuwa puste znaki z początku i końca
```

```
String[] napisy = {"raz", "dwa", "trzy"};
```

```
String.join(" ", napisy); // "raz dwa trzy"
String.join(" - ", napisy); // "raz - dwa - trzy"
```

Wyrażenia regularne

Wyrażenia regularne, czyli **regexy** to skomplikowany temat - służą do sprawdzania czy ciąg znaków spełnia określony przez nas wzorzec.

Podstawowe oznaczenia ilości znaków:

```
* // 0 lub więcej
+ // 1 lub więcej
{3} // dokładnie 3 razy
{2,4} // od 2 do 4 razy
? // 1 raz lub w ogóle
```

Podstawowe oznaczenia typu znaku:

```
[0-9] // liczba od 0 do 9
[A-Z] // znak od A do Z
[A-Za-z] // znaki od A do Z i a - z
. // jeden dowolny znak
```

Przykładowe regexy:

```
^[0-9]{3}-[A-Z]* // 213-ABC, 997-XDDDDD, 123- itp.
.+ // a, ASDASDAS, abCdA123, 2137 itp.
```

Aby sprawdzić, czy napis spełnia wzorec, należy użyć metody `matches()`, np.:

```
str.matches(".*") // true lub false
```

Formatowanie napisu

Jeżeli nasz string jest długi i łączy wiele zmiennych to połączenie tego wszystkiego będzie cholernie nieczytelne. Możemy użyć metody `String.format()`, która uprości nam jego zapis, np.:

```
int wiek = 12;
String imie = "Jan";
String nazwisko = "Paweł";

String wynik = String.format("%s %s - %d lat", imie, nazwisko, wiek);
// "Jan Paweł - 12 lat"
```

Format napisu (1 argument) posiada pewne oznaczenia, które mówią jaki typ danych wrzucamy w dane miejsce w napisie:

Oznaczenie	Typ danych
%s	String
%b	Boolean
%d	Integer
%f	Float, Double
%.2f	Float, Double; dwa miejsca po przecinku

Po argumencie z formatem napisu podajemy po kolei nasze dane. Ich ilość jest nieograniczona.

StringBuilder

String ma jeden problem - jego modyfikacja jest bardzo wolna. Jeżeli chcielibyśmy zrobić ogromną pętlę, która cały czas dodaje coś do napisu to moglibyśmy się za naszego życia nie doczekać końca.

Robiąc takie coś tworzymy za każdym razem nowy obiekt Stringa:

```
String text = "";
for (int i = 0; i < 100; i++) {
    text = text + " napis";
}
System.out.println(text);
```

W takiej sytuacji IntelliJ nawet wyrzuci ostrzeżenie i zapyta: Co ty kurwa robisz? Pojechało cię? I ma rację. Tak się nie robi.

Naszym rozwiązaniem jest **StringBuilder**, który pozwala na szybką modyfikację Stringa.

Poprawmy nasz kod:

```
StringBuilder text = new StringBuilder();
for (int i = 0; i < 100; i++) {
    text.append(" napis");
}
System.out.println(text);
```

Zamiast normalnego Stringa utworzyliśmy obiektu **StringBuilder**. Aby coś do niego dopisać używamy metody **append()**. Reszta działa tak samo.

Ten kod jest dużo szybszy, teraz jest spora szansa, że doczekamy się końca tej pętli przed naszą śmiercią.

Uwaga!

Zauważmy, że w **println()** wyświetlamy zmienną **text**, a to nie jest String - to obiekt **StringBuilder**. Dlaczego to działa? Ponieważ **StringBuilder** posiada metodę **toString()**, która automatycznie konwertuje obiekt na napis. Gdybyśmy pisali metodę, która ma zwracać Stringa to musielibyśmy skonwertować **StringBuilder** na napis:

```
return text.toString();
```

cz. 18.1 - pliki - File

Klasa **File** pozwala nam na reprezentację pliku w formie obiektu w Javie. Ta klasa sama w sobie nie pozwala na czytanie / pisanie do pliku, ale jest pomocna kiedy korzystamy z innych klas, które pracują z plikami.

Warto też wiedzieć, że ta klasa wspiera też katalogi – ale nie będę tu tego opisywać.

Tworzenie obiektu File

```
File file = new File("plik.txt");
```

Uwaga! To nie znaczy, że plik **plik.txt** zostanie utworzony lub otwarty - po prostu utworzyliśmy reprezentację pliku na podanej ścieżce. Ale nie wiadomo, czy on istnieje, czy nie.

Ważne metody w klasie File

