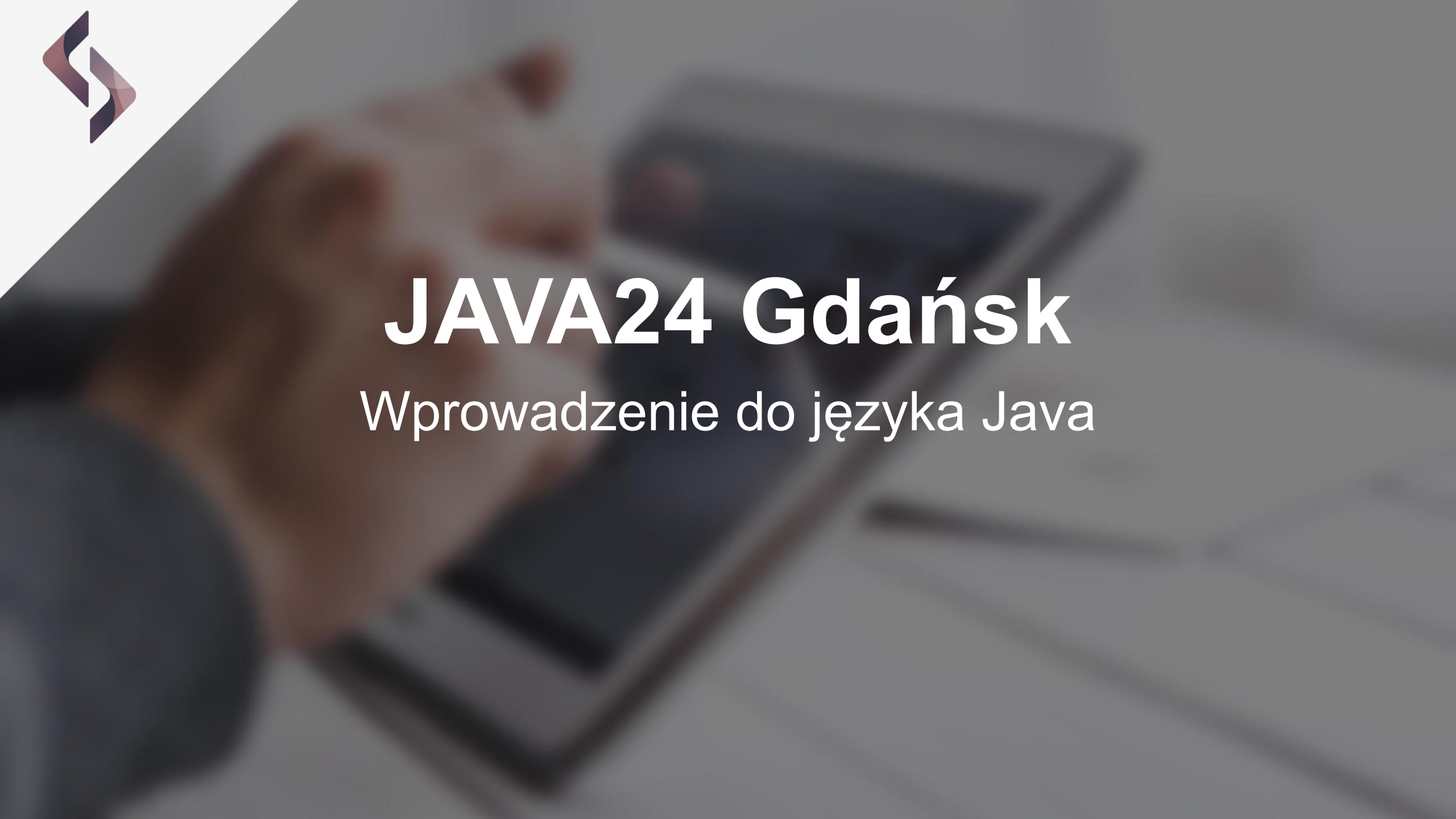


JAVA24 Gdańsk

Wprowadzenie do języka Java

Spotkanie #1

Wprowadzenie do języka Java

O mnie

Jarosław Skarżyński
j.p.skarzynski@gmail.com
Slack: @**Jarek Skarżyński**
LinkedIn: https://linkedin.com/in/jarosław-skarzynski

O Tobie

1. Jak się nazywasz ?
2. Dlaczego Java, dlaczego ten kurs ?
3. Jakie masz doświadczenie w IT ?
4. Jak wyobrażasz sobie ten kurs ?

Kontrakt

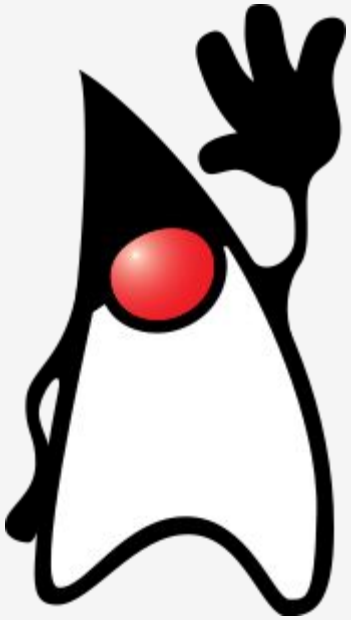
1. obecność i punktualność
2. wyciszamy komórki, wychodzimy jak musimy porozmawiać
3. pracujemy w domu - **dużo!**
4. pytamy, pytamy, pytamy - nie ma głupich pytań :)
    a. w czasie zajęć - w grupie lub indywidualnie
    b. slack - kanał **general** lub **direct message**
    c. ankiety
5. szacunek i wzajemna pomoc
6. przerwy
7. zaangażowanie
8. jesteśmy na “Ty”

Sposób pracy

1. **teoria + live-coding**
   a. śledzimy to co robi prowadzący
   b. nie przepisujemy kodu w tym samym czasie
   c. teoria to tylko wstęp, “zajawka”
2. **przykłady i zadania w kodzie** dostarczone przez prowadzących
   a. tylko do odczytu
   b. aktualizacja na każdych zajęciach
3. **zadania do rozwiązania** w czasie zajęć:
   a. samodzielnie przy swoim laptopie
   b. samodzielnie przy laptopie prowadzącego
   c. grupowo (2-3 osoby)
4. **zadania domowe**
5. **ankiety i testy**
6. code review - slack + github

Rozkład jazdy

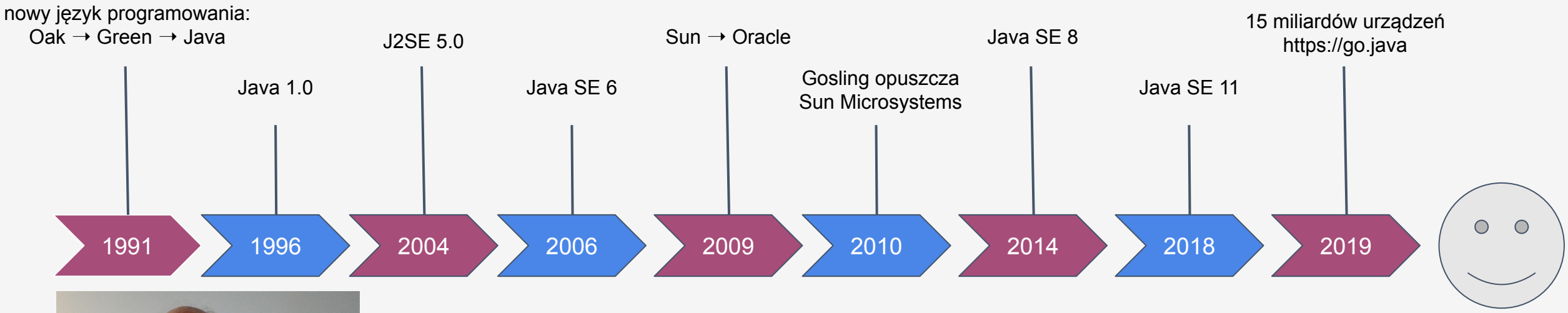
09:00 - powitanie
10:00 - rys historyczny, założenia języka, pierwszy program
10:30 - przerwa krótka
10:40 - konto na GitHub
11:20 - typy danych, operatory, instrukcje
12:40 - przerwa długa
13:00 - typy danych, operatory, instrukcje cd.
14:00 - elementy języka, instrukcje, bloki
14:30 - przerwa krótka
14:40 - elementy języka, instrukcje, bloki cd.



JAVA

trochę historii, podstawowe założenia

Rys historyczny



James Gosling
Father of Java

Sun microsystems

ORACLE®

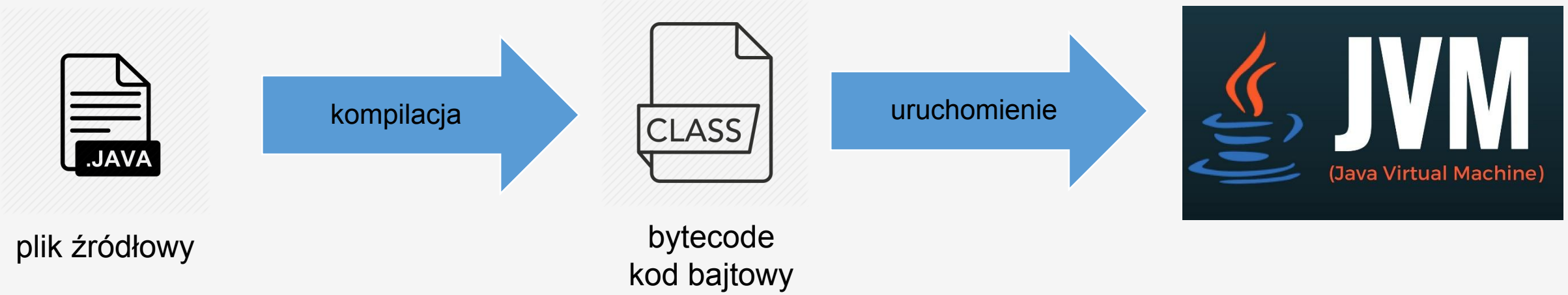
Podstawowe założenia

- **WORA** – **Write Once Run Anywhere** - wieloplatformowość
- sieciowość i obsługa programowania rozproszonego
- prosty, obiektowy język, łatwy do przyswojenia przez programistów
- niezawodność i bezpieczeństwo
- wydajność, wykorzystanie procesorów wielordzeniowych

Programowanie?

Java - język programowania wysokopoziomowego

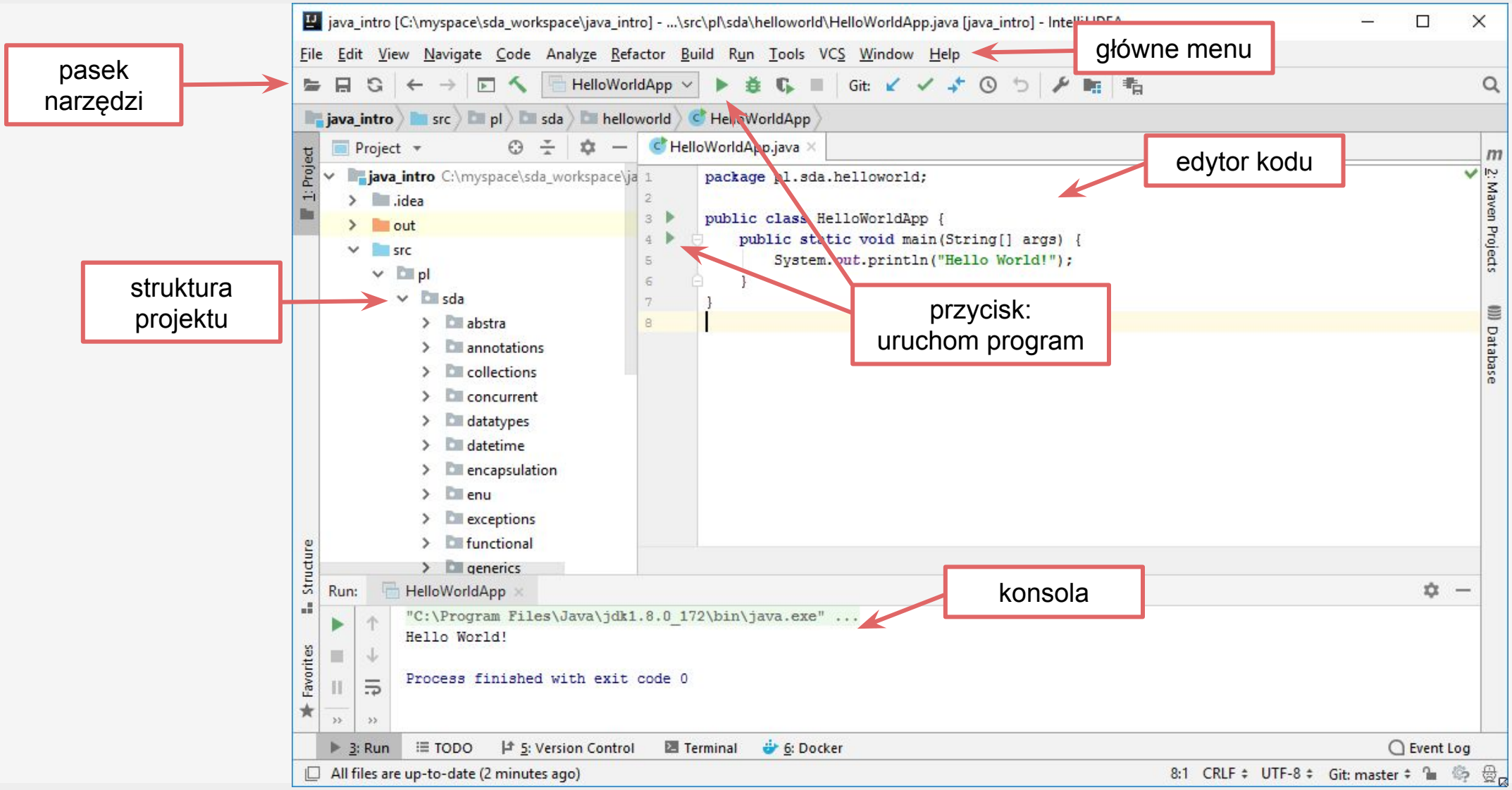
- **kod źródłowy** - kod napisany przez programistę w plikach z rozszerzeniem ***.java*** (np.: HelloWorld.java).
  - jeden plik *.***java** = dokładnie jedna publiczna klasa
  - w pliku mogą znajdować się definicje innych (niepublicznych) klas
  - nazwa pliku = nazwa publicznej klasy zdefiniowanej w pliku (uwaga: wielkość liter ma znaczenie!)
- **bytecode** - kod bajtowy (plik z rozszerzeniem ***.class***)
  - powstaje podczas kompilacji kodu źródłowego
  - jest interpretowany i tłumaczony na konkretne rozkazy procesora przez JVM
- wirtualna maszyna Javy (ang. **Java Virtual Machine**, w skrócie **JVM**) wykonuje bytecode Javy poprzez interpretację / kompilację na kod maszynowy



plik źródłowy

bytecode
kod bajtowy

IntelliJ IDEA - wprowadzenie



Instrukcja tworzenia nowego projektu: http://goo.gl/zdZP2N

Hello World!

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

Instrukcja tworzenia i uruchomienia projektu HelloWorld : https://goo.gl/bMF3Jv

Konwencje nazewnicze

- W czasie kompilacji białe znaki (spacja, tabulacja, koniec wiersza) są pomijane
- Można wobec tego dowolnie "formatować" kod programu, czyli dzielić na wiersze, dodawać dodatkowe spacje...
- Ważne jest żeby zachować przejrzysty **styl programowania** - elementy stylu będziemy poznawać sukcesywnie w trakcie nauki języka
- Na początek pamiętajmy o:

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

definicja klasy w osobnej linii (wiersz kończy otwierający nawias klamrowy)

sygnatura metody w osobnym wierszu + wcięcia

każda instrukcja w osobnym wierszu

zamykające nawiasy klamrowe w nowym wierszu

IntelliJ pomaga nam w zachowaniu przejrzystego stylu programowania:
*{w klasie}(**Ctrl + Alt + L** lub **Menu → Code → Reformat Code**)*

Komentarze w kodzie

```
/**
  * Komentarz dla całej klasy
  */
public class HelloWorldApp {
      public static void main(String[] args) {
            // komentarz do wyjątkowo skomplikowanej linijki kodu
            System.out.println("Hello World!");
      }
      /*
        Komentarz dla metody
       */
      public static Car getCar() {
                return new Car();
      }
}
```

wielowierszowe, dokumentacyjne - teksty umieszczone między znakami: /** i */ - przetwarzane przez oprogramowanie do tworzenia dokumentacji (np.: javadoc), podgląd dokumentacji:

{na klasie lub metodzie}

*(**Ctrl + Q** lub **Menu** → **View** → **Quick Documentation**)*

jednowierszowe - teksty umieszczone po znakach // do końca wiersza

wielowierszowe - teksty umieszczone między znakami: /* i */
Przydają się do zakomentowania kodu (czyli wyłączenia kodu z działania)

Zadania

#helloworld

Zadania

#helloworld

Po każdej zmianie programu należy program skompilować i uruchomić:

*{w klasie}(**Alt + Shift + F10** lub **PPM → Run HelloWorld.main()**)*

1. Zmodyfikuj klasę ***HelloWorldApp***, tak by wyprowadzała na konsolę różne napisy, np.:
   a. Twoje imię i nazwisko
   b. W jednym wierszu napis "Witaj", w następnym imię, itp.

2. Dodaj każdy rodzaj komentarza do swojego kodu

3. Zakomentuj pojedynczą linię kodu (przy pomocy //), potem zakomentuj kilka linii kodu (przy pomocy /*..*/)

Pamiętaj o wykorzystaniu repozytorium kodu Git! * Dla chętnych.

Git + GitHub - wprowadzenie

1. Załóż konto w serwisie:
   GitHub - https://github.com
2. Utwórz repozytorium **Git** i nowy projekt w **Intellij** na bazie linka do utworzonego repozytorium.
3. Wykorzystaj utworzone repozytorium do zapisywania postępów swojej pracy nad kodem.
4. Przykłady kodu + zadania: https://github.com/softwaredevelepmentacademy/java24gda_intro

Instrukcja pobierania (klonowania) projektu ze zdalnego repozytorium : https://goo.gl/kYzbEQ

Dane

typy, literały, zmienne, wyrażenia

Dane i ich typy

Dane

- to wartości (liczby, znaki, napisy itp.) zapisywane w pamięci komputera
- programowanie to głównie przetwarzanie danych
- mają swoje typy
- sposoby przedstawienia w kodzie:
  - literały - napis oznaczający wartość danej, np.: 12, 30.1, 'a', "Ala ma kota"
  - zmienne - symbol w programie oznaczający dane zapisane w pamięci
  - stałe - zmienne które nie mogą zmienić wartości

Typy danych

- zbiór możliwych wartości, np. typ **byte** ma zakres od -128 do 127
- zestaw operacji dozwolonych, np. typy liczbowe można dodawać, odejmować itp.
- rozmiar pamięci do zapisu danej
- Java posiada ścisłą kontrolę typów - każda dana w programie musi mieć typ
- klasy są również typami danych dla obiektów!