```
file.exists(); // sprawdza czy plik istnieje  
file.getName(); // zwraca nazwę pliku  
file.createNewFile(); // tworzy plik w podanej podczas tworzenia obiektu  
ścieżce  
file.delete(); // usuwa plik w podanej podczas tworzenia obiektu ścieżce
```


cz. 18.2 - pliki - FileWriter i FileReader

FileWriter

FileWriter służy do zapisywania ciągu znaków do pliku. To najbardziej podstawowy typ zapisywania do pliku.

Jeżeli plik docelowy nie istnieje, to **FileWriter** go stworzy.
Należy też pamiętać, że otwarty plik trzeba zamykać.

Uwaga! W poniższych przykładach używam stringa ze ścieżką "plik.txt", aby było prościej. Zamiast tego możemy stworzyć obiekt **File** i go użyć zamiast podawania ścieżki - zadziała to tak samo.

Utworzenie / otwarcie pliku i nadpisanie go:

```
FileWriter fw = new FileWriter("plik.txt");  
fw.write("napis");  
fw.close();
```

Domyślnie **FileWriter** nadpisuje plik. Aby dopisywał do niego dane należy dodać drugi argument:

```
FileWriter fw = new FileWriter("plik.txt", true);
```

FileWriter zapisuje dane na dysk dopiero przy wywołaniu metody **close()** albo **flush()**:

```
FileWriter fw = new FileWriter("plik.txt");  
fw.write("test"); // jeszcze nie ma tego w pliku  
fw.flush(); // teraz zostaje dodane do pliku  
fw.close();
```

FileReader

FileReader służy do czytania zawartości pliku w formie ciągu znaków. Czytanie następuje w formie strumienia danych znak po znaku.

Utworzenie FileReader'a i wczytanie danych do stringa:

```
String text = "";

FileReader fr = new FileReader("plik.txt");

int i;
while ((i = fr.read()) != -1) {
    text = text + (char)i;
}

fr.close();
```

Zawartość pliku zapiszemy do zmiennej "text".

W pętli za pomocą metody **read()** czytamy znak po znaku z pliku. Każdy pojedynczy znak zapisujemy w zmiennej **i**. Jeżeli **read()** zwróci wartość **-1**, to oznacza, że kolejnego znaku nie ma - wtedy pętla należy zakończyć.

Nie jest to wygodne i wygląda źle, bo trzeba się bawić w konwersje. Dlatego dalej opiszemy inne sposoby na zabawę z plikami.

Uwaga! W normalnych warunkach za takie dopisywanie do stringa dostalibyśmy wpierdol, bo jest to wolne. Powinniśmy tu skorzystać ze **StringBuilder**a, ale dla uproszczenia tego kodu i przyszłych tego nie robimy. Ale ty tak rób.

Uwaga nr 2! Pliki mogą wyrzucić błędy - każda zabawa z plikami powinna być owinięta **try-catchem** lub metoda powinna mieć dodane **throws**. Ponownie - żeby nie przekombinować teraz tego nie zrobiliśmy. Bez tego IntelliJ może nie odpalić programu.

cz. 18.3 - pliki - BufferedWriter i BufferedReader

Bawiąc się z plikami mamy duży problem. Po pierwsze - jeżeli pliki są duże lub zapisujemy dużo danych to będziemy się męczyć jak cholera. Po drugie - `FileReader` jest niewygodny.

Możemy do tego wykorzystać **BufferedWritera** i **BufferedReadera**, które optymalniej radzą sobie z dużymi ciągami znaków. Tak naprawdę obie te klasy przyjmują różnego rodzaju `Readery`, ale w naszym przypadku będziemy używać tylko tych z przedrostkiem „File”.

Uwaga! W poniższych przykładach ponownie używam stringa ze ścieżką „plik.txt”, aby było prościej. Zamiast tego możemy stworzyć obiekt `File` i go użyć zamiast podawania ścieżki - zadziała to tak samo.

BufferedWriter

BufferedWriter to „nakładka” (dekorator) na `FileWritera`, która dodaje buforowanie znaków. Bez zagłębiania się w technikalnia - dzięki temu zapisywanie do pliku jest dużo bardziej optymalne i szybsze. Praktycznie w 99% przypadków lepiej tego używać; chyba, że zapisujemy tylko jeden napis lub coś równie prostego.

```
BufferedWriter bw = new BufferedWriter(new FileWriter("plik.txt"));
bw.write("test");
bw.close();
```

BufferedWriter przyjmuje w argumencie `FileWritera`, ponieważ jest to „nakładka” (dekorator). Ta klasa tylko coś dodaje do istniejącej już klasy, ale nie zmienia sposobu jej użycia.

Dodatkowo, `BufferedWriter` posiada aż jedną dodatkową metodę wartą uwagi:

```
bw.write("test");
bw.newLine(); // dodaje nową linię
bw.write("test2");
```

Niesamowite.

Jeżeli chcemy, aby plik nie był nadpisywany, to robimy to samo, co ostatnio - do `FileWritera` dodajemy drugi argument:

```
BufferedWriter bw = new BufferedWriter(new FileWriter("plik.txt", true));
```

BufferedReader

Analogicznie, **BufferedReader** usprawnia `FileReader`a buforowaniem. Dodaje kilka nowych metod, które uprzyjemniają pracę z plikami.