Typy prymitywne

nazwa	rozmiar w pamięci (ilość bajtów)	zakres	znaczenie
byte	1	od -128 do 127	liczby całkowite
short	2	od -32768 do 32767	
int	4	od -2 147 483 648 do 2 147 483 647	
long	8	od -9 223 372 036 854 775 808 do 9 223 372 036 854 775 807	
float	4	max około 7 liczb po przecinku	liczby rzeczywiste
double	8	max około 15 liczb po przecinku	
char	2	od 0 do 65556	liczbowe kody dla znaków Unicode
boolean	1	*true*, *false*	wartości logiczne

Literały

Literał
napis reprezentujący w sposób bezpośredni wartość danej

Rodzaje literałów:

- liczbowe
  - całkowite np.: 100, 3, 100_990 (znak '_' jest ignorowany, poprawia czytelność) - typ takiego literału to domyślnie **int**, można zmienić na typ **long** dodając literę: l lub L np.: 100L, 10056l
  - rzeczywiste np.: 1.2, 45_100.9, .15, 5. - typ takiego literału to domyślnie **double,** można zmienić do typu **float** dodając literę: f lub F np.: 100.10f, .156F

- znakowe (typu **char**) zapisujemy jako pojedyncze znaki w apostrofach, np.: 'a', '+' itp. Są to kody (liczby całkowite nieujemne) znaków. Każdy kod oznacza jakiś znak w Unicode: https://pl.wikisource.org/wiki/Unicode/0

- łańcuchowe - napisy, czyli ciągi znaków zapisanych w cudzysłowie, np.: "to jest kurs Java", "dzisiaj świeci słońce". Literały łańcuchowe oznaczają obiekty klasy **String**

- logiczne (typu **boolean**): **true** i **false**

Unicode - https://pl.wikisource.org/wiki/Unicode/0

numer	znak
1 - 32	<control>
33	!
34	“
...	...
48	0
49	1
50	2
...	...
63	?
64	@

numer	znak
65	A
66	B
67	C
68	D
...	...
97	a
98	b
99	c
100	d
...	...

numer	znak
260	Ą
261	ą
262	Ć
263	ć
...	...
377	Ź
378	ź
379	Ż
380	ż
...	...

numer	znak
1044	Д
1045	Е
1046	Ж
1047	З
...	...
1096	ш
1097	щ
1098	ъ
1099	ы
...	...

numer	znak
8721	∑
8723	∓
8730	√
8734	∞
8747	∫
...	...
10226	⟲
10227	⟳
10232	⟸
10233	⟹

Zmienne

Zmienna

symbol w programie oznaczający obszar w pamięci komputera, w którym mogą być zapisywane różne dane

```
int x;
```
← deklaracja zmiennej

```
x = 10;
```
← inicjalizacja / zmiana wartości zmiennej

```
int x = 10;
```
← to samo co wyżej tylko w jednej linijce

```
final int MAX = 10;
```
← deklaracja i inicjalizacja stałej

- zmienne mają typ (zawsze!)
- zmienne mają nazwę (reguły!)
- zmienne mają wartość
- wartość może być wielokrotnie zmieniana
- dodając przed deklaracją słowo **final** tworzymy stałą
- stała może być zainicjalizowana i nie może być zmieniana

Słowa kluczowe i zarezerwowane

Słowa kluczowe

mają one specjalne znaczenie (np. oznaczają instrukcje sterujące) i nie mogą być używane w innych kontekstach poza znaczeniem opisanym przez składnię języka

Słów kluczowych nie można używać m.in.: jako literałów ani nazw zmiennych

Przykłady słów kluczowych:

- class
- return
- byte, int, char, ..
- new
- final
- ...

https://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html

Konwencje nazewnicze

Identyfikator
nazwa zmiennej, stałej, metody lub klasy

Zasady i konwencje tworzenia identyfikatorów:

1. zaczynają się od litery lub podkreślenia ('_') i mogą zawierać dowolny ciąg znaków alfanumerycznych (liter i cyfr) i / lub znaków podkreślenia, np:
   a. prawidłowe nazwy: **count**, **size**, **_element1, car123, objectWidth**
   b. nieprawidłowe nazwy: **1element, object-height, document!size**
   c. wielkość liter ma znaczenie, nazwa **count** i **Count** to dwie różne nazwy

2. nie można używać słów zarezerwowanych, np.: **class, int, final** itp.

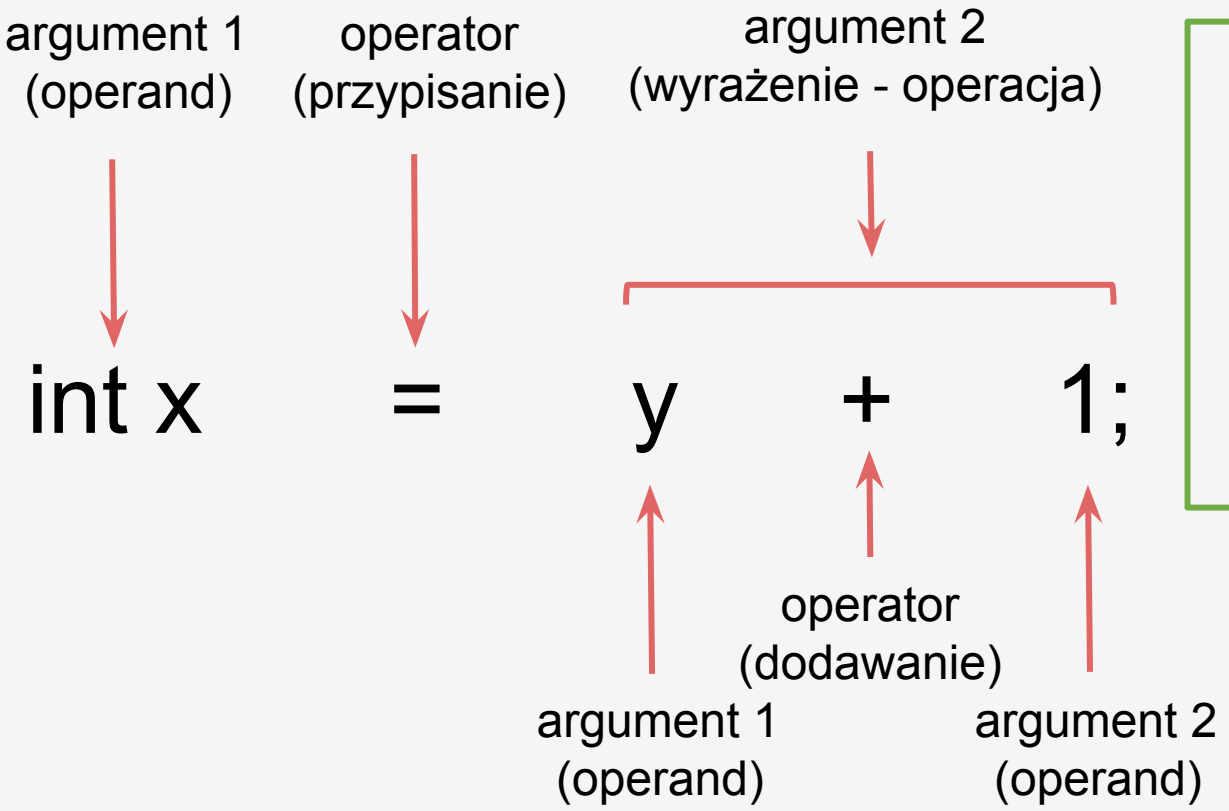
3. nazwy zmiennych i metod zaczynamy małą literą, każdy składnik wyróżniamy, rozpoczynając go wielka literą, np.: **x, price, numOfAllOccurs, getBackground()**

4. nazwy stałych piszemy wielkimi literami, składniki rozróżniając za pomocą podkreślenia, np.: **MAX_SIZE, VAT**

5. nazwy klas zaczynamy wielką literą i dalej wyróżniamy poszczególne składniki także wielką literą, np.: **Car, CarManager, ArrayList** - tzw. **CamelCase**

Operacje na danych

Operacje
działania na danych, które generują nową wartość



Wyrażenia (ang. expressions)
składowa języka, którą budujemy ze zmiennych, stałych, literałów oraz wywołań metod, posługują się operatorami i nawiasami.
Wyrażenia są wyliczane, a ich wyniki mogą być wykorzystywane, np. w przypisaniach, jako argumenty innych operatorów, metod itp

Przykłady wyrażeń i ich wartość (gdy y = 10):
y - wartość zmiennej y, czyli 10
1 - wartość literału, czyli 1
y + 1 - wartość operacji dodawania, czyli 11
x = y + 1 - wartość operacji przypisania, czyli 11

Operatory

Operacje arytmetyczne (numeryczne)

[wynik to zawsze większy z typów!]

- dodawanie (+)
- odejmowanie (-)
- dzielenie (/)
- reszta z dzielenia (%)
- mnożenie (*)
- zwiększanie o 1 (++)
- zmniejszanie o 1 (--)

Przykłady:

x +1, a - b, i++, x + 6.9
x / 5 - dzielenie całkowitoliczbowe
x / 5.0 - dzielenie pełne
x % 3 - tylko dla liczb całkowitych

Operacje relacyjne (numeryczne)

[wynik to zawsze boolean!]

- czy równe (==)
- czy nierówne (!=)
- czy większe(>)
- czy większe lub równe(>=)
- czy mniejsze (<)
- czy mniejsze lub równe (=<)

Przykłady:

x > 10, a != b, count =< 100

Operacje logiczne (boolean)

[wynik to zawsze boolean!]

- logiczna koniunkcja (&&)
- logiczna alternatywa (||)
- logiczne zaprzeczenie (!)

Przykłady:

boolean x = a && b;
hasSize || canMove
!notAMember

Operatory cd

Operacje przypisania:

[wynik to zawsze wynik wyrażenia po prawej stronie]

- przypisanie proste (=)
- przypisanie złożone (zmiana wartości i przypisanie) (op=)
  - zamiast: **x = x op y;**
  - możemy napisać krócej: **x op= y;**
  - gdzie **op** - to operator typu: +, -, *, /, %, a **y** to dowolne wyrażenie

Przykłady:

- przypisanie proste: x = 1, a = b, x = y = z
- przypisanie złożone:
  - zamiast: x = x +1, a = a * 10, z = z /10
  - możemy: x += 1, a *= 10, z /= 10

Właściwości operatorów

```
int x = y  -  1  +   z   *  10;
```

kolejność wykonania (dla z=5 i y=14):

```
int x = y  -  1  +  50;

int x = 13  +  50;

int x  =  63;
```

to samo tylko z nawiasami (czytelniejsze!):

```
int x = (y  -  1)  +   (z   *  10);
```

Priorytety operatorów:

1. jednoargumentowe: !, +, -, --, ++, (typ)
2. arytmetyczne: *, /, %
3. arytmetyczne: +, -
4. relacyjne: <, <=, >, >=
5. relacyjne: ==, !=
6. logiczne: &&
7. logiczne: ||
8. przypisania: = , op=

Wiązania operatorów
(kolejność wykonania gdy priorytet taki sam):

1. **prawe** - przypisania: =, op=
2. **lewe** - pozostałe

Konwersje arytmetyczne

Konwersja (ang. cast)

zmiana typu wyrażenia, np. int → double.

Operator konwersji ma postać: (**typ**) **wyrażenie**, np.: (int) (width * 0.2)

Konwersje automatyczne
[arytmetyczne, rozszerzające]

byte → short → int → long → float → double

char → int → long → float → double

Przykłady:

```
int a = 10;
double d = a;
char c = 'a';
int code = c;
```

Konwersje zawężające
[strata informacji!]

double → float → long → int→ short → byte

Przykład:

```
double d = 10.6;
int a = (int) d; // a = 10
```

Promocja numeryczna
[konwersja automatyczna wartości wyrażeń]

Zasady konwersji dla operatorów dwuargumentowych:

- jeden z argumentów ma typ **double** drugi zmieniany jest do **double**
- jeden z argumentów ma typ **float** drugi zmieniany jest do **float**
- jeden z argumentów ma typ **long** drugi zmieniany jest do **long**
- w przeciwnym razie oba argumenty zmieniane są do typu **int**

Przykład:

```
int a = 10;
double d = a + 5.5;
```

Przykłady w kodzie: pl.sda.datatypes.Operators

Zadania

#datatypes

Zadania

#datatypes

1. Napisz program, który utworzy dwie zmienne, a następnie wypisze na ekran ich sumę, różnicę i iloczyn.
2. Napisz program, który tworzy jedną zmienną, a następnie wypisze na ekran jej wartość podniesioną do 3 potęgi.
3. Napisz program, który utworzy jedną zmienną, a następnie wypisze na ekran tekst: **true** jeżeli wartość tej zmiennej jest liczbą parzystą lub **false** w przeciwnym przypadku.
4. Napisz program, który utworzy jedną zmienną, a następnie wypisze na ekran tekst: **true** jeżeli wartość tej zmiennej jest podzielna przez 3 i jednocześnie przez 5 lub **false** w przeciwnym przypadku.
5. Wyświetl na ekranie pięć pierwszych liter alfabetu: łacińskiego (zaczyna się od kodu: 65), hebrajskiego (zaczyna się od: 1488) i tybetańskiego (zaczyna się od: 3840)
6. Wyświetl na ekranie w jednej linijce znaki (**char**) dla kodów: 74, 65, 86, 65, 32, 8658, 32, 9786
7. W osobnej klasie ***FahrenheitConverter,*** w metodzie *main()* napisz program przekształcający dane o temperaturze podanej w skali Fahrenheit do skali Celsjusza. Dane wejściowe (temperatura w skali Fahrenheit) należy podać w inicjacji odpowiedniej zmiennej w programie. Sprawdź czy program poprawnie oblicza temperatury dla danych: **32 °F = 0 °C**; **212 °F = 100 °C**
8. W osobnej klasie ***ComputerPrice,*** w metodzie *main()* napisz program obliczający cenę komputera na podstawie jego części. Program ma wypisać na konsolę osobno cenę samego komputera: płyta główna, procesor, pamięć RAM, dysk twardy i osobno cenę komputera i monitora. W cenie należy uwzględnić podatek VAT = 23%.

Pamiętaj o wykorzystaniu repozytorium kodu Git!

Elementy języka

instrukcje, bloki

Wyrażenia, instrukcje, bloki

Wyrażenia (ang. expressions)

składowa języka, którą budujemy ze zmiennych, stałych, literałów oraz wywołań metod, posługując się operatorami i nawiasami.
Wyrażenia są wyliczane, a ich wyniki mogą być wykorzystywane, np. w instrukcjach przypisania, jako argumenty innych operatorów, metod itp

Instrukcja (ang. statement)

podstawowy element języka który może być "wykonany". Jest to "motor" działania programu.
Instrukcje kończymy znakiem średnika: ;
Przykłady instrukcji: **x = 10;** **b++;** **new String();** **double d = 10.56;**
To nie są instrukcje: **a + b; // - błąd: Not a statement** **double d;**

Bloki (ang. blocks)

zestaw instrukcji (zero lub więcej), które są zapisane wewnątrz nawiasów klamrowych i które mogą być użyte tam gdzie pojedyncze instrukcje są dozwolone.
Przykłady na następnej stronie.

Instrukcja sterująca - if

Instrukcja sterująca
to instrukcje które umożliwiają zmianę sekwencji (kolejności) wykonywania instrukcji programu.

if (*exp*) *ins1*
else *ins2*

```
if (x > 1000)
    print("Duża")
else
    print("Mała")
```

[print = System.out.println] !

exp - to wyrażenie, którego wynik to zawsze boolean, składniki:

- operatory relacyjne: <, <=, >, >=
  - dwa argumenty
  - wyłącznie typy numeryczne (od **byte** do **double,** również **char**)!
  - wynik to: **true** albo **false**
  - operator przypisania < **priorytet** < operatory arytmetyczne
- operatory równości / nierówności: ==, !=
  - dwa argumenty
  - typu numeryczne, logiczne lub referencyjne (po obu stronach ten sam typ!)
  - operator przypisania < **priorytet** < operatory relacyjne
  - uwaga: '==' to nie to samo co '=' !
- operatory logiczne (niższy priorytet!):
  - **!** - ("nie", jednoargumentowy) logiczne zaprzeczenie, np.: if(!(x>10))...
  - **&&** - ("i")równe true jeśli oba argument jest prawdziwy,np.: if(x>1 && x<10)...
  - **||** - ("lub") równe true jeśli przynajmniej jeden argument jest prawdziwy, np.: if(x<1 || x>10) ...

Instrukcja sterująca - if

jeżeli wyrażenie ***exp*** == true
to wykonają się instrukcje: ins1 i ins2

jeżeli wyrażenie ***exp*** != true
to wykonają się instrukcje: ins3, ins4 i ins5

```
if (exp) {
    ins1;
    ins2;
} else {
    ins3;
    ins4;
    ins5;
}
```

początek bloku 1
(instrukcja grupująca)

początek bloku 2

else - jest opcjonalne!

koniec bloku 1
(brak średnika!)

koniec bloku 2
(brak średnika!)

Przykłady w kodzie: pl.sda.statements.IfStatements

Instrukcja sterująca - switch

Switch

instrukcja wyboru wielowariantowego, w niektórych przypadkach może zastąpić wielokrotnie zagnieżdżone instrukcje if - else if

wyrażenie ***exp*** jest wyliczane i porównywane z wartościami wyrażeń stałych (fexp1, fexp2 itp)

etykiet **case** z wyrażeniami stałymi może być wiele, gdy wartości się zgadzają instrukcje po etykiecie są wykonywane

etykieta **default** (opcjonalna!) oznacza instrukcje wykonywane gdy żadna z etykiet **case** nie pasowała

```
switch (exp) {
   case fexp1: ins1; break;
   ...
   case fexp2: ins2; break;
   ...
   default: ins4
}
```

instrukcja **break** (opcjonalna!) przerywa sekwencję wykonywania instrukcji i przekazuje sterowanie poza **switch**

Instrukcja sterująca - switch

Przykład:

```
int a = 3, b = 4, result = 0;
char op = '+';
switch(exp) {
   case '+': result = a + b; break;
   case '-': result = a - b; break;
   case '*': result = a * b; break;
   case '/': result = a / b; break;
   default: print("Uknown type!");
}
```

W instrukcji **switch** jako wyrażenie wyliczane (*exp*) można użyć:

- typów numerycznych: **char**, **byte**, **short**, **int**
- typów wyliczeniowych (**enum**, o tym później)
- łańcuchów znaków - obiekty klasy **String**

Etykiety **case** są tworzone przez wyrażenia stałe (znane w fazie kompilacji), czyli mogą to być:

- literały, np.: '+', 1, "MAY"
- nazwy stałej (czyli zmienna dozwolonego typu + **final**), np.: LMAX, LMIN
- złożenia powyższych: literały + stałe + operatory, np.: LMAX + 3 / LMIN

Przykłady w kodzie: pl.sda.statements.SwitchStatements

Zadania

#statements

Zadania

#statements

1. Napisz program, który wypisze na ekran konsoli, czy dana liczba całkowita znajduje się w przedziale 1-10, 11-100, 101-1000, 1001-10000, czy też może jest mniejsza lub równa od 0 lub większa od 10000. Parametrem wejściowym niech będzie zmienna zainicjowana na początku programu.

2. Napisz program, który wypisze na ekran konsoli, słowo oznaczające ocenę dla podanej cyfry. Np. dla 1 - "niedostateczny", 2 - "mierny" itp. Obsłuż przypadek gdy cyfra jest poza skalą ocen.

3. Napisz program, który wypisze na ekran konsoli, cyfrę arabską dla podanej liczby rzymskiej (od 1 do 9). Czyli np. dla 'I' wypisze 1, dla 'V' 5 itp. Obsłuż przypadek gdy podana liczba rzymska jest nieprawidłowa.

4. Napisz program, który wypisze na ekran konsoli czy podany kod Unicode jest liczbą (0-9), małą literą (a-z) czy też dużą literą (A-Z). Kody każdej z grup znaków następują po sobie więc wystarczy znaleźć kod np. dla litery 'a' i 'z' i sprawdzić czy podany kod zawiera się w tym przedziale.

5. Napisz program, który dla podanej liczby wypisze na ekran konsoli dzień tygodnia (dla 1 - "poniedziałek", 2 - "wtorek" itp). Dodatkowo wyświetl ile dni zostało do weekendu, dla poniedziałku - 5 dni, wtorku - 4 itp.

6. * Zobacz w jaki sposób można pobrać dane od użytkownika (z konsoli) analizując klasę: pl.sda.statements.ScannerApp. Spróbuj dodać wczytywanie liczb z konsoli do każdego z zadań powyżej.

Pamiętaj o wykorzystaniu repozytorium kodu Git! * Dla chętnych.

Spotkanie #2

Wprowadzenie do języka Java

O mnie

Maciej Rzepiński
maciej.rzepinski@gmail.com
http://mrzepinski.pl
Slack: **@maciej.rzepinski**
LinkedIn: https://www.linkedin.com/in/maciejrzepinski/

O Tobie

- Imię i nazwisko?
- * Twoje doświadczenie z IT?
- * Dlaczego IT i dlaczego JAVA?
- * Czego oczekujesz od kursu?
- * Co chcesz robić po kursie?
- * Jak wyobrażasz sobie pracę programisty?

Kontrakt

1. obecność i punktualność
2. wyciszamy komórki, wychodzimy jak musimy porozmawiać
3. pracujemy w domu - **dużo!**
4. pytamy, pytamy, pytamy - nie ma głupich pytań :)
    a. w czasie zajęć - w grupie lub indywidualnie
    b. slack - kanał **general** lub **direct message**
    c. ankiety
5. szacunek i wzajemna pomoc
6. przerwy
7. zaangażowanie
8. jesteśmy na “Ty”

Sposób pracy

1. teoria + live-coding
   a. śledzimy to co robi prowadzący
   b. nie przepisujemy kodu w tym samym czasie
   c. teoria to tylko wstęp, “zajawka”
2. przykłady i zadania w kodzie dostarczone przez prowadzących
   a. tylko do odczytu
   b. aktualizacja na każdych zajęciach
3. zadania do rozwiązania w czasie zajęć:
   a. samodzielnie przy swoim laptopie
   b. samodzielnie przy laptopie prowadzącego
   c. grupowo (2-3 osoby)
4. zadania domowe
5. ankiety i testy

Polecana literatura

- **Java. Podstawy. Wydanie X**
  Autorzy: Cay S. Horstmann, Gary Cornell
- **Czysty kod. Podręcznik dobrego programisty**
  Autor: Robert C. Martin
- **Thinking in Java. Edycja polska. Wydanie IV**
  Autor: Bruce Eckel

Dodatkowe ćwiczenia

SPOJ - http://pl.spoj.com
Codility - https://codility.com/programmers
HackerRank - https://www.hackerrank.com
Project Euler - https://projecteuler.net

Dodatkowe kursy

https://goo.gl/1Gvb6f - darmowy, polski kurs
https://www.coursera.org
https://www.edx.org
https://www.khanacademy.org
https://www.udemy.com
https://eu.udacity.com
https://www.codecademy.com
https://www.lynda.com
https://teamtreehouse.com
https://www.youtube.com (!)

Szybka powtórka

- typy danych
- literał, zmienna, stała
- operacje
- instrukcje sterujące

OOP

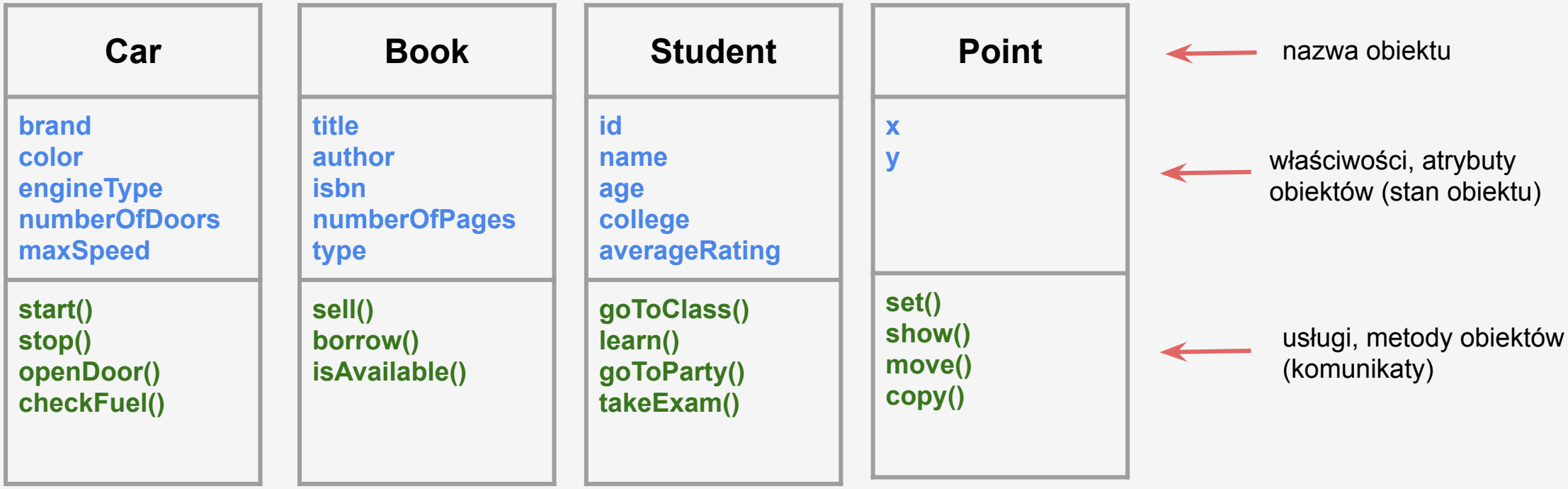
Object Oriented Programming
klasa, obiekt, stan, zachowanie

Co to jest obiekt?

Wszystko jest obiektem!

Programowanie obiektowe zakłada, że cały otaczający nas świat można przedstawić w postaci obiektów, które będą reprezentować uproszczony model świata rzeczywistego

Java - przykłady obiektów



Java - klasy i obiekty

Klasa to wzorzec, szablon zawierający zestaw atrybutów (stan) i metod (komunikacja) opisujących grupę podobnych obiektów.

Obiekt to egzemplarz utworzony z wzorca jakim jest klasa, ograniczony do zestawu atrybutów i metod opisanych w klasie, ale posiadający własne wartości dla atrybutów.

Car (obiekt 1)	Car (obiekt 2)	Car (obiekt 3)
brand = Audi color = White engineType = Diesel numberOfDoors = 3 maxSpeed = 240	brand = Toyota color = Black engineType = Diesel numberOfDoors = 5 maxSpeed = 220	brand = Ford color = Green engineType = Petrol numberOfDoors = 5 maxSpeed = 210
start() stop() openDoor() checkFuel()	start() stop() openDoor() checkFuel()	start() stop() openDoor() checkFuel()

Obiektów jest wiele …

Car
brand color engineType numberOfDoors maxSpeed
start() stop() openDoor() checkFuel()

Klasa jest jedna!

Java - ćwiczenie

Pomyśl o wzorcu, który mógłby opisywać zwierzę i utwórz klasę Animal (na kartce / w notatniku) wraz z odpowiednimi polami i metodami, a następnie podaj przykłady obiektów utworzonych na bazie klasy Animal.

Stan

Stan obiektu

Zestaw atrybutów / cech, które są wspólne dla wszystkich obiektów danej klasy. W Javie atrybuty klasy nazywamy **polami** klasy. Wartości pól są zawarte w obiektach (każdy obiekt ma swoje niezależne wartości!)

[mod_dost] **typ nazwa_pola** [= inicjator]

Przykłady definicji pól:

```
double meanValue; // definicja bez inicjacji
private String name; // definicja bez inicjacji z dostępem: private
public int count = 10; // definicja z inicjacją z dostępem: public
Date today = new Date(); // definicja pola o typie obiektowym
```

[mod_dost] - modyfikator dostępu (opcjonalny) określa "widoczność" pola (o tym później)

typ - określenie typu pola (typ prymitywny lub referencyjny / obiektowy)

nazwa_pola - nazwa pola (patrz reguły tworzenia identyfikatorów)

[= inicjator] - operator przypisania (opcjonalny), który nadaje wartość domyślną. Jeżeli brak, to domyślnie pola (ale nie zmienne lokalne!) będą miały wartość:

- 0 lub 0.0 dla typów numerycznych
- false dla typów logicznych
- null dla typów referencyjnych

Zachowanie

Zachowanie obiektu

Zestaw operacji, które można wykonywać na obiektach klasy. W Javie operacje klasy nazywamy **metodami** klasy. Są one wspólne dla wszystkich obiektów danej klasy.

```
[mod_dost] typ_wyniku nazwa([parametry]) {
    [kod - ciało metody]
}
```

Przykłady definicji (sygnatur) metod:

```
int count() // bez parametrów, zwracamy int
private String show() // bez parametrów z dostępem: private
int add(int x, int y) // 2 parametry wejściowe
public void display(String name) // brak wyniku
```

[mod_dost] - modyfikator dostępu (opcjonalny) określa "widoczność" metody (o tym później)

typ_wyniku - określenie typu wyniku zwracanego przez metodę. Jeżeli metoda nie zwraca wyników piszemy: **void**

nazwa - nazwa metody(patrz reguły tworzenia identyfikatorów)

[parametry] - lista parametrów (rozdzielona przecinkami, opcjonalna) czyli danych wejściowych, które "przekazujemy" metodzie.

Metoda - kod (ciało metody)

```
int sum(int x, int y) {
    int z = x + y;
    return z;
}

void show(String name) {
    if (name == null) {
        return;
    }
    System.out.println(name);
}
```

metoda "otrzymuje" dwa parametry: x i y

metoda zwraca daną zgodną z typem w deklaracji

metoda kończy działanie bez widocznego efektu

metoda kończy działanie gdy przetwarzanie dojdzie do nawiasu zamykającego

1. **kod metody (ciało metody)** - zapisujemy w nawiasach klamrowych (tak jak zwykły blok kodu)
2. między nawiasami umieszczamy zero lub więcej instrukcji tworzących ciało metody
3. metoda kończy swoje działania gdy przepływ programu dojdzie do nawiasu zamykającego albo natrafi na instrukcję **return**
4. **return** - służy do zwracania wartości z metody
5. w metodzie mamy dostęp do **3** rodzajów danych:
   a. parametry przekazane w wywołaniu metody
   b. zmienne lokalne utworzone w kodzie metody
   c. pola obiektu, w ramach którego wykonywana jest metoda

Metody dostępowe

Metody dostępowe (gettery / settery)

Metody służące do ustawiania i pobierania wartości z pól obiektu. Pola są "ukrywane" przed innymi klasami, a dostęp do nich odbywa się przez metody dostępowe. Nie jest to wymóg języka tylko dobra praktyka tworzenia oprogramowania obiektowego.

```
public class Car {
    private String brand;

    public String getBrand() {
        return brand;
    }

    public void setBrand(String brand) {
        this.brand = brand;
    }
}
```

pole brand ma dostęp: prywatny, czyli jest ukryte przez kodem spoza klasy Car

getter dla pola brand

setter dla pola brand

parametr brand "przysłania" pole obiektu o tej samej nazwie

żeby odwołać się do pola musimy użyć słowa kluczowego: **this**

Automatyczne tworzenie getterów / setterów w IntelliJ:

{w klasie}
(Alt + Insert → Getter and Setter)
lub
(PPM → Generate → Getter and Setter)

Konstruktor

Konstruktor
"specjalna" metoda w klasie która służy (głównie) do inicjowania pól obiektu

```
[mod_dost] nazwaKlasy([parametry]) {
    [ciało konstruktora]
}
```

```
public class Car {
    private String brand;
    private String color;

    public Car() {}

    public Car(String brand, String color) {
        this.brand = brand;
        this.color = color;
    }
}
```

konstruktor nie zwraca żadnej wartości (nawet **void**!)

konstruktor musi mieć taką samą nazwę jak klasa; może nie posiadać parametrów

konstruktorów może być wiele (albo żadnego); muszą się różnić parametrami

[mod_dost] - modyfikator dostępu (opcjonalny) określa "widoczność" konstruktora (o tym później)

nazwaKlasy - konstruktor musi mieć taką samą nazwę jak klasa do której należy

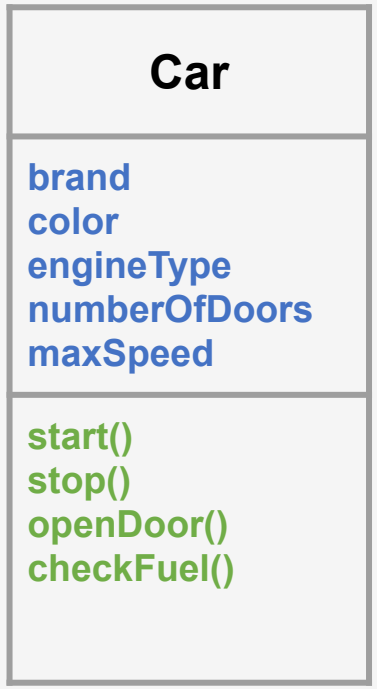
[parametry] - lista parametrów (rozdzielona przecinkami, opcjonalna)

[ciało konstruktora] - zestaw instrukcji, które inicjują pola obiektów i wykonują inne operacje potrzebne do utworzenia nowego obiektu tej klasy

liczebność - klasa może mieć od zera do wielu jawnie utworzonych konstruktorów

Istnieje konstruktor domyślny!

Kod klasy



nazwa

właściwości, atrybuty obiektów (stan obiektu)

usługi, metody obiektów (komunikaty)

```
public class Car {
    private String brand;
    private String color;
    private String engineType;
    private int numberOfDoors;
    private int maxSpeed;

    public Car(String brand) {
        this.brand = brand;
    }
    public String getBrand() {...}
    public void start() {...}
    public void stop() {...}
    public void openDoor() {...}
    public int checkFuel() {...}
}
```

nazwa klasy (patrz reguły tworzenia identyfikatorów) musi być taka sama jak nazwa pliku w którym jest zapisana

pola klasy

konstruktor klasy

metody klasy

Car.java

Obiekty - tworzenie i używanie

```
public class Car {
    private String brand;
    private String color;
    private int speed;

    public Car(String brand, String color) {
        this.brand = brand;
        this.color = color;
    }

    public String getBrand() {...}
    public String getColor() {...}
    public void start() {...}
    public void setSpeed(int speed) {...}
    public int getSpeed() {...}
    public void stop() {...}
}
```

nowe obiekty tworzymy przez użycie słowa kluczowego new i podanie konstruktora z parametrami

```
Car myCar = new Car("Ford", "Black");
Car neighborsCar = new Car("Skoda", "Red");
```

tworzymy 2 obiekty tej samej klasy, z różnymi parametrami, przypisane do zmiennych myCar i neighborsCar

dostęp do metod i pól obiektu (o ile pozwala na to modyfikator dostępu) uzyskujemy przez znak kropki, tzw.: selektor

```
myCar.start();
neighborsCar.start();
```

wysyłamy polecenie 'start()' do obiektu oznaczonego przez zmienną myCar

metody na obiektach możemy wywoływać wielokrotnie, zmieniają one tylko stan obiektu na rzecz którego są wywołane (poprzez selektor)

```
myCar.setSpeed(50);
neighborsCar.setSpeed(55);
myCar.setSpeed(70);
neighborsCar.setSpeed(75);
myCar.setSpeed(180);
neighborsCar.stop();
```

w nawiasach okrągłych podajemy parametry, które będę przekazane do metody

Zadania

#oop

Zadania

#oop

1. Utwórz klasę Car z polami: brand i color
   *{na katalogu}(**Alt + Insert** → **Java Class** lub **PPM** → **New** → **Java Class**)*

2. Dodaj konstruktor i oba pola jako argumenty
   *{w klasie}(**Alt + Insert** → **Constructor** lub **PPM** → **Generate** → **Constructor**)*

3. Dodaj metodę toString() i dodaj oba pola
   *{w klasie}(**Alt + Insert** → **toString()** lub **PPM** → **Generate** → **toString()**)*

4. W klasie HelloWorldApp utwórz nowy obiekt na podstawie klasy Car, np.

```
Car toyota = new Car("Toyota", "black");
```

5. Wypisz na konsoli informację o samochodzie

```
System.out.println("My car is: " + toyota);
```

6. Uruchom program
   *{w klasie}(**Alt + Shift + F10** lub **PPM** → **Run Car.main()**)*

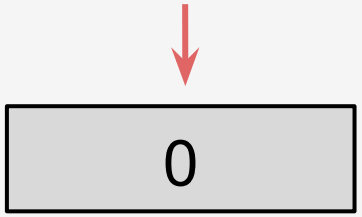
Typ referencyjny

Referencja
wartość, która oznacza lokalizację (adres) obiektu w pamięci

Typ prymitywny (numeryczny lub logiczny)

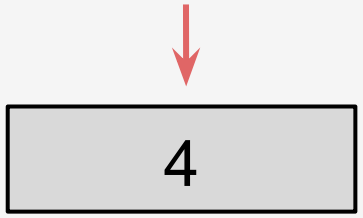
Deklaracja zmiennej **x** wydziela pamięć (1 bajt) do przechowywania liczby

```
byte x;
```



Przypisanie wartości zmiennej **x** zmienia obszar pamięci do niej przydzielonej

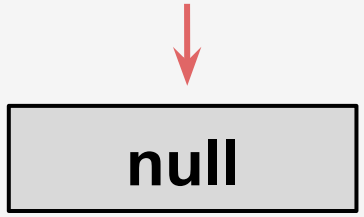
```
x = 4;
```



Typ referencyjny

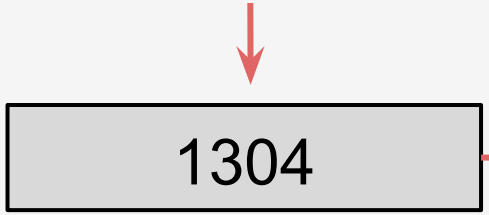
Deklaracja zmiennej **myCar** nie tworzy obiektu, tylko referencję (wydziela pamięć do jej przechowywania)

```
Car myCar;
```



Dopiero zastosowanie wyrażenia **new** tworzy nowy obiekt w pamięci i przydziela adres do obiektu

```
myCar = new Car();
```



Obszar pamięci przechowujący obiekty nazywa się: **sterta** (ang. **heap**)

adres	wartość
1296	...
1300	...
1304	"Ford"
1308	"Black"
1312	....
1316	....

Typ referencyjny

- każda zmienna deklarowana z nazwą klasy w miejscu nazwy typu, to zmienna **typu referencyjnego**, np:
  String message; Car newCar; Date today;
  - mogą one:
    - zawierać referencje (adres / odnośnik) do obiektów (referencja to nie to samo co obiekt!)
    - nie zawierać żadnej referencji (nie wskazywać na żaden obiekt) - wartość jest równa **null**

- wartości typów referencyjnych to liczby (adresy to liczby) lub wartość **null**
  (**null** to literał typu referencyjnego, tak jak 1 jest literałem typu **int**)

- dla wartości typów referencyjnych (mimo że są liczbami) niedopuszczalne są operacje arytmetyczne

- dopuszczalne są natomiast operacje:
  - porównanie referencji (**==**) / (**!=**), np.: myCar **==** yourCar; message1 **!=** message2
  - przypisanie wartości innej referencji lub "resetowanie" przez ustawienie wartości **null**, np.: myCar **= null**

- do przeprowadzenia operacji na obiekcie na który wskazuje referencja służy selektor (znak kropki), np.: myCar.start() - uwaga jeżeli referencja ma wartość **null**, to w czasie wykonania programu wystąpi błąd, a konkretnie wyjątek: **NullPointerException**

Przykłady w kodzie: pl.sda.oop.PrimitivesVsReferences

Metody - przekazywanie przez wartość

W Javie argumenty przekazywane są metodom wyłącznie przez wartość. Czyli w samej metodzie odwołujemy się do kopii a nie do faktycznego argumentu.

```
int x = 10;
increment(x);

System.out.println(x);
…
…
…
...
void increment(int y) {

    ++y;

}
```

deklaracja i inicjalizacja zmiennej **x**, **x** ma wartość 10

przekazujemy zmienną x jako argument do metody

x ma nadal wartość 10!

parametr metody **y** jest kopią zmiennej **x** i ma wartość 10

po tej operacji zmieni się wartość **y**, teraz **y** = 11.
x pozostaje niezmieniony

```
Car car1 = new Car("Audi", "Black");
Car car2 = new Car("Toyota", "White");

switchCars(car1, car2);
System.out.println(car1.getBrand()); // Audi
System.out.println(car2.getBrand()); // Ford
…
…
void switchCars(Car car1, Car car2) {
        Car tmp = car1;
        car1 = car2;
        car2 = tmp;
        car1.setBrand("Ford");
}
```

tu są tworzone nowe referencje do których kopiowane są adresy obiektów przekazanych w wywołaniu metody. Obiekty na które wskazują parametry są te same więc można je (obiekty) zmienić, np.: przez settery

Klasy opakowujące

Klasa opakowująca (ang. wrapper)
klasa, która opakowuje wartości typów pierwotnych (int, char, double) czyniąc z nich zwykłe obiekty.