```
String text = "";

BufferedReader br = new BufferedReader(new FileReader("plik.txt"));

String line;
while ((line = br.readLine()) != null) {
    text += line + "\n";
}

br.close();
```

Podobnie, jak w `FileReaderze`, czytamy w pętli `while` dane - tym razem nie po znaku, a linia po linii. Każdą linię dopisujemy do stringa.

Metoda **`readLine()`** zwraca wartość `null`, kiedy nie ma następnej linii do odczytania.

Ponownie zrobiliśmy pewien błąd - nie użyliśmy **`StringBuildera`**. Poprawmy się, bo policja już jedzie, a nie zaopatrzyliśmy się w makowiec.

Tym razem zrobimy klasę z metodą **`getAllLines()`**, która czyta wszystkie linie z pliku i zwraca je w formie `Stringa`:

Test.java

```
public class Test {
    public static String getAllLines(String filePath) {
        StringBuilder str = new StringBuilder();

        try (BufferedReader br = new BufferedReader(new FileReader(filePath))) {
            String line;
            while ((line = br.readLine()) != null) {
                str.append(line).append("\n");
            }
        } catch (IOException e) {
            System.out.println("Błąd: " + e.getMessage());
        }

        return str.toString();
    }
}
```

Main.java

```
String lines = Test.getAllLines("plik.txt");
System.out.println(lines);
```

Jeżeli nie rozumiesz jakiegokolwiek części tego kodu to mamy spory problem. Jeżeli wszystko rozumiesz to gratulacje - umiesz Javę w stopniu początkującym i prawdę mówiąc reszta omówionych tu rzeczy nie powinna być specjalnie trudna.

Wykorzystaliśmy tu:

- Klasy
- Metody statyczne
- StringBuilder
- Try-with-resources
- FileReader i BufferedReader

cz. 18.4 - pliki - Scanner

Scanner

Klasa **Scanner** służy do odczytu danych z jakiegokolwiek źródła. Możemy za jej pomocą odczytać dane z pliku.

Czym to się różni od `FileReader` / `BufferedReader`?

- `Scanner` ma mniejszą wydajność. Przy pracy z dużymi plikami będzie sobie gorzej radził. A jeżeli plik będzie za duży, to jest szansa, że nie zadziała w ogóle.
- `FileReader` tylko czyta plik znak po znaku. `Scanner` pozwala na wczytywanie z ciągu znaków poszczególnych danych, np. liczb.

Przeróbmy kod `BufferedReader` z poprzedniej części na `Scanner`:

```
String text = "";
File file = new File("plik.txt");

Scanner scanner = new Scanner(file);
while (scanner.hasNextLine()) {
    text += scanner.nextLine() + "\n";
}

scanner.close();
```

W **Scannerze** musimy podać obiekt `File` – tutaj sama ścieżka do pliku już nie zadziała. Cała zabawa ze `Scannerem` zaczyna się w jego metodach. Na przykład tutaj użyliśmy metody **`hasNextLine()`**, która sprawdza, czy w pliku jest następna linia. Tą linię pobieramy metodą **`nextLine()`**.

Ale możemy zrobić znacznie więcej. Dla przykładu - w pliku mamy następujący napis:
przykładowy napis z 1 numerem

Użyjemy klasy `Scanner`, aby pobrać z niego tylko tę liczbę:

```
File file = new File("plik.txt");
Scanner scanner = new Scanner(file);

int number = 0;
while (scanner.hasNext()) {
    if (scanner.hasNextInt()) {
        number = scanner.nextInt();
        break;
    }
    scanner.next();
}

scanner.close();
System.out.println(number);
```

Tworzymy pętlę, w której sprawdzamy, czy następny wyraz istnieje. Jeżeli tak, to w instrukcji warunkowej sprawdzamy, czy ten wyraz jest typu **int**. Jeżeli tak, to zapisujemy ten wyraz pod zmienną **number** i kończymy (przerywamy) pętlę.

scanner.next() na końcu pętli jest potrzebny, aby Scanner przeszedł na następny wyraz w wypadku, gdy wyraz nie będzie liczbą.

Na tokenizację jest lepszy sposób, więc w praktyce Scannera w tym celu raczej się nie używa, a jeżeli nie musimy nic z naszego pliku wyciągać, to nie ma sensu używać go w ogóle.

Najważniejsze metody w Scannerze

```
scanner.next(); // zwraca następne słowo (o ile nie ustawimy innego rozdzielnika)
scanner.useDelimiter("-"); // ustawia rozdzielnik na myślnik (next() będzie wtedy
rozdzielało po myślniku)
scanner.hasNext(); // czy jest następny element po rozdzielniku
scanner.nextInt(); // zwraca znalezione inta w elemencie - jeżeli go nie ma to
wyrzuca błąd
scanner.nextDouble();
scanner.nextBoolean();
// itd.
```

cz. 18.5 - pliki - FileOutputStream i FileInputStream

Poprzednie klasy do pracy z plikami czytały tylko ciągi znaków. Teraz opiszemy klasy, które będą pracować z plikami binarnymi.

Oczywiście tekst to też są dane binarne, ale do pracy z tekstem znacznie lepsze są klasy opisane wcześniej. Tu raczej chodzi o dane pokroju obiektów, liczb i innych dupereli.

FileOutputStream

FileOutputStream to podstawowa klasa do czytania danych binarnych (bajtów) z pliku. Używa się jej prawie identycznie jak **FileWritera**.

Tworzenie obiektu:

```
FileOutputStream fout = new FileOutputStream("plik.bin");
```

Aby zawartość była dopisywana do pliku dodajemy drugi argument:

```
FileOutputStream fout = new FileOutputStream("plik.bin", true);
```

Przykładowe użycie:

```
FileOutputStream fout = new FileOutputStream("plik.bin");
byte[] data = {1, 2, 3, 4, 5};
fout.write(data);
fout.close();
```

Należy pamiętać, że to wszystko trzeba zamknąć w **try-catch**.

Metoda **write()** zapisuje do pliku same bajty. Jeżeli chcemy zapisać np. ciąg znaków, to musimy go przerobić na bajty. Analogicznie jest z innymi typami danych.

Na przykład:

```
String str = "testowy napis";
byte[] bytes = str.getBytes();

fout.write(bytes);
```


FileInputStream

FileInputStream służy do odczytywania danych binarnych (bajtów) z pliku. Używa się go prawie identycznie jak **FileReader**.

Przykładowe użycie:

```
FileInputStream fin = new FileInputStream("plik.bin");

int b;
while ((b = fin.read()) != -1) {
    System.out.print((char) b);
}

fin.close();
```

Metoda **read()** zwraca **-1**, kiedy nie ma następnego bajtu do odczytania. Wtedy pętla musi się zakończyć. W pętli każdy bajt konwertujemy na typ **char** i wyświetlamy ten znak na ekranie.

cz. 18.6 - pliki - DataOutputStream i DataInputStream

Podobnie jak z `FileWriterem` i `BufferedWriterem`, `FileOutputStream` także posiada "nakładkę" (dekorator), która dodaje kilka ułatwiających życie metod.

DataOutputStream

DataOutputStream daje możliwość łatwego zapisywania typów prostych do pliku. `FileOutputStream` może zapisywać tylko bajty, a `DataOutputStream` wszystkie typy proste, np. liczby, znaki.

Używa się go podobnie jak `BufferedWritera` z `FileWriterem`:

```
FileOutputStream fout = new FileOutputStream("plik.bin");
DataOutputStream dout = new DataOutputStream(fout);

int[] arr = {2, 1, 3, 7};
for (int number : arr) {
    dout.writeInt(number);
}

dout.close();
```

Tworzymy tu pętlę, w której do pliku zapisujemy po kolei każdy element tablicy. Elementy są zapisywane jako inty, a nie bajty – tak jak to było wcześniej.

`DataOutputStream` posiada nowe metody do zapisywania danych o typie prostym, m.in.:

```
dout.writeInt(dane);
dout.writeBoolean(dane);
dout.writeChar(dane);
dout.writeDouble(dane);
...
```

DataInputStream

DataStream ułatwia odczytywanie typów prostych z danych binarnych.

Jak można się już spodziewać, używa się go podobnie jak `BufferedReader`:

```
FileInputStream fin = new FileInputStream("plik.bin");
DataInputStream din = new DataInputStream(fin);

List<Integer> list = new ArrayList<>();

while (din.available() > 0) {
    list.add(din.readInt());
}

din.close();
```

Metoda **available()** zwraca liczbę pozostałych do odczytania bajtów. W pętli odczytujemy wszystkie bajty jako inty i dodajemy je do listy.

`DataInputStream` również posiada nowe metody do wczytywania danych:

```
din.readInt();
din.readBoolean();
din.readChar();
din.readDouble();
...
```

cz. 18.7 - pliki - ObjectOutputStream i ObjectInputStream

Jeżeli myślisz, że to koniec zabawy z plikami binarnymi to się kurwa grubo mylisz. Nie poddawajmy się, lecimy dalej.

ObjectOutputStream i **ObjectInputStream** służą do zapisywania i wczytywania obiektów z/do formy danych binarnych. Aby obiekt dało się tak zapisać, musi on implementować interfejs "**Serializable**":

```
public class Test implements Serializable {  
    public Test() {}  
}
```

Aby klasę dało się serializować, najwyższa klasa w hierarchii dziedziczenia musi mieć bezparametrowy konstruktor. To znaczy: jeżeli klasa `Child` dziedziczy po klasie `Parent`, to aby dało się serializować `Child`, klasa `Parent` musi mieć konstruktor bez żadnego parametru.

Pojebane.

ObjectOutputStream

ObjectOutputStream służy do zapisywania obiektów w formie danych binarnych.

Na przykład, zapiszmy obiekt klasy `Test` do pliku:

```
Test test = new Test(21, 37);  
  
FileOutputStream fout = new FileOutputStream("dane.bin");  
ObjectOutputStream oout = new ObjectOutputStream(fout);  
  
oout.writeObject(test);  
  
oout.close();
```

ObjectInputStream

Może to być zaskoczeniem, ale używamy go do wczytania obiektów z danych binarnych.