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

```
Integer a = new Integer(5);
int b = a.intValue();

int c = a;
Integer d = 10;

Double d = Double.parseDouble("2.5");
double e = d.doubleValue();

double f = d;
d = 10.1;
```

wartość 5 jest opakowana w obiekt **Integer**. Instrukcja *new* tworzy nowy obiekt.

z obiektu o typie **Integer** wyciągamy wartość jak typ prymitywny **int**

auto - unboxing

auto - boxing

metody *parseXXX*() konwertują napisy do odpowiednich wartości liczbowych

auto - unboxing

auto - boxing

Przykłady w kodzie: pl.sda.datatypes.Wrappers

Zadania

#oop

Zadania

#oop

1. Utwórz klasę **Triangle** i napisz metodę isRectangular(), która jako argumenty przyjmować będzie trzy liczby całkowite. Metoda powinna zwrócić true jeśli z odcinków o długości przekazanych w argumentach można zbudować trójkąt prostokątny. Wzór który może pomóc: $c^2 = a^2 + b^2$

2. Przenieś kod zapisany w metodach main() klas **FahrenheitConverter** i **ComputerPrice** do osobnych metod, np.: convertToCelsius(double temperatureInFahrenheit), getComputerPrice()

3. W klasie **FahrenheitConverter** dodaj metodę, która konwertuje temperatury w drugą stronę (Celsjusz → Fahrenheit)

4. W klasie **ComputerPrice** wydziel metody getComputerPrice(), getMonitorPrice() i getComputerAndMonitorPrice(), ostatnia z metod ma korzystać z dwóch pierwszych. Zmienną VAT ustaw jako pole klasy **ComputerPrice**.

5. * Utwórz nową klasę **Temperature**, która będzie posiadała pola: double temperature, String date, String hour. Klasa określa temperaturę w skali Celsjusza w konkretnym dniu i o konkretnej godzinie. Dodaj konstruktor inicjalizujący wszystkie trzy pola, metody-gettery dla każdego pola + dodaj metodę show() która będzie zwracała napis w postaci: {date} {hour} - {temperature} °C, np: 2018-10-01 10:45 - 13 °C

6. * Dodaj do klasy **Temperature** metodę showInFahrenheit() która zwróci taki sam napis jak wyżej tylko w skali Fahrenheit. Do konwersji temperatur użyj klasy **FahrenheitConverter.**

Pamiętaj o wykorzystaniu repozytorium kodu Git! * Dla chętnych.

JavaFx - Hello World!

```
public void start(Stage primaryStage) {
        Label label = new Label("Hello World from JavaFX!");

        Button button = new Button("Click me!");
        button.setOnAction(e -> System.out.println("Button was clicked!"));

        VBox box = new VBox();
        box.setAlignment(Pos.CENTER);
        box.getChildren().addAll(label, button);

        primaryStage.setTitle("Hello World - JavaFX");
        primaryStage.setScene(new Scene(box, 300, 200));
        primaryStage.show();
}
```

Zadania

#helloworld

Po każdej zmianie programu należy program skompilować i uruchomić:

*{w klasie}(**Alt + Shift + F10** lub **PPM → Run HelloWorldFx.main()**)*

1. Zmodyfikuj klasę HelloWorldFx:
   a. Zmień tekst przycisku
   b. Zmień tekst dla obiektu typu Label

2. Dodaj do klasy HelloWorldFx kontrolki typu:
   a. TextField
   b. Button
   c. Label

3. * Spraw by po kliknięciu przycisku dodanego w pkt. 2 tekst wpisany do kontrolki typu TextField został skopiowany do kontrolki typu Label.

Pamiętaj o wykorzystaniu repozytorium kodu Git! * Dla chętnych.

Spotkanie #3

Wprowadzenie do języka Java

Rozkład jazdy

Zanim zaczniemy proszę zaktualizować projekt: **java24gda_intro !**

09:00 - powtórka
10:00 - pakiety, modyfikatory dostępu, hermetyzacja
10:30 - przerwa krótka
10:40 - pakiety, modyfikatory dostępu, hermetyzacja cd.
12:00 - klasa String
12:40 - przerwa długa
13:00 - klasa String cd.
14:00 - pętle
14:30 - przerwa krótka
14:40 - pętle cd.
16:00 - koniec zajęć :)

Instrukcja aktualizacji projektu z GitHub: https://goo.gl/m3BHMr

Krótka powtórka

1. programowanie w Javie polega głównie na posługiwaniu się obiektami
2. przed użyciem obiekty muszą być tworzone
3. do tworzenia obiektów służy wyrażenie **new**
4. po słowie **new** podajemy odwołanie do konstruktora z odpowiednimi argumentami
5. wyrażenie **new** zwraca referencję (odniesienie, swoisty adres) do nowo utworzonego obiektu
6. posługiwanie się obiektami w Javie polega wyłącznie na operowaniu na referencjach

Instrukcja tworzenia nowej klasy: https://goo.gl/NSmFTP

Zadanie na rozgrzewkę

1. Tworzymy model danych dla drzewa genealogicznego
2. Przyjmujemy prosty model rodziny: 2 rodziców + 1 dziecko
3. Pojedyncza osoba powinna mieć dane: imię, nazwisko, wiek
4. Pojedyncza rodzina powinna zawierać: wszystkich członków rodziny (jako osobne pola dla każdej z osób: mąż, żona, dziecko) oraz nazwę całej rodziny, np.: “Rodzina Kowalskich”
5. Dodatkowo obiekt rodziny powinien mieć:
   a. metodę, która zwróci opis całej rodziny jako String
   b. *metodę, która zwróci sumę lat wszystkich członków rodziny
   c. *metodę, która zwróci średnią arytmetyczną wieku członków rodziny
6. W osobnej klasie FamilyTest tworzymy 2-3 rodziny i wypisujemy informacje o nich na ekran
7. *Dodaj klasę, która będzie reprezentować małżeństwo. Pola w klasie: mąż, żona, data ślubu. Użyj tej klasy w klasie rodziny.

* Dla chętnych.

Pakiety, modyfikatory dostępu, hermetyzacja

Pakiety

Pakiety

- grupują klasy posiadające wspólne cechy w strukturę podobną do drzewa katalogów na dysku
- zapewniają unikalne nazwy dla klas zapobiegając konfliktom nazw

pierwsza linia w pliku z klasą powinna zaczynać się od deklaracji pakietu. Brak deklaracji == domyślny pakiet

```
package pl.sda.carstore;
```

package to słowo kluczowe które można wykorzystać tylko w jednym miejscu

Pakiety mają strukturę hierarchiczną:

- kropki służą do oddzielania poziomów
- struktura powinna być odzwierciedlona w strukturze katalogów na dysku

Konwencje nazewnicze:

- używamy małych liter, cyfr, kropek, znaków podkreślenia
- o ile to możliwe stosujemy odwrócony adres internetowy jako prefix
- pakiet o nazwie: **java** jest zastrzeżony dla klas wbudowanych Javy

Nazwa kwalifikowana

(pakiet + nazwa klasy) - np.: **pl.sda.carstore.Car**

"pełna" nazwa klasy, która identyfikuje ją w sposób unikalny (swoisty "adres" klasy). Dzięki niej **JVM** "potrafi" odnaleźć naszą klasę (zamiast na przykład klasy **com.ford.Car**) - dwie klasy o nazwie **Car**, ale innym pakiecie mogą istnieć w tym samym programie bez konfliktu

Pakiety - przykład

```
package pl.sda.carstore;

import pl.sda.carstore.details.CarType;

public class Car {
    private String brand;

    private Person owner;

    private CarType type;

    private pl.sda.utils.Color color;
}
```

deklaracja pakietu w pierwszej linii kodu

import klasy pozwala używać w kodzie samej "krótkiej" nazwy, **import** to słowo kluczowe!

klasa **String** znajduje się w pakiecie **java.lang**, który jest domyślnie importowany do każdej klasy

klasa **pl.sda.carstore.Person** znajduje się w tym samym pakiecie co **Car** - nie musimy jej importować żeby użyć krótkiej nazwy

dzięki temu że klas CarType został "zaimportowana" wcześnie nie musimy używać kwalifikowanej nazwy

tutaj import nie został użyty więc podajemy pełną nazwę klasy razem z pakietem

Modyfikatory dostępu

Modyfikatory dostępu
regulują dostęp do klas głównych oraz składowych klasy (pól, metod) oraz klas wewnętrznych.

Na poziomie składowych klasy wyróżniamy cztery modyfikatory:

private	prywatny, dostępny tylko w danej klasie	`private int width = 10;`
[brak]	przyjazny (pakietowy), dostępny tylko dla klas w danym pakiecie	`String message = "Hello";`
protected	chroniony, dostęp z danej klasy, wszystkich klas dziedziczących oraz klas z danego pakietu	`protected double mean = 2.5;`
public	publiczna, dostęp z każdego miejsca	`public char firstLetter = 'a';`

Na poziomie klas głównych można użyć tylko dwóch modyfikatorów: **public** albo **pakietowego**

Hermetyzacja

Hermetyzacja / Enkapsulacja (ang. encapsulation)
to ukrywanie szczegółów implementacji (danych i metod) wewnątrz klasy, tak aby z zewnątrz klasy było dostępne tylko to, co użytkownikowi będzie potrzebne do pracy z obiektem (publiczne API)

Środki do osiągnięcia takiego celu:

- dobry projekt klasy, który oddziela metody służące do komunikacji ze "światem zewnętrznym" od metod "roboczych" potrzebnych do prawidłowego działania programu. Tutaj dobra zasada to: im mniej - tym lepiej

- ukrywanie stanu obiektu (pól obiektu) przed dostępem z zewnątrz. Taki dostęp powinien być możliwe przez publiczne metody. Wtedy mamy kontrolę nad zmianami w obiekcie

- zastosowanie modyfikatorów dostępu do ukrywania albo wystawiania składowych klasy

Zalety takiego podejścia:

- większa odporność programu na błędy, zwiększenie bezpieczeństwa programu poprzez kontrolę nad stanem obiektu (np. poprzez sprawdzanie czy wartości które nadajemy polom obiektu spełniają wymagania utworzone przez twórcę klasy)

- łatwiejsze zmiany definicji klas w miarę rozwoju programu (**refactoring**) - zmiana implementacji bez zmiany interfejsu publicznego

Hermetyzacja - przykład

```
public class Product {
    private final double NO_DISCOUNT = 1.0;
    private final double SATURDAY_DISCOUNT = 0.8;
    private String name;
    private double price;

    public Product (String name, double price) {
      if (name == null) {
          // obsługa błędnego parametru
      }
      if (price <= 0) {
          // obsługa błędnego parametru
      }
      this.name = name;
      this.price = price;
    }
  ……
```

wszystkie pola (zmienne i stałe) są "ukryte" przed dostępem z zewnątrz. Nikt poza twórcą klasy nie może zmienić, np. sobotniej zniżki. Dlatego używamy tutaj modyfikatora: **private**

jedyna możliwość ustawienia nazwy i ceny produktu to konstruktor. Tutaj też następuje kontrola danych. Złe dane nie przejdą :)!

dopiero po sprawdzeniu poprawności danych wejściowych przypisujemy je do pól obiektu

Hermetyzacja - przykład

```
    ....
    public String getName() {
       return name;
    }
    public int getPrice() {
       double discount = getDiscount();
       return price * discount;
    }
    private double getDiscount() {
          if (todayIsSaturday()) {
             return SATURDAY_DISCOUNT;
       }
       return NO_DISCOUNT;
    }
   private boolean todayIsSaturday() {
         // kod sprawdzający datę
    }
}
```

publiczne API stanowią dwie metody: *getName()* i *getPrice()*. Dlatego używamy tutaj modyfikatora: **public**

metody: *getDiscount()* i *todayIsSaturday()* to "robocze" metody, stanowiące szczegóły implementacyjne. Dlatego używamy tutaj modyfikatora: **private.** Dzięki temu że są ukryte wraz z polami stałymi jeżeli będziemy chcieli zmienić sposób wyliczania zniżek (np. skorzystać z zewnętrznego serwisu) bez problemu możemy to zrobić, nie martwiąc się tym że popsujemy komuś kod który korzysta z klasy **Product**

Przykłady w kodzie: pl.sda.encapsulation.Product, pl.sda.encapsulation.Vehicle

Zadania

#encapsulation

Zadania

#encapsulation

1. Umieść klasy które utworzyłeś do tej pory w swoim projekcie w osobnych pakietach tak żeby rozgraniczyć poszczególne bloki / zagadnienia nauki programowania
2. Dodaj do klas utworzonych w zadaniu na początku zajęć (drzewo genealogiczne) odpowiednie modyfikatory dostępu na poziomie pól, konstruktorów i metod.
3. W repozytorium kodu, w pakiecie **encapsulation** znajduje się kolejny pakiet nazwany **task**, gdzie jest kilka klas, które należy uporządkować i stworzyć dla nich odpowiednie pakiety. To Twoje zadanie. Działaj wyłącznie w obrębie pakietu **task**.
4. * W klasach, które właśnie zostały uporządkowane ktoś popełnił błędy i nie zadbał o prawidłową hermetyzację danych oraz modyfikatory dostępu. Przejrzyj klasy i postaraj się by kod był zgodny z tym co było powiedziane na zajęciach - przeprojektuj klasy by spełniały zasady hermetyzacji, a przy tym udostępniały do "świata zewnętrznego" tylko potrzebne API. Zadbaj o prawidłowe użycie danych, dodaj konstruktory gdzie trzeba. Utwórz przykłady użycia (tworzenie produktów i użytkowników) w klasie ***OnlineShop***. Sprawdź, czy Twój kod jest odporny na błędy.
5. * Utwórz klasę koszyka, która umożliwi robienie zakupów: użytkownik dodaje wybrane przedmioty w zadanej ilości do swojego koszyka, a następnie dokonuje zakupu. Wykonywane operacje powinny być wyświetlane na konsoli. Zadbaj o odpowiednią hermetyzację swojego API oraz właściwe modyfikatory dostępu.

Pamiętaj o wykorzystaniu repozytorium kodu Git! * Dla chętnych.

klasa String

Klasa String

String

obiekt klasy java.lang.String reprezentujący łańcuch znaków (tablicę znaków). Obiekt jest niezmienny, raz utworzony nie może zmienić stanu. String jest jedną z najczęściej używanych obiektów w Javie dlatego posiada kilka wyjątkowych właściwości: własny literał, rozszerzony operator ‘+’, string pool

```
String s1 = "Hello World!"; // literal

String s2 = new String("Hello World!");
```

Klasa String
konkatenacja (concatenation)

- Jeśli występują tylko liczby, to mamy do czynienia z **działaniem matematycznym**.

```
System.out.println(1 + 2 + 3);
// 6 (int)
```

- Jeśli jednym z czynników jest String, to mamy do czynienia z konkatenacją.
- Wyrażenie jest wykonywane **od lewej do prawej**.

```
System.out.println(1 + 2 + "3");
// 33 (String)
```

Klasa String
konkatenacja (concatenation)

- każdy obiekt może być argumentem operatora konkatenacji - jest to duże ułatwienie przy wyświetlaniu danych
- dzieje się tak dzięki metodzie toString()

```
LocalDate today = LocalDate.now();
Person person = new Person("Jan", "Kowalski");
System.out.println("Witaj: " + person + ", dzisiaj jest: " + today);
```

Klasa String

== vs equal

```
String s1 = "Hello World!";
String s2 = new String("Hello World!");

System.out.println(s1 == s2);
// false
System.out.println(s1.equals(s2));
// true
```

Klasa String

ważne metody

- length(): int; zwraca długość łańcucha znaków
- charAt(): char; zwraca znak na danej pozycji
- indexOf(): int; zwraca pierwsze wystąpienie znaku
- substring(): String; zwraca podciąg łańcucha znaków
- toLowerCase(): String; zamiana na małe znaki
- toUpperCase(): String; zamiana na wielkie znaki
- equals(): boolean; czy równe
- equalsIgnoreCase(); boolean; czy równe bez względu na wielkość znaków
- startsWith(): boolean; czy łańcuch zaczyna się od ...
- endsWith(): boolean; czy łańcuch kończy się na ...
- concat(): String; dodaje podany String na koniec aktualnego
- contains(): boolean; czy String zawiera podany tekst
- replace(): String; zamiana podciągu na inny
- trim(): String; obcięcie białych znaków z obu stron łańcucha

Klasa String

niezmienność (immutability)

Klasa String jest **final** i nie posiada **settera**, co oznacza, że nie da się zmienić raz utworzonego obiektu typu String.

Wszystkie operacje wykonywane na klasie String powodują utworzenie nowego obiektu typu String.

```
String s4 = "Hello";
System.out.println(s4.concat(" World!"));
// Hello World!
System.out.println(s4);
// Hello
```

Klasa String
łączenie metod

Metody możemy wykonywać jedna po drugiej, co nazywa się łączeniem metod (method chaining).

```
String s = " A long time ago in a galaxy far, far away ";
System.out.println(s
  .trim()
  .replace("galaxy", "Poland")
  .toUpperCase()
);
// A LONG TIME AGO IN A POLAND FAR, FAR AWAY
```

Klasa String

StringBuilder

```
String s = "1";
s = s + "2";
System.out.println(s);
```

=

```
String s = new StringBuilder("1").append("2").toString();
System.out.println(s);
```

Klasa String

StringBuffer, StringBuilder

StringBuffer

stara klasa (poprzednik StringBuilder), która napisana została z myślą o wielowątkowości (metody są synchronizowane).

StringBuilder

nowa klasa (wprowadzona w **Java 5**), która miała zastąpić klasę StringBuffer. Jest szybsza niż StringBuffer, ale nie zapewnia odpowiedniej obsługi w aplikacjach wielowątkowych.

```
StringBuilder s12 = new StringBuilder("123");
System.out.println(s12.reverse().toString());
// 321
```

Przykłady w kodzie: pl.sda.strings.StringExamples

Zadania

#strings

Zadania

#strings

1. Napisz klasę, która wykorzysta większość z metod dostępnych w klasie String.
2. Napisz metodę, która zwróci tekst: “Simon says: [*{text}*]”, gdzie *{text}* - to argument metody. Użyj konkatenacji lub StringBuildera.
3. Napisz metodę, która jako argument otrzyma jedną zmienną typu String, usunie z niej białe znaki z początku i końca tekstu oraz zamieni wszystkie litery na małe.
4. Dodaj do klas reprezentujących osobę i rodzinę utworzonych w zadaniu na początku zajęć metody *toString(),* które w czytelny sposób wyświetlą informacje o obiekcie.
5. Napisz metodę, która jako argumenty będzie przyjmować dwie zmienne typu String i zwróci true jeżeli oba teksty zaczynają się od tego samego znaku.
6. Napisz metodę, która jako argumenty będzie przyjmować dwie zmienne typu String i zwróci true jeżeli 3 ostatnie znaki w obu tekstach są takie same.
7. * W ramach zadania nr 4 użyj StringBuildera do tworzenia wersji tekstowej obiektów.
8. * Napisz metodę sprawdzającą, czy dany łańcuch zawiera co najmniej trzy razy słowo “nie”.