```
FileInputStream fin = new FileInputStream("dane.bin");
ObjectInputStream oin = new ObjectInputStream(fin);

Test test = (Test) oin.readObject();

oin.close();
```

Metoda **readObject()** wczytuje obiekt z danych binarnych. Zapisujemy ten obiekt do zmiennej **test**. Przy wczytywaniu obiektu nie wiemy czym on wcześniej był i skąd on pochodzi, więc musimy rzutować jego typ. W naszym przypadku na **Test**.

cz. 18.8 - Files i Paths

Pewnie myślisz sobie "ja pierdole znowu kolejny sposób na pracę z plikami?". Ale obiecuję, ten jest naprawdę zajebisty.

Paths i Files

Klasa **Path** to zastępstwo dla **File**, omawianego na początku plików (część 18.1). Tak naprawdę, to **File** jest już przestarzałe i używa się go tylko do pracy ze starymi klasami.

Obiekt **Path** to referencja do pliku, a tak właściwie do danej ścieżki, na której może znajdować się plik (lub katalog):

```
Path path = Paths.get("plik.txt");
```

Ten obiekt sam w sobie nie ma za dużo metod. Co najwyżej jakieś pobranie nazwy pliku, nazwy katalogu lub inne tego typu bzdety. Zabawa zaczyna się, kiedy połączymy go z klasą **Files**.

Możemy oczywiście zrobić to samo (a nawet i dużo więcej) co w klasie **File**:

```
Files.exists(path); // czy plik istnieje
Files.notExists(path); // czy plik nie istnieje
Files.createFile(path); // tworzy plik
Files.delete(path); // usuwa plik
Files.deleteIfExists(path); // usuwa plik, jeżeli istnieje (nie wywala błędu)
Files.createDirectory(path); // tworzy katalog
```

Ale zajmijmy się najważniejszym. Jak wczytać wszystkie linie z pliku? Jakaś pętla? Lecenie znak po znaku? HAMUJ SIĘ KURWA, żyjemy w XXI wieku. Wystarczy zrobić to:

```
List<String> lines = Files.readAllLines(path);
```

Jak wczytać całą zawartość pliku bez dzielenia na linie?

```
String fileContent = Files.readString(path);
```

A jak zapisać coś do pliku? Powiedzmy, że mamy taką listę:

```
List<String> lines = new ArrayList<>();
lines.add("Linia 1");
lines.add("Linia 2");
lines.add("Linia 3");
```

Bardzo prosto, mianowicie tak:

```
Files.write(path, lines);
```

Aby włączyć tryb dopisywania zawartości dodajemy kolejny argument:

```
Files.write(path, lines, StandardOpenOption.APPEND);
```

Uwaga! Jeżeli plik nie istnieje, to wystąpi błąd. Jeżeli chcemy dopisywać zawartość lub stworzyć plik to dodajemy drugi tryb:

```
Files.write(path, lines, StandardOpenOption.CREATE,  
StandardOpenOption.APPEND);
```

Ok, wygląda prosto i niesamowicie. Jakie są wady?

Jest jedna wada - każda operacja, np. **Files.write()** otwiera i zamyka plik automatycznie. To znaczy, że jeżeli mamy pętlę, w której używamy **Files.write()** to będzie się to całkiem wolno wykonywać. Na szczęście łatwo takie problemy rozwiązać - wystarczy nie używać **Files.write()** w pętli. Genialne.

cz. 19 - Random

Klasa **Random** służy do generacji losowych danych, nie tylko liczb. Możemy wygenerować m.in. inty, booleany, double i inne proste typy danych.

Tworzenie obiektu:

```
Random generator = new Random();
```

Najważniejsze metody

```
generator.nextBoolean(); // true lub false
generator.nextDouble(); // liczba z przedziału 0 - 1, np. 0.51
generator.nextDouble() * 10; // liczba z przedziału 0 - 10
generator.nextInt(); // liczba całkowita z całego przedziału inta (+ ujemne)
generator.nextInt(10); // liczba całkowita od 0 do 9
generator.nextInt(10 + 1 - 5) + 5; // liczba z zakresu 5-10
...
```


cz. 20 - data i czas

Do reprezentacji daty i czasu w Javie istnieją następujące klasy:

- **LocalDate** - data
- **LocalTime** - czas
- **LocalDateTime** - data i czas
- **DateTimeFormatter** - formatuje datę z/do stringa

Aktualny czas i godzina

```
LocalDateTime current_datetime = LocalDateTime.now();
LocalDate current_date = LocalDate.now();
LocalTime current_time = LocalTime.now();
```

Metoda **now()** w każdej z powyższych klas zwraca aktualny czas. Pierwsza linijka zwraca obiekt z aktualną godziną i datą, druga linijka tylko z aktualną datą a trzecia tylko z aktualną godziną.

Formatowanie daty i czasu

Jeżeli chcemy wyświetlić lub wczytać datę to najpewniej chcemy to zrobić w określonym formacie. Możemy się posłużyć klasą **DateTimeFormatter**, która sformatuje datę do/ze stringa o podanym formacie.