Pamiętaj o wykorzystaniu repozytorium kodu Git! * Dla chętnych.

Pętle
while, do .. while, for

Pętle iteracyjne

Pętla iteracyjna
konstrukcja pozwalająca powtarzać instrukcję (lub grupę instrukcji) określoną liczbę razy.
Każde powtórzenie instrukcji lub grupy instrukcji nazywa się **iteracją**.

Trzy rodzaje pętli w Javie:

while(*exp*) *ins;*

do *ins;* **while**(*exp*);

for(*init; exp; upd*) *ins;*

init - inicjalizacja jednej lub wielu zmiennych (zwykle liczników)

exp - wyrażenie logiczne (zwraca true / false)

upd - wyrażenie lub lista wyrażeń (zwykle zmieniających licznik)

ins - instrukcje do wykonania

Pętla while

Działanie pętli **while**:

1. wyrażenie *exp* jest wyliczane
2. jeżeli zwróci **true** wykonywana jest grupa instrukcji *ins*
3. w przeciwnym razie pętla **while** kończy działanie (sterowanie przechodzi do pierwszej instrukcji po pętli)
4. po zakończeniu wykonywania grupy instrukcji *ins* wracamy do punktu 1 i zaczynamy kolejną iterację
5. **pętla** kończy się gdy:
   a. wyrażenie *exp* zwróci **false**
   b. w instrukcji *ins* zawarto instrukcję **break** albo **return**
6. **iteracja** kończy się gdy w instrukcji *ins* zawarto instrukcję **continue** (sterowanie przechodzi na początek pętli do punktu 1)

while(*exp*) *ins*

```
char c = 'a';
while(c <= 'z') {
  System.out.print(c);
  c++;
}
```

Pętla do-while

Działanie pętli **do-while** jest bardzo podobne do pętli **while** z drobną różnicą:

1. najpierw wykonywana jest grupa instrukcji *ins*
2. dopiero później wyrażenie *exp* jest wyliczane
3. jeżeli zwróci **true** pętla zaczyna się od nowa
4. w przeciwnym razie pętla kończy działanie

do *ins* **while**(*exp*);

```
int i = 1;
do {
    System.out.println(i);
 } while(++i <= 10);
```

Pętla **while** może nie wykonać się ani razu.

Pętla **do-while** wykona się co najmniej raz.

Przykłady w kodzie: pl.sda.loops.WhileLoop

Pętla for

Działanie pętli **for**:

1. najpierw wykonywana jest inicjalizacja zmiennych w sekcji *init*
2. wyliczane jest wyrażenie *exp*
3. jeżeli zwróci **false** pętla kończy działanie, a w przeciwnym razie działanie jest kontynuowane
4. wykonywana jest grupa instrukcji *ins*
5. obliczane są wyrażenia w sekcji *upd*
6. działanie jest wznawiane od punktu 2
7. **pętla** kończy się gdy:
   a. wyrażenie *exp* zwróci **false**
   b. w instrukcji *ins* zawarto instrukcję **break** albo **return**
8. **iteracja** kończy się gdy w instrukcji *ins* zawarto instrukcję **continue** (sterowanie przechodzi na początek pętli do punktu 1)

for(*init ; exp ; upd*) *ins*

Przykład:

```
int n = ...;
int sum = 0;
for (int i = 1; i <= n; i++) {
    sum += i;
 }
```

Odpowiednik z pętlą **while**:

```
init;
while (exp) {
     ins;
     upd;
}
```

Przykłady w kodzie: pl.sda.loops.ForLoop

Pętle - ćwiczenia - Której pętli użyć?

1. **Przejdź 5 kroków do przodu, dostępne operacje:**
   a. zrób krok
2. **Wyjmij wszystkie jabłka z koszyka**
   a. sprawdź czy koszyk jest pusty
   b. wyciągnij jabłko
   c. połóż jabłko na stole
3. **Umyj wszystkie talerze ze stosu brudnych talerzy**
   Załóżmy, że talerze inna osoba może dokładać
   a. sprawdź czy są brudne talerze
   b. wyciągnij talerz ze stosu brudnych talerzy
   c. umyj i wysusz talerz
   d. odłóż talerz na stos czystych talerzy
4. **Przebiegnij 10km robiąc przystanki co 1km**
   a. biegnij 1km
   b. odpoczywaj 1 minutę
5. **Wejdź po schodach na 5 piętro (każde piętro to 8 schodów)**
   a. podejdź do początku schodów
   b. wejdź na stopień schodów
6. **Poszukaj książki w bibliotece**
   a. sprawdź czy jest następna półka z książkami
   b. przejdź do następnej półki
   c. pobierz ilość książek na danej półce
   d. wyciągnij i sprawdź książkę znajdującą się na pozycji x (gdzie 0 <= x < ilość książek na półce)
7. **Zgaduj-zgadula. Zadawaj pytania dopóki nie dostaniesz prawidłowej odpowiedzi:**
   a. zadaj pytanie
   b. pobierz odpowiedź
   c. sprawdź czy odpowiedź się zgadza

Zadania

#loops

Zadania

#loops

1. Napisz metodę, która wyświetli na ekranie n-pierwszych liczb parzystych**.** Zmienna **n** to parametr metody. Czyli np. dla n = 4 program powinien wypisać: 2, 4, 6, 8
2. Napisz metodę, która policzy **n**-tą potęgę (**n >= 0**) liczby całkowitej **a.** Parametry metody to: **n** i **a**.
3. Wypisz na ekran co drugą, dużą literę alfabetu łacińskiego, zaczynając od 'A' i kończąc na 'Z'. Użyj pętli **for**, a potem spróbuj przerobić program używając pętli **while**.
4. Napisz metodę która sprawdzi czy dwa podane Stringi (zmienne typu String) są takie same - bez użycia metody *equals()*.
   Podpowiedź: możesz porównać oba teksty znak po znaku używając jednej z metod klasy String.
5. Napisz metodę sprawdzającą ilość wystąpień frazy: **phrase** w tekście: **text**. Parametry metody to: **phrase** i **text**.
   Podpowiedź: użyj metody klasy String która sprawdza index dla podanej frazy
6. Zmień metodę pl.sda.loops.ForLoop.*sumNumbersFromUser*() tak by przyjmowała liczby typu float. W podsumowaniu oprócz sumy wypisz także średnią arytmetyczną podanych liczb.
7. * Napisz metodę która wyświetli na ekranie prostokąt o podanych rozmiarach: **width** i **height** (to są parametry metody).
   Podpowiedź: zobacz metodę: pl.sda.loops.ForLoop.*leftTriangle*()
8. * Utwórz program który będzie pobierał od użytkownika liczby typu float aż do momentu osiągnięcia limitu podanego jako parametr metody. Na koniec wypisz ile było tych liczb, jaka była ich suma (z częścią ułamkową) i jaka jest ich średnia arytmetyczna.
9. * Napisz metodę sprawdzającą, czy dany łańcuch znaków jest palindromem.

Pamiętaj o wykorzystaniu repozytorium kodu Git! * Dla chętnych.

Spotkanie #4

Wprowadzenie do języka Java

Szybka powtórka

- hermetyzacja, modyfikatory dostępu, pakiety
- klasa String
- pętle

JavaFX tutorial - https://goo.gl/baJEo6

Rozkład jazdy

- enum
- tablice, varargs
- kompozycja, dziedziczenie, polimorfizm

Zanim zaczniemy proszę zaktualizować projekt: **java24gda_intro !**

Instrukcja aktualizacji projektu z GitHub: https://goo.gl/m3BHMr

Enum

Enum

Enum (typ wyliczeniowy, ang. enumeration)

specjalny rodzaj danych umożliwiających reprezentowanie ograniczonego zestawu stałych wartości. Możemy go użyć do reprezentowania: kolorów, stron świata, dni tygodnia itp.

enum to słowo kluczowe za pomocą którego definiujemy typ wyliczeniowy. Stosujemy je zamiast **class**

nazwa enuma

```
public enum Colors {
    RED,
    WHITE,
    BLACK;
}
```

zestaw wszystkich wartości naszego enuma zapisujemy po przecinku

na końcu opcjonalnie (jeżeli enum nie ma pól i / lub metod) średnik

Enum

```
public enum Colors {
        RED(226, 56, 19),
        WHITE(255, 255, 255),
        BLACK(0, 0, 0);

        private int redCode;
        private int greenCode;
        private int blueCode;

        Colors(int red, int green, int blue) {
                redCode = red;
                greenCode = green;
                blueCode = blue;
        }

        public int getRedCode() {
                return redCode;
        }
        ....
}
```

- **enum** to specjalna klasa która może mieć pola, metody i konstruktory
- każda wartość enuma (np. RED, WHITE) to obiekt (instancja) klasy
- nie można utworzyć innej instancji enuma, np.: używając operatora **new**
- konstruktory enuma mogą mieć tylko widoczność: private albo pakietową
- do porównań enumów można używać operatora **==** ponieważ stałe oznaczające wartości enuma są final
- metoda toString() może być nadpisana w enumie

Przykłady w kodzie: pl.sda.enums.EnumExample

Zadania

#enums

Zadania

#enums

1. Utwórz enum **Currency**, który ma reprezentować walutę. Ograniczmy się do 5 walut: polski złoty, dolar, euro, jen, funt brytyjski.
2. Utwórz enum **Operation**, dla którego występować będą wartości: **PLUS**, **MINUS**, **MULTIPLY**, **DIVIDE**.
3. Do enum **Currency** dodaj pole oznaczające symbol waluty: "PLN", "USD" itp., a do **Operation** reprezentację tekstową: "+", "-" itp
4. Dodaj do enum **Operation** metodę *calculate(double a, double b)*, która dla dwóch podanych liczb wykona odpowiednią operację matematyczną oraz wyświetli jej wywołanie w "ładny" sposób na konsoli. Zadbaj o przykład użycia i wykonanie kilku operacji matematycznych.
5. Do enum **Currency** dodaj pole oznaczające kurs waluty (w stosunku do polskiego złotego) oraz metodę która wyliczy wartość podanej kwoty w obcej walucie (np.: 100 PLN ~ 23 EUR)
6. * Zadbaj by można było na bazie reprezentacji tekstowej ("PLN", "+" itp) znaleźć odpowiednią wartość enum **Currency** i **Operation.**
7. * Utwórz klasę **Money** zawierającą dwa pola: **currency** i **value**. Nadpisz metodę do tworzenia wartości tekstowej (*toString()*) tak żeby wyświetlała wartość z symbolem waluty, np. 40 EURO. Dodaj metodę exchange(Currency currency), która zwróci nowy obiekt **Money** w nowej walucie dla aktualnej wartości.

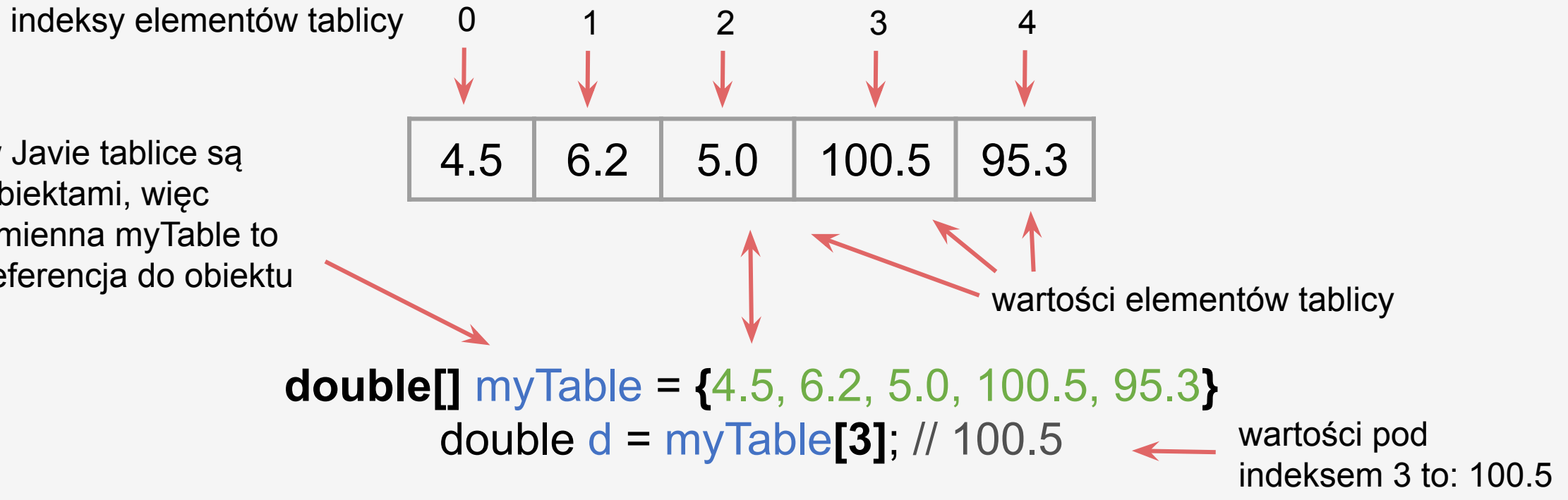
Pamiętaj o wykorzystaniu repozytorium kodu Git! * Dla chętnych.

Tablice

Tablica - definicja

Tablica

zestaw elementów (wartości) tego samego typu, ułożonych na określonych pozycjach. Do każdego z tych elementów mamy bezpośredni dostęp poprzez nazwę tablicy i pozycję elementu w zestawie (określaną jako **indeks** tablicy)



Tablica - deklaracja i inicjalizacja

Deklaracje tablic:

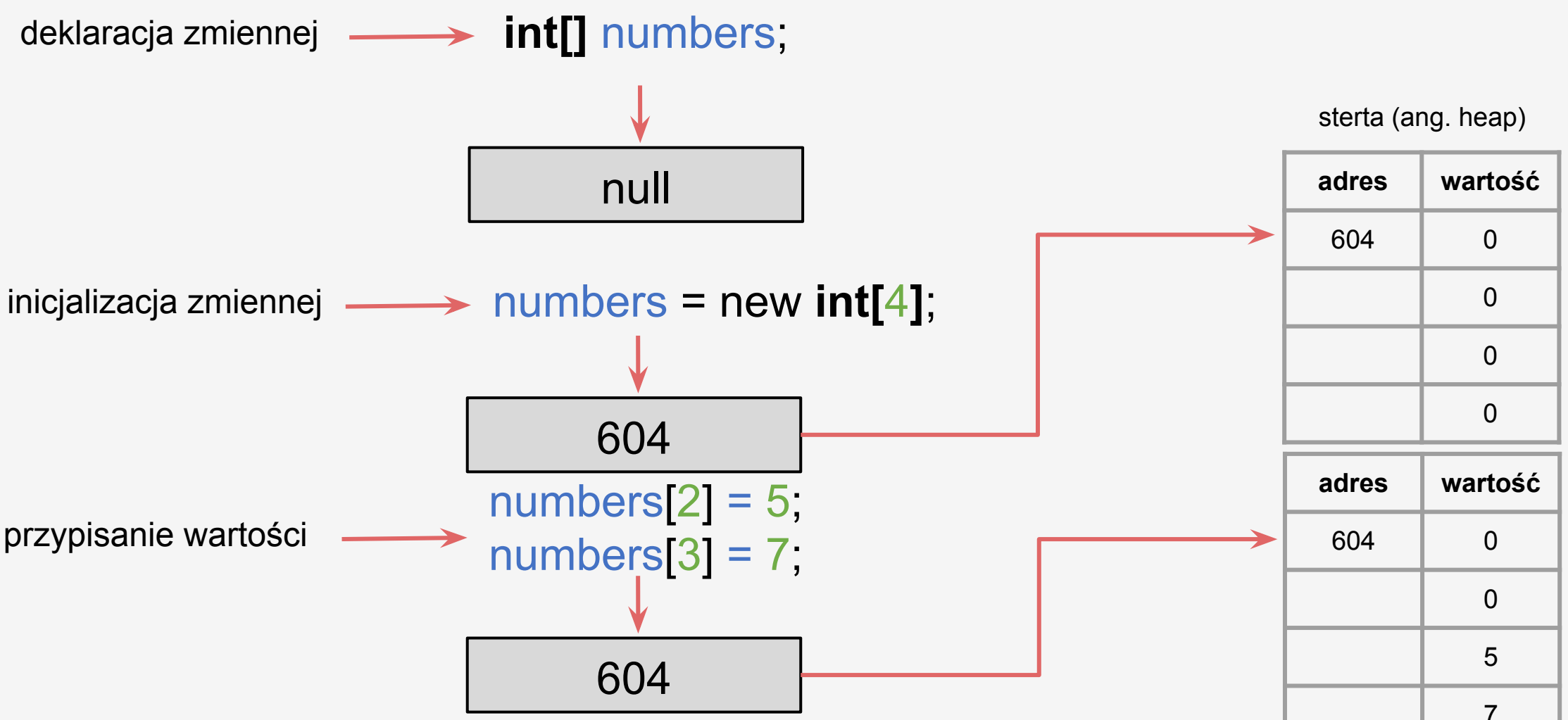
```
int[] arr;

String[] names;

Button[] buttons;

double[][] prices;
```

- tablica zawiera elementy tego samego typu (pierwotnego lub referencyjnego)
- tablice mogą być wielowymiarowe
- rozmiar tablicy jest niezmienny, jest ustawiany raz podczas inicjalizacji
- pierwszy indeks tablicy to zawsze 0
- rozmiar tablicy uzyskujemy przez odwołanie do pola obiektu: nazwa_tablicy.**length** (np. names.**length**)
- w przypadku próby wyciągnięcia wartości z tablicy gdzie indeks będzie poza zakresem (indeks < 0 lub indeks > rozmiar tablicy - 1) zostanie wyrzucony wyjątek: **ArrayIndexOutOfBoundsException**

Inicjalizacja tablic:

```
int[] arr = {1 ,5 ,8, 1};
int[] arr = new int[4];
int[] arr = new int[]{1 ,5 ,8, 1};

String[] names = new String[10];


double[][] prices = {
    {3.4, 5.5}, {1.2, .5}
};
```

Tablica - przykład użycia



Tablica - przykład użycia cd

pobieramy trzecią wartość

```
System.out.println(numbers[2]);
```

pobieramy pierwszą wartość i przypisujemy do zmiennej

```
int a = numbers[0];
```

pobieramy drugą wartość

```
System.out.println(numbers[1]);
```

tu koniecznie musi być ‘<’ a nie ‘<=’

wypisujemy wszystkie wartości tablicy

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

Pętla for each

Pętla for each

to konstrukcja, która pozwala na sekwencyjne przeglądanie różnych zbiorów danych. Mogą nimi być tablice, a także dynamiczne struktury jak na przykład listy (o tym później). Najłatwiejszy sposób na iterowanie się po tablicach i innych kolekcjach.

```
int[] numbers = {1, 3, 5, 7, 9};

for (int i : numbers) {
    System.out.println(i);
}
```

w zmiennej i będą pojawiać się wszystkie wartości z tablicy od pierwszej do ostatniej

Czego nie można z pętlą for-each:

- usuwać elementów w czasie ich przeglądania
- zmieniać aktualnie przeglądanego miejsca tablicy lub kolekcji
- iterować się po wielu tablicach na raz

Przykłady w kodzie: pl.sda.arrays.Arrays

Metody o zmiennej liczbie argumentów (varargs)

Varargs
mechanizm pozwalający na tworzenie metod o zmiennej ilości argumentów, bez konieczności tworzenia tablic przechowujących te argumenty.