```
LocalDateTime current = LocalDateTime.now();
DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
System.out.println( current.format(formatter) ); // 2024-04-23 20:34:40
```

Tworzymy obiekt (formatter) **DateTimeFormatter** z metody **ofPattern()**. Podajemy w niej w jaki sposób nasz czas i data mają wyglądać. Potem używamy metody **format()**, która znajduje się w obiekcie z naszą datą. W argumencie podajemy utworzony wcześniej formatter.

Najważniejsze symbole do formatu czasu i daty

Symbol	Krótki opis	Dokładniejszy opis
y	rok	-
M	miesiąc	Miesiąc w roku
d	dzień	Dzień w miesiącu
H	godzina	Godzina w dobie
m	minuta	Minuta w godzinie
s	sekunda	Sekunda w minucie
W	tydzień	Tydzień w miesiącu
S	część sekundy	Dokładność do 4 miejsc

cz. 21 - lambda; interfejs funkcyjny

Interfejs funkcyjny

Interfejs funkcyjny to tak naprawdę funkcja, którą przypisujemy do zmiennej lub przekazujemy w argumentcie:

```
Function<Integer, String> funkcja = (x) -> {  
    return String.format("Podano liczbę %d", x);  
};
```

Na początku ustalamy jej typy danych. U nas jest to funkcja z dwoma typami generycznymi - pierwszy typ to typ argumentu, drugi typ to typ zwracany funkcji.

Po prawo piszemy tzw. wyrażenie lambda. Przed strzałką w nawiasie podajemy po kolei nazwy argumentów, a po strzałce podajemy ciało funkcji.

Jeżeli ciało funkcji ma tylko jedną linijkę, to możemy to skrócić w następujący sposób:

```
Function<Integer, String> funkcja = (x) -> String.format("Podano liczbę %d", x);
```

Dodatkowo, jeżeli mamy tylko jeden argument to możemy nie pisać nawiasów:

```
Function<Integer, String> funkcja = x -> String.format("Podano liczbę %d", x);
```

Aby użyć tej funkcji używamy metody **apply()** z odpowiednimi argumentami:

```
Function<Integer, String> funkcja = x -> String.format("Podano liczbę %d", x);  
funkcja.apply(5); // "Podano liczbę 5"
```

Przypisywanie istniejących metod do interfejsu funkcyjnego

Jeżeli mamy już jakąś istniejącą funkcję (czyli metodę w klasie), to możemy ją spokojnie przypisać do zmiennej o typie funkcyjnym, np.:

Test.java

```
public class Test {  
    public static Integer add5(Integer a) {  
        return a + 5;  
    }  
}
```

Main.java

```
Function<Integer, Integer> fun = Test::add5;  
int result = fun.apply(5);  
System.out.println(result); // 10
```

Ten dziwny zapis - **Test::add5** oznacza, że odwołujemy się do metody `add5()` z klasy `Test`. Co najważniejsze – ta metoda się nie odpala. To jest tylko samo odwołanie. To nam się przydaje, bo często chcemy przekazać do np. metody sortującej istniejące już metody z innych klas, których nie chcielibyśmy od razu wykonywać.

Przekazywanie funkcji do metody

```
public class Logger {  
    public static void print(String message) {  
        System.out.println(message);  
    }  
}
```

Mamy prostą klasę z metodą `print()`. Chcemy do tej metody dodać drugi parametr, którym będzie funkcja. Ta funkcja ma się wykonać przed wyświetleniem wiadomości na ekranie i coś z tą wiadomością zrobić.

Dodajemy więc interfejs funkcyjny:

```
public class Logger {  
    public static void print(String message, Function<String, String> action) {  
        message = action.apply(message);  
        System.out.println(message);  
    }  
}
```

W drugim parametrze metody dodaliśmy funkcję (interfejs funkcyjny, gdyby ktoś się srał o nazewnictwo), która przyjmuje stringa i zwraca stringa. I tak też się dzieje w metodzie - używamy tej funkcji, przekazując jej stringa `message` i to, co ona zwraca zapisujemy z powrotem do tej zmiennej.

Wywołajmy tę metodę w Mainie:

```
Function<String, String> action = str -> str.toUpperCase();  
Logger.print("Testowa wiadomość", action); // TESTOWA WIADOMOŚĆ
```

Wszystko śmiga. Co ciekawe, powyższy zapis funkcji „**action**” też da się skrócić:

```
Function<String, String> action = String::toUpperCase;
```

Nasze wyrażenie lambda (prawa strona) działa identycznie, jak samo użycie metody `toUpperCase` z klasy `String`, więc możemy po prostu użyć jej referencji. Obie te wersje działają tak samo, a IntelliJ często nam podpowiada takie rzeczy.

Przekazywanie istniejącej metody do funkcji

Ale co jeśli jesteśmy jebanymi leniami i nie chce nam się za każdym razem pisać funkcji, kiedy chcemy wyświetlić wiadomość używając naszej klasy `Logger`?

To dobre podejście, szkoda na to czasu. Możemy zrobić gotową metodę od razu w klasie `Logger`, a potem jej używać. Na przykład dodajmy metodę `addTime()`:

Logger.java

```
public class Logger {
    public static void print(String message, Function<String, String> action) {
        message = action.apply(message);
        System.out.println(message);
    }

    public static String addTime(String message) {
        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("HH:mm:ss");
        LocalTime current = LocalTime.now();
        return String.format("[%s] %s", current.format(formatter), message);
    }
}
```

Main.java

```
Logger.print("Testowa wiadomość", Logger::addTime);
// [23:12:44] Testowa wiadomość
```

W klasie `Logger` stworzyliśmy metodę `addTime()`, która dodaje aktualną godzinę do wiadomości.

W `Main`ie używamy referencji do tej metody. Dzięki temu nie musimy za każdym razem tworzyć takiej samej funkcji tylko tworzymy ją raz, a potem korzystamy z niej gdzie tylko chcemy.

Typy interfejsów funkcyjnych

`Function<argument, return>`

Function to zwykły interfejs funkcyjny. Wymaga jednego argumentu i tego, aby funkcja coś zwracała:

```
Function<Integer, Boolean> isEven = x -> x % 2 == 0;
```

`Consumer<argument>`

Consumer wymaga jednego argumentu ale **nic nie zwraca**:

```
Consumer<String> printName = name -> System.out.println("Siema, " + name);
printName.accept("Wojtek"); // Siema, Wojtek
```

Aby wywołać tę funkcję, używamy metody `accept()`.

Predicate<argument>

Predicate przyjmuje jeden argument i zwraca wartość prawda-fałsz (boolean). Służy do testowania warunków:

```
Predicate<Integer> isEven = x -> x % 2 == 0;  
isEven.test(5); // false
```

Tę funkcję wywołuje się metodą **test()**.

Supplier<return>

Supplier nie przyjmuje argumentów i coś zwraca, np.:

```
Supplier<Double> getRandomNumber = () -> Math.random();  
getRandomNumber.get(); // 0.11305
```

Pustym nawiasem () oznacza się brak argumentów. Funkcję wywołuje się metodą **get()**.

Comparator

O komparatorze będzie mowa w części poświęconej sortowaniu.

cz. 22 - strumienie

Warto sobie przypomnieć jak działały klasy buildery i lambdy, ponieważ na strumieniach operuje się bardzo podobnie. Nie jest to wymagane, ale na pewno pomoże.

Strumienie

Strumienie (stream) pomagają nam w modyfikacji danych. Załóżmy, że przetwarzamy zawartość pliku. Musimy pominąć trzy pierwsze linijki, do każdej linijki dodać jakiś prefiks, każdą kończącą się na "a" usunąć, a na koniec to wszystko dodać do listy.

Normalnym sposobem, tzn.: robiąc `BufferedReader`, dodając wszystko do listy, usuwając co drugi element listy w pętli itd. Idzie się zająć. A przecież jest po co żyć.

Za pomocą strumieni da się to zrobić czytelnie w kilka linijek:

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

Files.lines(Paths.get("plik.txt"))
    .skip(3)
    .map(line -> String.format("[Prefiks] %s", line))
    .filter(line -> !line.endsWith("a"))
    .limit(1)
    .forEach(lines::add);
```

Strumień działa podobnie jak **Builder**, to znaczy: możemy cały czas dodawać do niego kolejne metody z działaniami, a na samym końcu wykonujemy funkcję kończącą, która wszystko sfinalizuje i zwróci wynik.

Files.lines() czyta każdą linię z pliku i wszystko konwertuje na strumień.

skip() pomija linie. W tym przykładzie pomijamy 3 pierwsze linie.

map() służy do modyfikacji każdej linii. W tym przykładzie do każdej linii dodajemy prefiks „[Prefiks] ”.

filter() zostawia tylko te linie, które spełniają jakiś warunek. W tym przykładzie zostawia tylko te, które NIE KOŃCZĄ się na "a".

limit() usuwa linie, które wykraczają poza ustalony limit elementów. W tym przykładzie zostawia tylko jedną linię.

forEach() jest funkcją finalizującą wszystkie działania. Tworzy pętle po każdym elemencie w strumieniu. W naszym przykładzie każdy element zapisujemy do listy „lines”.