[typ] nazwa_metody([lista_parametrów], [typ]... nazwa_parametru)

Deklaracja:
void getNumbers(int… numbers)

zmienna typu tablicowego

float compute(float first, float… theRest)

Wywołanie:
getNumbers()
getNumbers(1)
getNumbers(1, 2, 3, 4, 5)

compute(1.0F)
compute(5.5F, 6.1F, 6.6F)

Przykłady w kodzie: pl.sda.arrays.Varargs

Zadania

#arrays

Zadania

#arrays

1. Napisz metodę, która jako parametr przyjmuje zmienną typu **String[]** i wyświetla wszystkie elementy tablicy na konsoli (użyj różnych rodzajów pętli).
2. Napisz metodę, która jako jedyny parametr przyjmuje zmienną typu **int[]** i zwróci sumę wszystkich elementów tablicy.
3. Napisz metodę, która jako parametr przyjmuje parametr **int count** i w wyniku zwraca tablicę wypełnioną liczbami parzystymi zaczynając od 2, tablica ma zawierać ilość liczb wskazanych przez parametr **count**.
4. Napisz metodę, która jako parametr przyjmuje tablicę typu **float[]** i w wyniku swojego działania powoduje podwojenie wartości dla każdego indeksu przekazanej tablicy.
5. Napisz metodę, która jako parametr przyjmuje zmienną typu **double[]** i wyświetla na konsoli: pierwszy, środkowy (jeden lub dwa środkowe) i ostatni element tablicy, a także średnią arytmetyczną wszystkich liczb z tabeli.
6. Napisz metodę, która jako parametr przyjmuje zmienną typu **Car[]** (klasę Car powinieneś mieć utworzoną w ramach poprzednich zadań) i zwróci tablicę z odwróconą kolejnością elementów.
7. Przerób każdą z poprzednich metod (poza nr 3) tak żeby przyjmowała parametry jako varargs.
8. * Przerób metodę z zadania nr 2 tak, żeby metoda przyjmowała tablicę dwuwymiarową typu **int[][]** i liczyła sumę z wszystkich tablic.
9. * Napisz klasę, która pozwala tworzyć mapę gry w statki dla pojedynczego użytkownika. Mapa powinna być tworzona na bazie dwuwymiarowej tablicy. Zadbaj o metody pozwalające tworzyć statki na mapie, a sam konstruktor klasy powinien pozwalać na utworzenie planszy o zadanych wymiarach. Utwórz także przykład użycia.

Pamiętaj o wykorzystaniu repozytorium kodu Git! * Dla chętnych.

Spotkanie #5

Wprowadzenie do języka Java

Rozkład jazdy

Zanim zaczniemy proszę zaktualizować projekt: **java24gda_intro !**

09:00 - powtórka
09:30 - kompozycja, dziedziczenie, polimorfizm
10:30 - przerwa krótka
10:40 - kompozycja, dziedziczenie, polimorfizm cd.
11:40 - klasy i metody abstrakcyjne
12:40 - przerwa długa
13:00 - interfejsy
14:00 - data i czas - Java API
14:30 - przerwa krótka
14:40 - data i czas - Java API cd.
16:00 - koniec zajęć :)

Instrukcja aktualizacji projektu z GitHub: https://goo.gl/m3BHMr

Szybka powtórka

- hermetyzacja, modyfikatory dostępu, pakiety
- klasa String
- pętle
- enum
- tablice, varargs

Zestawienie instrukcji: https://goo.gl/TLmyZt

Kompozycja, dziedziczenie, polimorfizm

Ponowne użycie klas

Kompozycja
to inaczej zawieranie jednego obiektu w drugim. Jeden obiekt jest częścią składową drugiego (jak np.: żarówka jest częścią lampy).
Kompozycję uzyskujemy przez definiowanie pól obiektowych w klasie.

Podejście obiektowe umożliwia ponowne wykorzystanie (ang. *reusing*) już gotowych klas przy tworzeniu klas nowych, co znacznie oszczędza pracę przy kodowaniu, a także czyni programowanie mniej podatne na błędy.
Istnieją dwa sposoby ponownego wykorzystania klas: **kompozycja** i **dziedziczenie**

Dziedziczenie
przejęcie właściwości (pól) i funkcjonalności (metod) innej klasy i ewentualnej modyfikacji tych właściwości i funkcjonalności tak by były bardziej wyspecjalizowane. Klasa potomna "dziedziczy" z klasy bazowej zestaw pól i metod, które może ponownie wykorzystać.

Kompozycja - przykład

```
class Car {
    private Engine engine;
    ...
    public boolean start() {
        return engine.start();
    }
}
```

obiekt engine jest zawarty w obiekcie klasy Car

do obiektu engine delegowana jest metoda: *start()*
Klasa Car nie musi definiować jej sama.

Dziedziczenie - przykład

```
class Animal {
  private String name;

  public Animal(String name) {
    this.name = name;
  }
  ...
}

class Lion extends Animal {
  private int dailyMeatDemand;

  public Lion(String name, int demand) {
    super(name);
    this.dailyMeatDemand = demand;
  }
}
```

- klasa Lion to podklasa (klasa pochodna) klasy Animal
- klasa Animal to nadklasa (klasa bazowa) klasy Lion

klasa Lion dziedziczy po klasie Animal i przejmuje od niej wszystkie (nie-prywatne i nie-statyczne) pola i metody

Dziedziczenie - inicjalizacja klasy pochodnej

Sekwencja inicjalizowania klasy pochodnej:

1. wywoływany jest konstruktor klasy pochodnej
2. jeżeli pierwszą instrukcją jest **super**({params}), wykonywany jest konstruktor nadklasy z podanymi parametrami
3. w przeciwnym przypadku wywoływany jest konstruktor bezparametrowy klasy bazowej (musi istnieć inaczej kod się nie skompiluje!)
4. wykonywane są instrukcje konstruktora klasy pochodnej

```
class Lion extends Animal {
  private int dailyMeatDemand;

  public Lion(String name, int demand) {
    super(name);
    this.dailyMeatDemand = demand;
  }
}
```

super(name**)** to odniesienie do konstruktora nadklasy:

- jeżeli wywołujemy konstruktor z nadklasy musi to być pierwsza instrukcja w ciele konstruktora podklasy
- jeżeli nie wywołamy konstruktora nadklasy wprost domyślny konstruktor (bezparametrowy) zostanie wywołany niejawnie

Dziedziczenie - nadpisywanie metod

```
class Animal {
  private String name;
  ...
  public String getName() {
      return name;
  }
  ...
}

class Lion extends Animal {
   …
   public String getName() {
      return  "Lion: " + super.getName();
   }
}
```

- **nadpisanie** (przedefiniowanie, ang. *overriding*) metody oznacza utworzenie w klasie pochodnej metody, która ma taką samą **sygnaturę** i **typ wyniku** co w klasie bazowej, ale inną definicję ciała metody
- **sygnatura** metody to jej nazwa i zestaw parametrów
- metody prywatne i statyczne oraz oznaczone słowem kluczowym **final** nie mogą być nadpisane
- w naszym przykładzie metoda ***getName()*** z klasy Lion nadpisuje metodę o tej samej nazwie z klasy Animal
- przy nadpisywaniu metod można rozszerzać dostęp, ale nie zawężać go (czyli można zmienić modyfikator dostępu z np.: protected na public - ale nie odwrotnie)
- odwołania do nadpisanych metod z poziomu metody podklasy realizowane są za pomocą wywołania: **super.nazwa_metody(**{params}**)**

Przeciążanie metod

```
class Animal {

  public void move() {
      print("Animal is moving...");
  }

  public void move(int speed) {
      print("Animal is moving with speed: " + speed);
  }

  public void move(String destination) {
      print("Animal is moving to: " + destination);
  }
}
```

[print = System.out.println] !

- **przeciążanie** (ang. *overloading*) metody oznacza utworzenie w klasie metody o tej samej nazwie ale różnym zestawie parametrów (różna liczba i / lub typie parametrów)
- przeciążone metody mogą należeć do tej samej lub różnych klas (z których jedna pośrednio lub bezpośrednio dziedziczy inną)
- w naszym przykładzie metoda *move()* jest przeciążona dwukrotnie
- podobnie mogą być przeciążone konstruktory - przykład na następnym slajdzie

Przeciążanie konstruktorów

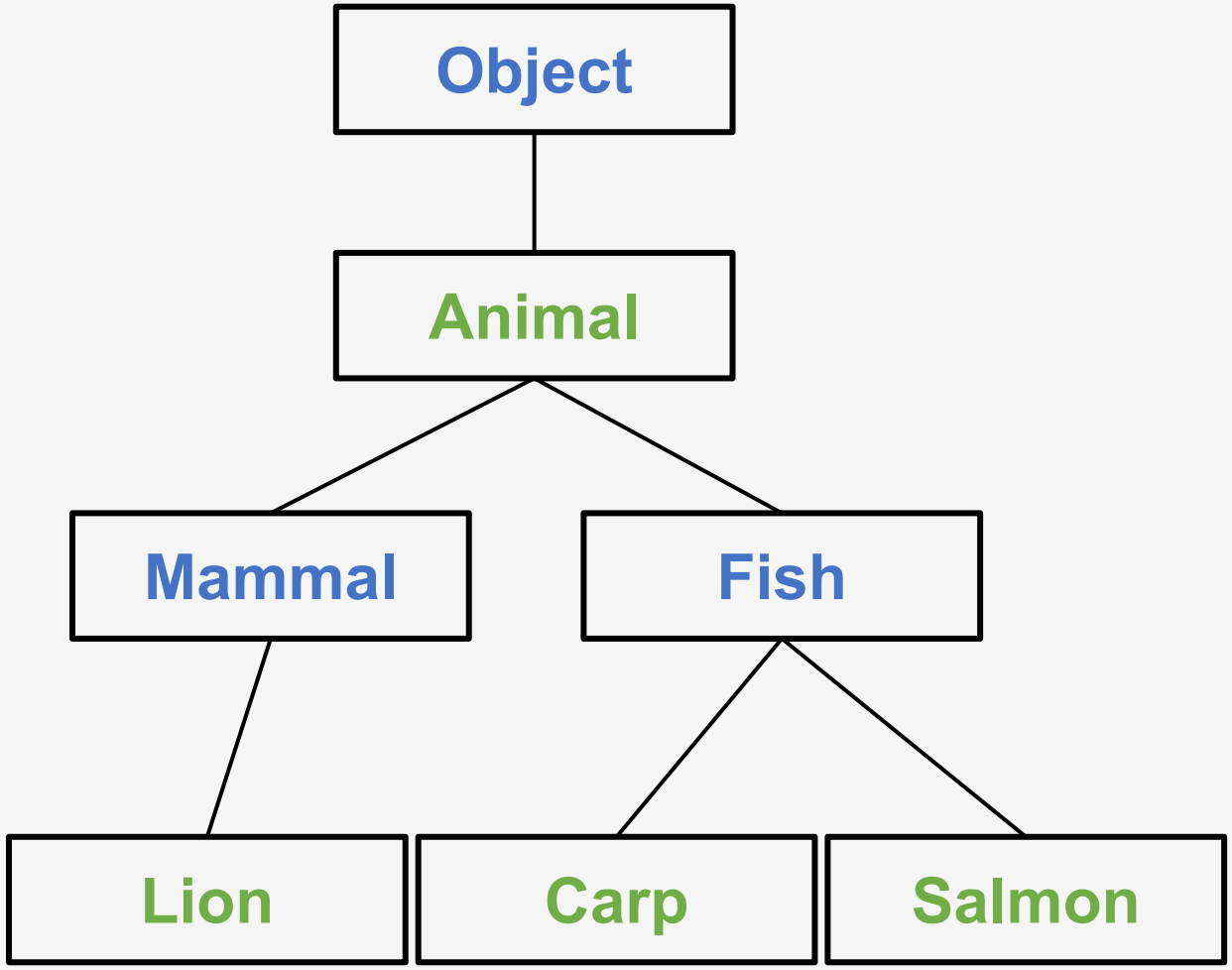
```
class Animal {
    private String name;
    private int age;

    public Animal(String name, int age) {
       this.name = name;
       this.age = age;
    }

    public Animal(String name) {
       this(name, 0);
    }

   public Animal() {
       this("Animal X");
    }
}
```

konstruktor z dwoma parametrami ustawia wartości dla pól, tutaj można też przeprowadzić np.: walidację danych

konstruktor z jednym parametrem wywołuje konstruktor dwuparametrowy. Wywołanie **this(**{params}**),** podobnie jak **super(**{params}**)** jeżeli występuje musi być pierwszą instrukcją w ciele konstruktora

konstruktor domyślny (bezparametrowy) wywołuje konstruktor z parametrem String name

Hierarchia dziedziczenia



- w Javie nie ma wielodziedziczenia, każda klasa może bezpośrednio dziedziczyć tylko po jednej nadklasie
- każda klasa pośrednio może mieć dowolnie wiele nadklas (hierarchia dziedziczenia)
- hierarchia dziedziczenia wszystkich klas zaczyna się w jednym miejscu, każda klasa dziedziczy (bezpośrednio lub pośrednio) po klasie **java.lang.Object**
- można powiedzieć że np.: obiekt klasy Lion jest równocześnie obiektem klasy: Mammal, Animal i Object
- metody *toString()* i *equals()* (i kilka innych) są dziedziczone z klasy **Object**, możemy je nadpisać

Konwersja typów referencyjnych

```
Lion lion = new Lion(..);
Mammal mammal = lion;
Animal animal = lion;
Object object = lion;
```

- zmienne wskazujące na obiekty dowolnej klasy mogą być przypisane do zmiennych, które oznaczają typ dowolnej z nadklas (**lion** ma typ: Lion, Mammal, Animal i Object).
- jest to **rozszerzająca konwersja obiektowa** (ang. *upcasting*).
- konwersja ta dokonywana jest automatycznie przy:
  - przypisywaniu zmiennej
  - przekazywaniu argumentów metodzie
  - zwracaniu wyniku z metody

```
Lion lion2 = (Lion) animal;

Animal animal2;
if (object instanceof Animal) {
    animal2 = (Animal) object;
}
```

- konwersja w drugą stronę (**zawężająca konwersja obiektowa,** ang. *downcasting*) wymaga jawnego użycia operatora konwersji
- w przypadku gdy konwersja zostanie przeprowadzona do niewłaściwego typu zostanie wyrzucony wyjątek: ***ClassCastExcpeption***
- do sprawdzania do jakiej klasy należy obiekt służy operator **instanceof** - zwraca on true jeżeli zmienna po lewej stronie operatora jest typu wskazanego po prawej stronie.

Przykłady w kodzie: pl.sda.inheritance.ReferenceConversion

Krótka powtórka

```
class Shape {
    public void draw() {
        System.out.println("Shape");
    }
}

class Square extends Shape {
    public void draw() {
        System.out.println("Square");
    }
    public double getArea() {
        return 10.5;
    }
}

Square square =  new Square();
Shape shape = square;
Object object = square;
```

- zmienne referencyjne mogą być tylko jednego typu i raz zadeklarowany typ nie może ulec zmianie (zmienna **square** nie może zmienić typu na Object czy Shape)

  square = new Square(); // ok
  square = new Shape(); // taki kod się nie skompiluje

- zmienne referencyjne mogą zmieniać referencje w wyniku przypisania (o ile nie jest zadeklarowana ze słowem **final**)

- typy zmiennych referencyjnych determinują metody, które mogą być wywołane w obiekcie za pomocą zmiennej, do której on się referuje, np:

  double area = **square.**getArea(); // ok
  square.draw(); // ok
  double area = **shape.**getArea(); // taki kod się nie skompiluje
  shape.draw(); // ok

- zmienna referencyjna może referować się do każdego obiektu tego samego typu jaki został zadeklarowany, ale też może referować się do każdego **podtypu** zadeklarowanego typu

Polimorfizm

Polimorfizm (gr. wielopostaciowość)
mechanizm pozwalający programiście używać zmiennych na kilka różnych sposobów. Inaczej mówiąc jest to możliwość wyabstrahowania wyrażeń od konkretnych typów.

```
Square square = new Square();
Shape shape = square;
Object object = square;

square.draw(); // "Square"
shape.draw(); // "Square"
```

- obiekt klasy Square może być traktowany polimorficznie, w zależności od typu zmiennej referencyjnej który na nią wskazuje może być traktowany jako: Object, Shape lub Square
- kod obok tworzy tylko jeden obiekt w pamięci, ale są trzy zmienne które na niego wskazują, typy tych zmiennych determinują to jakie metody na tym obiekcie można wywołać
- mimo tego że zmienne mają różne typy, to podczas wywołania metod zawsze będzie wywołana metoda, która pochodzi z typu (klasy) z jakiej powstał obiekt, czyli w przypadku obiektu klasy Square przedstawionego w przykładzie metoda *draw()* zawsze wyświetli napis "Square" niezależnie czy będzie używana zmienna **square** czy **shape** - jest to polimorficzne wywołanie metody

Zadania

#coinpo

Zadania

#coinpo

1. Rozwiń przykład kompozycji w oparciu o klasę ***Car*** - dodaj klasę ***Entertainment***, która zarządzać będzie systemem rozrywki w Twoim samochodzie, a następnie zadbaj o to, by tworząc obiekt typu ***Car*** konieczne było podanie obiektu typu ***Entertainment***. Klasa ***Car*** powinna wykorzystywać metody dostępne w klasie ***Entertainment***.

2. Utwórz klasę ***Tool***, która będzie reprezentować narzędzia do kupienia w sklepie. Każde narzędzie powinno mieć swój model i cenę. Dodatkowo utwórz klasy: ***Hammer*** i ***Saw***, które będą dziedziczyć po klasie ***Tool***. Klasa ***Hammer*** powinna mieć dodatkowe pole z wagą młotka, a klas ***Saw*** z długością piły. Utwórz klasę ***ToolsShop*** w której utwórz kilka narzędzi i wyświetl ich ceny.

3. Dodaj do klasy ***Tool*** metodę która zwraca opis narzędzia (model + cena). Dodatkowo klasy ***Hammer*** i ***Saw*** powinny rozszerzać opis o swoje unikatowe cechy.

4. * Utwórz klasę ***ShoppingCart***, która będzie reprezentować koszyk z zakupami. Wewnątrz klasy dodaj pole ze zmienną tablicową która będzie przechowywać wybrane produkty (powiedzmy, że maksymalnie można zakupić 10 narzędzi). Dodaj metody do dodawania narzędzi, do wyświetlania ich listy i metodę która zwróci łączną sumę zakupów.

Pamiętaj o wykorzystaniu repozytorium kodu Git! * Dla chętnych.

Klasy i metody abstrakcyjne

Klasy i metody abstrakcyjne

```
public abstract class Animal {
    …
    public abstract AnimalType getType();
    public abstract String getVoice();

    public String speak() {
        return getType() + " of name: " +
                getName() + " says: " + getVoice();
    }
}
```

- metoda abstrakcyjna nie ma implementacji (ciała) i powinna być zadeklarowana ze specyfikatorem abstract
- klasa w której zadeklarowano jakąkolwiek metodę abstrakcyjną, jest klasą abstrakcyjną
- metody abstrakcyjne to takie, co do których nie wiemy jeszcze, jaka może być ich konkretna implementacja
- metody abstrakcyjne muszą mieć implementację w każdej konkretnej klasie bazowej
- klasa abstrakcyjna nie musi mieć metod abstrakcyjnych
- abstrakcyjność klasy oznacza że nie można tworzyć jej obiektów (instancji)
- metody abstrakcyjne mogą być użyte w zwykłych metodach mimo że nie mają jeszcze implementacji

Przykłady w kodzie: pl.sda.abstra.Shapes

Klasy i metody abstrakcyjne - przykład użycia

```
public abstract class Animal {
    …
    public abstract AnimalType getType();
    public abstract String getVoice();

    public String speak() {
        return getType() + " of name: " +
                getName() + " says: " + getVoice();
    }
}
```

jeżeli co najmniej jedna metoda jest abstrakcyjna - klasa też musi być abstrakcyjna

metody abstrakcyjne można używać jak zwykłe metody w kodzie

klasa Mammal jest abstrakcyjna bo nie implementuje metody getVoice() z nadklasy Animal

klasa, która dziedziczy po klasie abstrakcyjnej musi zaimplementować wszystkie jej metody abstrakcyjne albo sama stać się klasą abstrakcyjną

```
public abstract class Mammal extends Animal {
    …
    public AnimalType getType() {
        return AnimalType.MAMMAL;
    }
}
```

Klasy i metody abstrakcyjne - przykład użycia

```
public class Dog extends Mammal {
    …
    public String getVoice() {
        return "Hau, Hau!";
    }
}
```

klas Dog implementuje (pośrednio lub bezpośrednio) wszystkie metody abstrakcyjne swoich przodków - nie musi wobec tego być abstrakcyjną klasą

obiekt myDog jest klasy Dog, ale również Mammal i Animal

```
Animal myDog = new Dog("Reksio");
String speech = myDog.speak();
System.out.println(speech);
Animal myDog2 = new Animal("Reksio");
```

możemy wywołać na nim metody z nadklas

taki kod się nie skompiluje - z klas abstrakcyjnych nie można tworzyć obiektów!

Pełny kod: pl.sda.abstra.Zoo i pl.sda.abstra.ZooKeeper

Zadania

#abstra

Zadania

#abstra

1. Utwórz klasę abstrakcyjną o nazwie **Food** i dodaj do niej metodę abstrakcyjną **getTaste()**, która zwraca String z opisem smaku jedzenia Dodaj klasy: **Chicken**, **Ham**, **Carrot**, **Salad** rozszerzające klasę **Food** oraz zaimplementuj w każdej z nich wymaganą metodę.
2. Utwórz dwie nowe klasy abstrakcyjne: **Meat** i **Vegetable** rozszerzające klasę **Food**. Zmień klasy bazowe dla klas: **Chicken**, **Ham**, **Carrot**, **Salad** tak by dziedziczyły po jednej z klas: **Meat** lub **Vegetable**
3. Dodaj do klasy **Food** metodę abstrakcyjną **getType()**, która zwróci rodzaj jedzenia w postaci enuma (utwórz enum). W klasach **Meat** i **Vegetable** zaimplementuj nową metodę.
4. W klasie **Food** dodaj pole **name** i zwykłą metodę (gettera) do pobierania nazwy. Stwórz konstruktor, który będzie ustawiał pole **name**. Stwórz odpowiednie konstruktory w klasach pochodnych.
5. W klasie **Food** dodaj metodę **describe()**, która wypisze na ekran informacje o nazwie, typie i smaku jedzenia. Sprawdź swój kod, stwórz po jednym obiekcie z każdej klasy: **Chicken**, **Ham**, **Carrot**, **Salad** i wyświetl na ekran ich opis.
6. * Stwórz klasę **Recipe**, która zawierać będzie nazwę i spis składników (obiektów klasy **Food**). Dodaj konstruktor który ustawi nazwę przepisu i wszystkie składniki(jako varargs) i drugi który ustawi nazwę przepisu i ilość składników (jako liczbę). Dodaj metodę do dodawania składników do listy.
7. * W klasie **Recipe** stwórz metody do wyświetlania wszystkich składników (same nazwy) oraz metodę do wyświetlania opisu potrawy składającego się z opisu poszczególnych składników.
8. * Dodaj własne klasy-składniki tak by stworzyć swój ulubiony przepis kulinarny :).

Pamiętaj o wykorzystaniu repozytorium kodu Git! * Dla chętnych.

Interfejsy

Interfejsy - definicje

Interfejs klasy

sposób komunikowania się z jej obiektami. Inaczej - zestaw jej dostępnych do użycia z poziomu innej klasy metod.

API (ang. Application Programming Interface)

sposób, rozumiany jako ściśle określony zestaw reguł i ich opisów, w jaki programy komputerowe, systemy, serwisy i moduły komunikują się między sobą. Definiuje się go na poziomie kodu źródłowego dla składników oprogramowania, na przykład aplikacji, bibliotek, systemu operacyjnego

Interfejs

specjalna konstrukcja w Javie (podobna do klasy), której celem jest wyeksponowanie funkcjonalności (sposobów komunikacji) pomiędzy klasami, bibliotekami czy frameworkami. Wspomaga hermetyzację kodu i częściowo rozwiązuje problem wielodziedziczenia.

Implementacja interfejsu

zdefiniowanie w klasie wszystkich metod interfejsu

Interfejsy

```
public interface Figure {
    double PI = 3.14159;

    double getArea();
    double getPerimeter();

    default void print() {
        System.out.println(getArea());
        System.out.println(getPerimeter());
    }
}
```

- interfejs jest podobny do klasy abstrakcyjnej, w której wszystkie metody są abstrakcyjne
- podobnie jak klasy interfejsy wyznaczają typy zmiennych
- nie można utworzyć instancji interfejsu
- umożliwia dziedziczenie wielobazowe
- interfejsy mogą dziedziczyć inne interfejsy
- klasa może rozszerzać wiele interfejsów
- jeśli posiada pole, to jest ono z definicji **public**, **static**, **final**
- od Java 8 metoda może posiadać domyślną implementację (**default**)
- pozostałe metody (inne niż domyślne) są z definicji **public** i **abstract**

Interfejsy - przykład użycia

```
public interface Figure {
    double PI = 3.14159;

    double getArea();
    double getPerimeter();

    default void print() {
        System.out.println(getArea());
        System.out.println(getPerimeter());
    }
}
```

stała **PI** jest z definicji publiczna, statyczna i niezmienna (final)

publiczne, abstrakcyjne metody które muszą być zaimplementowane w podklasach

domyślna metoda (oznaczamy ją słowem kluczowym default), która posiada kod (ciało), może być nadpisywana przez klasy implementujące

interfejs Drawable z jedną metodą abstrakcyjną

```
public interface Drawable {
    void draw();
}
```

Interfejsy - przykład użycia

```
public class Rectangle implements Figure, Drawable {
    @Override
    public double getArea() {
        return width * height;
    }
    @Override
    public double getPerimeter() {
        return 2 * width + 2 * height;
    }
    @Override
    public void draw() {...}
}
```

implementować można więcej niż jeden interfejs

do oznaczenia relacji między interfejsem a implementacją używamy słowa kluczowego implements

klasa, która implementuje interfejs musi posiadać implementacje wszystkich metod zawartych w interfejsie albo być klasą abstrakcyjną!

obiekt figure jest klasy Rectangle, ale również Figure

```
Figure figure = new Rectangle(3, 5);
figure.print();
```

obiekt figure jest klasy Rectangle, ale również Drawable

```
Drawable figure = new Rectangle(3, 5);
figure.draw();
```

Pełny kod: pl.sda.interfaces.Figures

Zadania

#interfaces

Zadania

#interfaces

1. Utwórz interfejs **Animal** oraz dodaj do niego sygnatury metod: **getName()** i **speak()**.
2. Utwórz kilka różnych implementacji interfejsu **Animal** po jednym dla: ptaków, ssaków, ryb, gadów, owadów.
3. Utwórz kolejne interfejsy:
   a. **Flyable** z sygnaturą metody **fly()**
   b. **Swimmable** z sygnaturą metody **swim()**
4. Dodaj do klas zwierząt implementacje odpowiednich interfejsów:
   a. dla ryb i gadów **Swimmable**,
   b. dla ptaków i owadów: **Flyable**
5. Utwórz nowy interfejs **Being** i dodaj do niego:
   a. sygnaturę metody **getAge()**
   b. pole **MAX_AGE** = 100
   c. metodę domyślną: **isAlive()** - która zwróci **true** jeżeli wiek istoty jest mniejszy od **MAX_AGE**.
6. Dodaj do interfejsu **Animal** dziedziczenie z interfejsu **Being**
7. Nadpisz w jednej z klas zwierząt metodę **isAlive()**
8. Stwórz obiekty dla każdego zwierzęcia i wyświetl informacje o nich (nazwę, wiek, mowę, czy żyją?)
9. * Utwórz interfejs **Plant**, który będzie dziedziczył po **Being** i nadpisze metodę **isAlive()** tak żeby limit wieku był równy 1000 lat. Dodaj kilka klas implementujących interfejs **Plant.**
10. * Utwórz klasę **Swimmingpool** w konstruktorze pobierz listę obiektów typu **Swimmable** i dodaj metodę, która sprawi że wszystkie zwierzęta będą pływać.

Pamiętaj o wykorzystaniu repozytorium kodu Git! * Dla chętnych.

Data i czas

Data i czas

Unix Epoch
początek roku 1970 (1 stycznia 1970 r. UTC), który wyznacza początek tzw. ery Unixa. Czas w Javie liczony jest jako liczba milisekund lub nanosekund od Epoch.

do Java 8	od Java 8
java.util.Date (JDK1.0) java.util.Calendar (JDK1.1) JodaTime - http://www.joda.org/joda-time	Java Date API (wzorowane na JodaTime) java.time.LocalDate java.time.LocalTime java.time.LocalDateTime java.time.ZonedDateTime ...

Data i czas - Date i Calendar

```
Date now = new Date();
System.out.println("Current date: " + now);

Date epoch = new Date(1);
System.out.println("EPOCH: " + epoch);
System.out.println("Now is older: " + now.before(epoch));

Calendar calendar = Calendar.getInstance();
System.out.println("Current calendar: " + calendar);

Calendar beginOf21century = Calendar.getInstance();
beginOf21century.set(Calendar.YEAR, 2000);

Date date = calendar.getTime();
calendar.setTime(date);
```

- klasa Date ma dwa konstruktory (jeden bezargumentowy i jeden liczbowy (**long**) oznaczający liczbę milisekund od daty Epoch)
- dodatkowo klasa Date ma metody: **before**(Date) i **after**(Date) które mogą porównać dwa obiekty klasy Date
- klasa Calendar posiada metody do ustawiania poszczególnych składników daty (rok, miesiąc, dzień) a także czasu (godzina, minuta, itp)
- data zapisana w obiektach klasy Date jest niezależna od strefy czasowej, natomiast dla obiektów klasy Calendar można ustawić strefę czasową

Data i czas - formatowanie

```
SimpleDateFormat dateFormat = new
SimpleDateFormat("yyyy-MM-dd HH:mm:ss");

Date now = new Date();
String dateAsString = dateFormat.format(now);
System.out.println("Today is:  " + dateAsString);
//Today is:  2018-11-09 10:02:19
```

Symbole we wzorcu:

y - rok, dostępne wersje: yy or yyyy.
M - miesiąc roku, wersje: MM, MMM, MMMMM
d - dzień miesiąca, wersje: d, dd
h - godzina dnia, 1-12 (AM / PM), wersje: hh
H - godzina dnia, 0-23, wersje: HH
m - minuta, 0-59, wersje: mm
s - sekundy, 0-59, wersje: ss
S - milisekundy, 0-999, wersje: SSS
E - dzień tygodnia (wtorek, środa itp),
wersje: EE, EEEE
D - dzień roku (1-366)
z - strefa czasowa, wersje: zz, zzzz

'{text}' - znaki pomiędzy cudzysłowami będą wyświetlane wprost

Przykłady w kodzie: pl.sda.datetime.DateAndCalendarExamples

String - formatowanie

```
String greetings = String.format(
"Hello Folks, welcome to %s !", "SDA course");

System.out.printf("|%-20.5s|%n", "Hello World");

System.out.printf("|%010d|%n", 1234);

System.out.printf("|%-15.2f|%n", 55.6789);

System.out.printf("Minutes: %tM %n", new Date());
```

Symbole we wzorcu:

%n - znak nowej linii
%s - formatuj zmienną do stringa
%d - formatuj zmienną do integera
%f - formatuj zmienną do float
%t - formatuj zmienną do daty/czasu

Przykładowe ustawienia:

%{n}.{m}s - konwertuj zmienną do stringa o minimalnej długości **n** (resztę wypełnij spacjami) i przytnij wartość zmiennej do **m** znaków
%-{n}.{m}s - to samo co wyżej ale wyrównaj tekst do lewej
%0{n}d - konwertuj zmienną do integera o minimalnej długości **n** (resztę wypełnij zerami)
%{n}.{m}f - konwertuj zmienną do integera o minimalnej długości **n** (resztę wypełnij spacjami) i przytnij część ułamkową do m cyfr po przecinku

Przykłady w kodzie: pl.sda.strings.StringFormats

Data i czas
problemy z Date i Calendar

- trudne w użyciu
- mają problemy ze strefami czasowymi
- zmieniają stan
- mała dokładność (ms)

--- "Some of the date and time classes also exhibit quite poor API design. For example, years in java.util.Date start at 1900, months start at 1, and days start at 0—not very intuitive." ---

https://www.oracle.com/technetwork/articles/java/jf14-date-time-2125367.html

Data i czas

java.time.* - klasy bez określonej strefy czasowej

```
LocalDate localDate = LocalDate.now();
System.out.println("LocalDate: " + localDate);
localDate = LocalDate.of(2015, 2, 20);
System.out.println("LocalDate: " + localDateOf);

LocalTime localTime = LocalTime.now();
System.out.println("LocalTime: " + localTime);
localTime  = LocalTime.parse("09:00");
System.out.println("LocalTime: " + localTime );

Instant instant = Instant.now();
System.out.println("Instant = " + instant);
```

LocalTime – reprezentuje czas, bez powiązania go z konkretną strefą czasową

LocalDate – to samo co wyżej, ale reprezentuje datę

LocalDateTime – klasa łącząca dwie powyższe

Instant – reprezentuje (podobnie jak java.util.Date) konkretny punkt, jednoznacznie określony w czasie (z dokładnością do nanosekundy, java.util.Date ma dokładnośc do milisekundy).

Przykłady w kodzie: pl.sda.datetime.LocalDateTimeExamples

Data i czas

java.time.* - klasy z określoną strefą czasową

```
LocalDateTime localDateTime =
LocalDateTime.now();

ZoneId zoneId = ZoneId.of("Europe/Warsaw");
ZonedDateTime zonedDateTime =
ZonedDateTime.of(localDateTime, zoneId);

ZoneOffset zoneOffset = ZoneOffset.of("+02:00");
OffsetDateTime offSetByTwo =
OffsetDateTime.of(localDateTime, zoneOffset);
```

ZonedDateTime – data i czas powiązane z konkretną strefą czasową

ZoneId – identyfikator strefy czasowej jako rejonu geograficznego

OffsetDateTime – data i czas powiązane z konkretną strefą czasową, ale rozumianą jako przesunięcie czasu

ZoneOffset – przesunięcie czasu związane ze strefą czasową opisane w godzinach

Przykłady w kodzie: pl.sda.datetime.ZoneDataAndTimeExamples

Data i czas

java.time.* - klasy pomocnicze, odstępy czasowe

Period - klasa reprezentuje odstęp czasu liczony w dniach, miesiącach i latach

```
LocalDate initialDate  = LocalDate.parse("2017-09-30");
LocalDate finalDate = LocalDate.parse("2018-10-01");
Period between = Period.between(finalDate, initialDate);
System.out.println("Period: " + between); //P-1Y-1D
```

Duration - klasa reprezentuje odstęp czasu liczony w sekundach, minutach i godzinach

```
LocalTime initialTime = LocalTime.of(6, 30, 0);
LocalTime finalTime = LocalTime.of(15, 30, 0);
Duration duration = Duration.between(finalTime, initialTime);
System.out.println("Duration: " + duration); //PT-9H
```

Przykłady w kodzie: pl.sda.datetime.PeriodAndDurationExamples

Zadania

#datetime

Zadania

#datetime

1. Za pomocą klasy **Calendar** stwórz obiekt klasy **Date**, który reprezentuje datę i godzinę Twoich urodzin. Wyświetl ją na konsolę.
2. Wyświetl na konsoli datę z pkt 1 z nazwą miesiąca i nazwą dnia tygodnia.
3. Wyświetl na konsoli datę z pkt 1, ale uwzględniając strefę czasową: Tokio w Japonii i Hawaii w USA.
4. Powtórz zadanie 1 i 2 używając klasy **LocalDateTime**.
5. * Powtórz zadanie 3 używając klasy **LocalDateTime** i **ZonedDateTime** .
6. Dla swojej daty urodzenia wyświetl ilość lat, miesięcy, dni i godzin, które miały miejsce do obecnego momentu. Następnie wypisz ile minut (całkowicie) i sekund (całkowicie) minęło od daty urodzenia.
7. Napisz program, który wyświetli dni tygodnia, w których obchodzić będziesz urodziny przez kolejne 10 lat.
8. Napisz metodę, która pobierze tablicę dat (**LocalDate**) i zwróci najnowszą datę.
9. Napisz metodę, która pobierze tablicę dat i godzin (**LocalDateTime**) i zwróci najstarszą datę.
10. * Napisz metody, które przetransformują stary format Date na LocalDate i LocalDateTime.
11. * Napisz metody, które przetransformują LocalDate i LocalDateTime na stary format Date.

Pamiętaj o wykorzystaniu repozytorium kodu Git! * Dla chętnych.