Zauważmy, że pierwotna metoda - **Files.lines()** nic nie zwraca. To właśnie dlatego potrzebujemy funkcji finalizującej, w której zrobimy coś z wynikami. Bez tego nic by się nie zmieniło ani nie wyświetliło.

Każda linijka za coś odpowiada i możemy się w miarę łatwo domyślić co robi. Bez strumieni to samo zrobilibyśmy w 5x dłuższy sposób.

Zobrazujmy to wszystko jeszcze raz, tym razem z przykładowymi danymi. Powiedzmy, że w pliku mamy takie linie:

```
Linia numer jeden  
Linia numer dwa  
Linia numer dwa  
Linia numer dwa  
Linia numer trzy  
Linia numer cztery
```

Używając metody **Files.lines()** czytamy te linie i konwertujemy to wszystko na strumień, z którym musimy coś zrobić.

Strumień przekazujemy do metody **skip()**, która usuwa trzy pierwsze linie. Zostaje nam:

```
Linia numer dwa  
Linia numer trzy  
Linia numer cztery
```

Strumień z takimi danymi przekazywany jest dalej - do metody **map()**, która dodaje do każdej linii prefiks:

```
[Prefiks] Linia numer dwa  
[Prefiks] Linia numer trzy  
[Prefiks] Linia numer cztery
```

A strumień z takimi danymi przekazywany jest do metody **filter()**, która u nas usuwa linie kończące się na "a":

```
[Prefiks] Linia numer trzy  
[Prefiks] Linia numer cztery
```

To wszystko przekazane jest do metody **limit()**, która zostawia tylko jedną, pierwszą linię:

```
[Prefiks] Linia numer trzy
```

Pora to wszystko sfinalizować i coś z tym zrobić. Przekazujemy te dane do metody **forEach()**, która każdą z tych linii doda do listy "lines".

Koniec. Operacje ze strumieniem zostały zakończone. Bez strumieni nie moglibyśmy użyć tych wszystkich metod, więc warto się tego nauczyć.

Bardzo dużo klas wspiera strumienie. Aby stworzyć strumień z np. listy wystarczy napisać:

```
lista.stream()
```


cz. 23 - sortowanie

Komparator

Komparator to rodzaj interfejsu funkcyjnego, który ma za zadanie wykonać porównanie dwóch elementów i zwrócić:

- `<0` jeżeli $a < b$
- `0` jeżeli $a = b$
- `>0` jeżeli $a > b$

Tworzenie komparatora:

```
Comparator<Integer> comparator = (a, b) -> a - b;
```

Powyższy komparator porównuje dwa Integery i zwraca wynik operacji **a - b**. Jeżeli posortujemy tablicę według tego komparatora to dostaniemy liczby posortowane rosnąco.

Komparator posiada tylko jedną przydatną dla nas metodę:

```
comparator.reversed() // odwraca sortowanie
```

Sortowanie tablicy

Sortowanie domyślne:

```
int[] numbers = {2, 1, 3, 7};  
Arrays.sort(numbers); // 1, 2, 3, 7
```

Sortowanie z komparatorem:

```
Comparator<Integer> comparator = (a, b) -> b - a;  
Integer[] numbers = {2, 1, 3, 7};  
Arrays.sort(numbers, comparator); // 7, 3, 2, 1
```

Sortowanie z odwróconym komparatorem:

```
Comparator<Integer> comparator = (a, b) -> b - a;  
Integer[] numbers = {2, 1, 3, 7};  
Arrays.sort(numbers, comparator.reversed()); // 1, 2, 3, 7
```

Sortowanie strumieni

Sortowanie domyślne:

```
List<Integer> numbers = Arrays.asList(2, 1, 3, 7);

numbers = numbers.stream()
    .sorted()
    .collect(Collectors.toList()); // 1, 2, 3, 7
```

Tworzymy listę `numbers` z wartościami: 2, 1, 3, 7.

Następnie generujemy z niej strumień i go sortujemy metodą **`sorted()`**.

W metodzie finalizującej **`collect()`** konwertujemy wynik na listę (ponieważ jest strumieniem) i go „zbieramy”. Ten zebrany wynik zapisujemy do zmiennej `numbers`.

Sortowanie z komparatorem:

```
Comparator<Integer> comparator = (a, b) -> b - a;
List<Integer> numbers = Arrays.asList(2, 1, 3, 7);

numbers = numbers.stream()
    .sorted(comparator)
    .collect(Collectors.toList()); // 7, 3, 2, 1
```

Sortowanie z odwróconym komparatorem:

```
Comparator<Integer> comparator = (a, b) -> b - a;
List<Integer> numbers = Arrays.asList(2, 1, 3, 7);

numbers = numbers.stream()
    .sorted(comparator.reversed())
    .collect(Collectors.toList()); // 1, 2, 3, 7
```



KONIEC