Spotkanie #6

Wprowadzenie do języka Java

Szybka powtórka

- kompozycja, dziedziczenie, polimorfizm
- klasy i metody abstrakcyjne
- interfejsy
- data, czas

Rozkład jazdy

- pola, metody i klasy statyczne
- wyjątki
- kolekcje

Zanim zaczniemy proszę zaktualizować projekt: **java24gda_intro !**

Instrukcja aktualizacji projektu z GitHub: https://goo.gl/m3BHMr

Pola, metody i klasy statyczne

Pola i metody statyczne

```
public class Vehicle {
    private static int count;
    private int vehicleId;

    public Vehicle(String name) {
        ...
        this.vehicleId = ++count;
    }

    public static int getCount() {
        return count;
    }

    public int getVehicleId() {
        return vehicleId;
    }
```

- składowe klasy (pola i metody) mogą być: statyczne i niestatyczne
- składowe niestatyczne (vehicleId, getVehicleId()) zawsze wiążą się z istnieniem jakiegoś obiektu
- składowe statyczne (count, getCount()):
  - są deklarowane przy pomocy słowa kluczewego static
  - mogą być używane nawet wtedy, gdy nie istnieje żaden obiekt klasy
- ze statycznych metod nie można odwoływać się do niestatycznych składowych klasy
- ze statycznych metod można odwoływać się do innych statycznych składowych klasy
- spoza klasy do jej statycznych składowych można się odwołać za pomocą:
  - *NazwaKlasy*.*NazwaSkładowej* - np. Vehicle.getCount()
  - gdy istnieje jakiś obiekt za pomocą selektora '.' jak do innych składowych, np:
    Vehicle **vehicle** = new Vehicle();
    vehicle.getCount()

Inicjacja pól obiektu

```
public class Vehicle {
    private static int count = 100;
    private int vehicleId;

    public Vehicle(String name) {
        ...
        this.vehicleId = ++count;
    }
    ...
}
```

- pola klasy mają zagwarantowaną inicjację na wartość ZERO:
  - 0 dla typów liczbowych
  - false dla logicznych
  - null dla referencji
- zwykle w konstruktorze dokonuje się reinicjacji pól
- można również posłużyć się jawną inicjacją przy deklaracji pól:
  ```
  private static int count = 100
  private int vehicleId = 5;
  ```
- reguły inicjacji:
  - każde pierwsze odwołanie do klasy inicjuje najpierw pola statyczne:
    - `Vehicle.getCount()` //odwołanie do składowej statycznej
    - `Vehicle` **`vehicle`** `= new Vehicle();` //utworzenie obiektu
  - tworzenie obiektu (**new**) inicjuje pola niestatyczne po czym wykonywany jest konstruktor
  - kolejność inicjacji pól:
    - najpierw statyczne od góry do dołu
    - potem niestatyczne od góry do dołu

Bloki inicjacyjne

```
public class Vehicle {
    private static int count;
    private int vehicleId;
    {
        System.out.println("non-static");
        vehicleId = count++;
    }

    static {
        System.out.println("static");
        count = 100;
    }
    ...
}
```

- niestatyczny blok inicjacyjny dodajemy, ujmując kod wykonywalny w nawiasy klamrowe i umieszczając taką konstrukcję poza ciałem jakiejkolwiek metody
- kod takiego bloku zostanie wykonany za każdy razem gdy tworzony jest nowy obiekt - **przed** wywołaniem konstruktora
- statyczny blok inicjacyjny wykonuje się raz przy pierwszym odwołaniu do klasy
- tworzymy go podobnie jak niestatyczny blok, ale dodajemy przed otwierający nawias słowo kluczowe static
- z takiego bloku możemy się odwoływać tylko do składowych statycznych

Pełny kod: pl.sda.stat.Vehicle

Klasy wewnętrzne

```
public class ExternalClass {
    public static class InnerStaticClass {
        public InnerStaticClass() {
            System.out.println("InnerStaticClass");
        }
    }

    protected class InnerNormalClass {
        InnerNormalClass() {
            System.out.println("InnerNormalClass");
        }
    }
    ...
}
```

- klasa wewnętrzna to klasa zdefiniowana wewnątrz innej klasy
- klasy wewnętrzne mogą być ukryte przed innymi klasami pakietu (mogą być private!)
- dzięki nim można uniknąć kolizji nazw
- porządkują kod
- mogą być niestatyczne - wtedy mają dostęp do wszystkich składowych klasy otaczającej
- mogą być statyczne - zadeklarowane ze specyfikatorem static - wtedy mają dostęp tylko do statycznych składowych klasy otaczającej

Przykłady w kodzie: pl.sda.stat.StaticExample

Argumenty metody main

main() to publiczna, statyczna metoda od której zaczyna się wykonanie programu

parametrem metody main() jest tablica stringów, w której przekazane są parametry wywołania programu

zmienna **args** przechowuje referencje do tablicy z stringami, możemy się nią posługiwać jak zwykłą tablicą

```
public static void main(String[] args) {
    if (args.length == 0) {
        return;
    }

    String name = args[0];
    String number = "?";
    if (args.length > 1) {
        number = args[1];
    }
    …
}
```

Pełny kod: pl.sda.stat.MainArgs

Film z instrukcją: https://goo.gl/4djZM7

Zadania

#stat

Zadania

#stat

1. Napisz własną klasę **Math**, która będzie posiadać metody statyczne: **add**, **subtract**, **multiply**, **divide**, **min**, **max**, **pow** gdzie każda z nich będzie przyjmowała dwa parametry liczbowe. Dla każdej z metod utwórz przykład użycia.
2. Dodaj do klasy **Math** statyczną stałą **PI = 3,14**, a następnie dodaj metodę do wyliczania pola koła (πr^2). W przykładzie użycia oblicz pole koła o promieniu: **8**.
3. Dodaj do swojej klasy konstruktor prywatny, który nie pozwoli utworzyć instancji klasy.
4. * Dodaj do klasy **Math** statyczne klasy wewnętrzne i zadbaj o odpowiedni podział metod. Przykład: Operation [multiply, divide, add, minus], Compare [min, max]
   Zmodyfikuj swoje przykłady użycia.
5. Utwórz klasę **Product** z polami: **id**, **name**, **price**. Dodaj konstruktor z polami **name** i **price**, gettery i metodę toString(). Pole **id** powinno być inicjalizowane w bloku incjacyjnym i powinno mieć wartość pobieraną ze statycznego pola **counter,** które zlicza wszystkie stworzone produkty - liczenie powinno się zaczynać od liczby 100.
6. Utwórz kilka produktów i wyświetl informacje o nich na konsoli.
7. * Dodaj do klasy **Product** stałą **DISCOUNT**. W statycznym bloku incjacyjnym ustaw wartość pola **DISCOUNT** na 0.3 (30%) jeżeli dzisiaj jest poniedziałek i 0.0 w pozostałych przypadkach. Uwzględnij rabat przy wyliczaniu ceny.

Pamiętaj o wykorzystaniu repozytorium kodu Git! * Dla chętnych.

Wyjątki

Wyjątek
Exception

Wyjątek - występuje w momencie, gdy zostaje zaburzone działanie aplikacji.

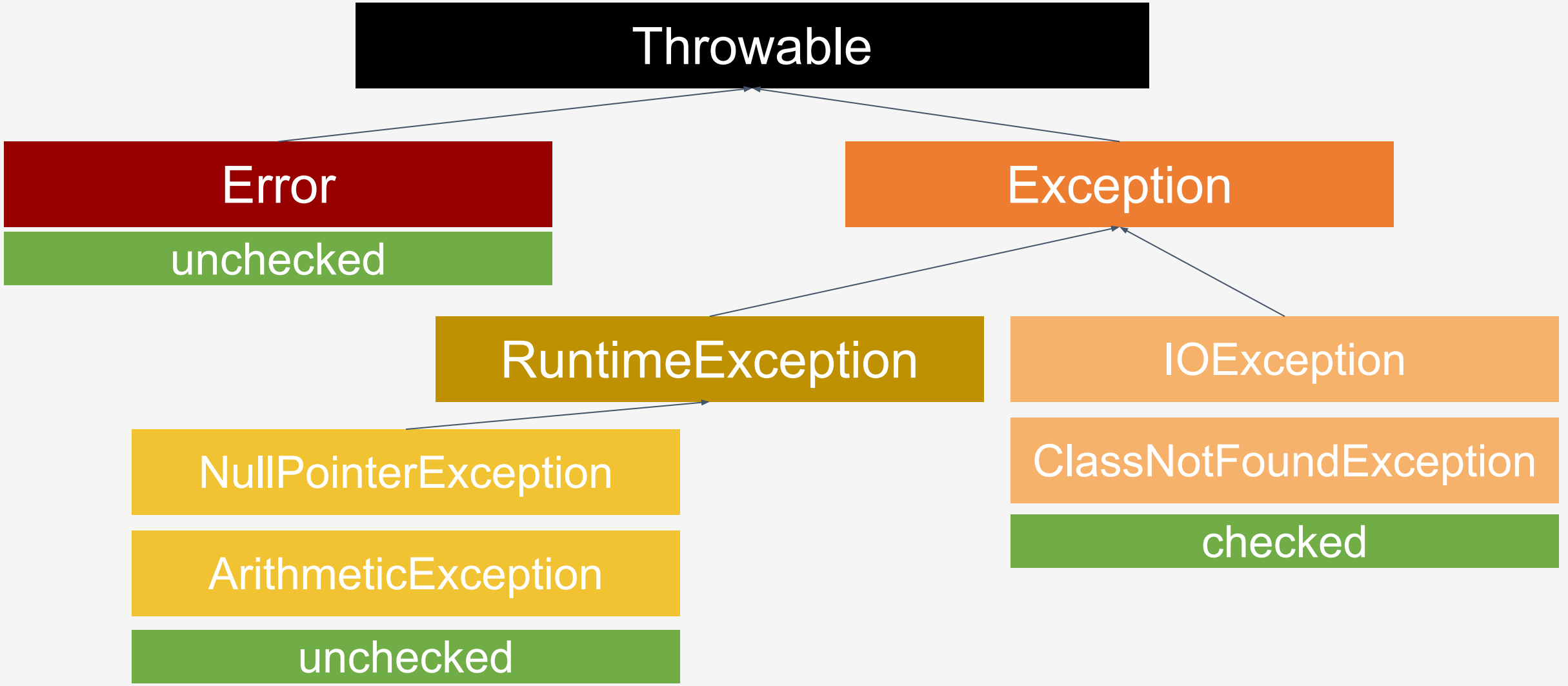
Wyjątkowe sytuacje wymagają specjalnego traktowania w naszym kodzie.

Nazwa **Exception** pochodzi od **exceptional event**.

Wyjątki

- **Throwable** - klasa bazowa
- **Error** - zgłaszane przez JVM lub środowisko - nie jesteśmy w stanie ich obsłużyć
- **Exception** - wyjątki, które musimy obsłużyć w naszym kodzie = wyjątki jawne (checked)
- **RuntimeException** - wyjątki, które mogą się pojawić w trakcie pracy naszej aplikacji, ich obsługa jest opcjonalna = wyjątki niejawne (unchecked)

Wyjątki
hierarchia



Wyjątki
throw vs throws

```
...
throw new IllegalArgumentException();
...

public void loadFile() throws IOException {
...
}
```

Wyjątki
try .. catch .. finally

```
try {
...
} catch (Exception e) {
...
} finally {
...
}
```

Zadania

#exceptions

Zadania

#exceptions

1. Dodaj obsługę wyjątków w klasach **GetNumber** oraz **PrintTable**.
2. Dodaj sekcję **finally**, która w klasach GetNumber oraz PrintTable wykona się i wyświetli komunikat końcowy.
3. Przerób klasę **ExceptionExample** tak, by "łapała" dwa wyjątki w jednej sekcji **catch**.
4. Napisz program, który przyjmie od użytkownika ciąg liczb oddzielonych spacją. Następnie obliczy sumę podanych liczb i wyświetli ją na ekranie. Dodaj obsługę wyjątków w taki sposób, by na ekranie zawsze pojawiła się odpowiedź - samodzielnie znajdź możliwe do wystąpienia wyjątki.
5. * Utwórz własny wyjątek, który będzie rzucany przez metodę do sumowania liczb z poprzedniego punktu, w momencie gdy suma będzie mniejsza od zera.
6. * Napisz obsługę własnego wyjątku.
7. * Napisz klasę Account, która będzie zawierać metodę: **withdraw()** w celu podjęcia środków. Metoda powinna obsługiwać przypadek niedostatecznej ilości środków na koncie za pomocą wyjątku.

Pamiętaj o wykorzystaniu repozytorium kodu Git! * Dla chętnych.

Kolekcje

Zacznijmy jednak od tablic

Czym są?

Kolekcje

+++

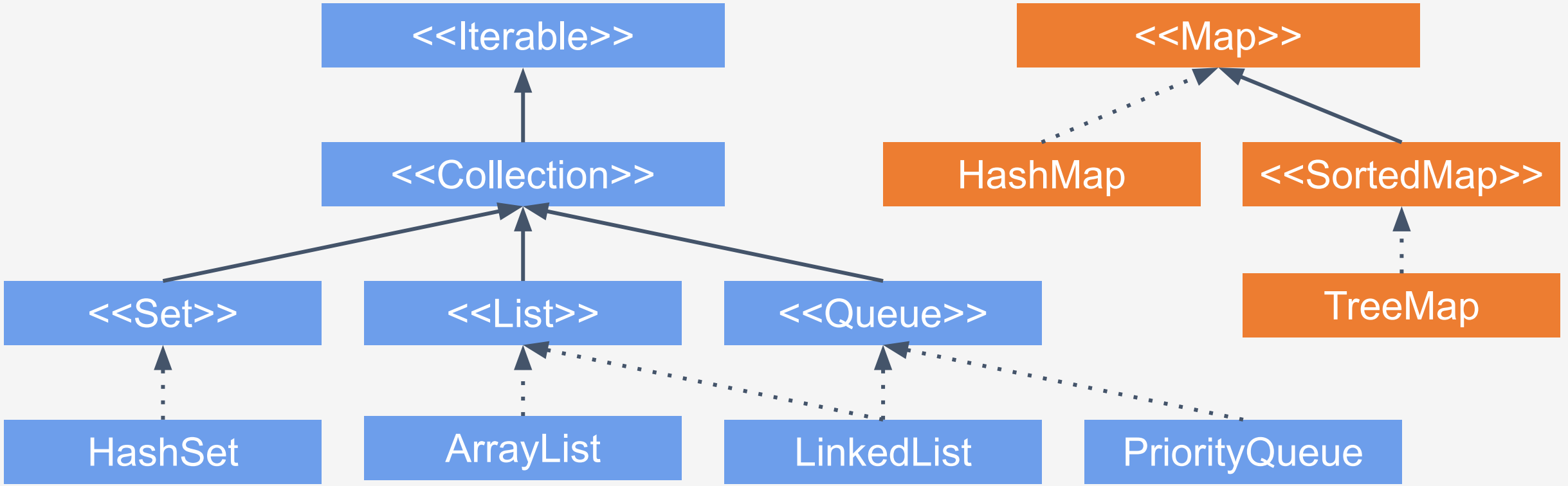
Specjalna grupa klas do przechowywania zbiorów obiektów, tworząca w ten sposób zestaw danych, na których możemy jednocześnie wykonywać operacje oraz przeglądać poszczególne elementy.

+++

Kolekcje
to

- "tablice na sterydach"
- struktury danych, które działają lepiej lub gorzej w zależności od spełnianych warunków
- kolejność elementów lub jej brak
- dynamicznie zmieniany rozmiar
- metody dostępowe
- java.util.Collection (java.util.Map)

Kolekcje
diagram



<<...>> - interface, ·····► - implements, ——► - extends

Kolekcje
rodzaje

java.util.**List** - odpowiednik tablicy z dynamicznym rozmiarem oraz metodami dostępowymi. Zapewnia kolejność poszczególnych elementów. Każdy obiekt ma przypisany własny indeks (miejsce w kolekcji). Ten sam obiekt może być elementem kolekcji wielokrotnie.

- java.util.**ArrayList**

Używana w większości przypadków. Zapewnia liniowy dostęp do poszczególnych elementów.

- java.util.**LinkedList**

Liniowe dodawanie oraz usuwanie elementów. Każdy obiekt kolekcji zawiera informację o kolejnym.

Kolekcje
rodzaje

java.util.**Set** - zbiór, w którym nie dostaniemy się do obiektu za pomocą indeksu. By pobierać kolejne elementy kolekcji w sposób losowy lub posortowany (w zależności od implementacji), potrzebujemy obiektu typu **Iterator**. Obiekty nie mogą się powtarzać.

- java.util.**HashSet**

Zbiór nieposortowany. Kolejność elementów może się zmieniać. Szybkie dodawanie oraz sprawdzanie istnienia elementów.

- java.util.**LinkedHashSet**

To samo co HashSet (dziedziczy po niej), ale kolejność elementów jest stała.

- java.util.**TreeSet**

Struktura drzewiasta, gdzie elementy są posortowane według porządku naturalnego jeśli implementują one interfejs **Comparable**

Kolekcje
rodzaje

java.util.**Queue** - kolejki dzielą się na dwa typy – **LIFO** (last-in, first-out) oraz **FIFO** (first-in, first-out). Ideą kolejki jest przechowywanie obiektów do przetworzenia w określonej kolejności. Wyróżniamy głowę oraz ogon kolejki.

- java.util.**LinkedList**
- java.util.**ArrayDeque**

Przechowuje elementy w tablicy. Zapewnia dostęp FIFO oraz LIFO (poprzez ogon).

- java.util.**PriorityQueue**

Nie pozwala na wartości **null**. Każdy element ma swój priorytet wykonania.

Kolekcje
rodzaje

java.util.**Map** - nie implementuje interfejsu Collection, ale należy do Java Collections API. Jest to zbiór **klucz -> wartość** danego typu, gdzie klucze muszą być unikalne, a wartości mogą się powtarzać.

- java.util.**HashMap**

Mapa nieposortowana, gdzie kolejność iteracji jest nieokreślona. Ogólna implementacja, którą możemy utożsamiać ze słownikiem.

- java.util.**LinkedHashMap**

Dziedziczy po HashMap, ale zapewnia taką samą kolejność elementów w momencie iterowania po kolekcji.

- java.util.**TreeMap**

Mapa posortowana według kluczy w sposób naturalny gdy elementy implementują Comparable. Sortowanie zapewnione jest już na etapie dodawania nowego elementu.

Kolekcje
kontrakt hashCode i equals

- **equals** - służy do porównywania obiektów, a na dodatek jest:
  - zwrotna - object.equals(object) == true
  - symetryczna - a.equals(b) == b.equals(a)
  - przechodnia - a.equals(b), b.equals(c), a.equals(c)
  - spójna - zawsze zwraca ten sam wynik dla tego samego porównania
- **hashCode** - metoda zwracająca "skrót" danego obiektu (hash) w formie typu **int**
  - najlepiej rozkład jednostajny
  - przyporządkowanie do grupy
  - więcej niż jeden obiekt może mieć ten sam hash
  - zawsze zwraca tą samą wartość dla tego samego obiektu

X.hashCode() == Y.hashCode() - obiekty równoznaczne, należy sprawdzić equals
X.hashCode() != Y.hashCode() - equals również zwróci false

Kolekcje
klasy narzędziowe

- java.util.**Arrays** - klasa narzędziowa dla tablic. Posiada sporo metod pozwalających na poruszanie się po tablicach oraz transformacji tablic do kolekcji.

- java.util.**Collections** - klasa narzędziowa dla kolekcji. Pozwala na proste operacje tworzenia, modyfikacji, przeszukiwania, sortowania dla kolekcji.

Zadania

#collections

Zadania

#collections

1. Bazując na kodzie utworzonym na potrzeby omawiania interfejsów, utwórz kolekcję figur każdego typu. Na podstawie utworzonej kolekcji, oblicz pola i obwody wszystkich figur znajdujących się w kolekcji.
2. Napisz metodę, która porówna ze sobą dwie kolekcje i wyświetli rezultat na konsoli.
3. Napisz klasę o nazwie **Person**, która będzie zawierać pola: **firstName** i **lastName**. Klasa powinna implementować interfejs **Comparable**. Następnie utwórz kolekcję i dodaj do niej kilka obiektów klasy Person. Kolekcja powinna sortować alfabetycznie wszystkie dodawane elementy.
4. Napisz metodę, która będzie przyjmować imiona od użytkownika, a wprowadzenie znaku "q" przerwie swoje działanie i wyświetli wszystkie unikalne imiona dodane do kolekcji.
5. Poćwicz wykorzystanie klas **Arrays** oraz **Collections**. Zapoznaj się z ich API i postaraj się wykorzystać dostępne metody do operacji na tablicach i kolekcjach.
6. * Rozszerz klasę **Car** z przykładu i dodaj do niej pola opisujące własności samochodu. Utwórz kolekcję samochodów możliwych do kupna (przynajmniej 10 pozycji). Następnie napisz program pozwalający użytkownikowi na przeszukiwanie kolekcji na podstawie podawanych parametrów. Wynikiem powinna być lista dostępnych pojazdów.
7. * Napisz program, który porówna wydajność działania operacji na kolekcjach typu ArrayList, LinkedList, HashSet, TreeSet oraz HashMap, TreeMap.

Pamiętaj o wykorzystaniu repozytorium kodu Git! * Dla chętnych.

Spotkanie #7

Wprowadzenie do języka Java

Rozkład jazdy

Zanim zaczniemy proszę zaktualizować projekt: **java24gda_intro !**

09:00 - powtórka w formie testu
09:30 - mapy
10:30 - przerwa krótka
10:40 - typy generyczne
12:30 - przerwa długa
13:00 - podsumowanie testu
13:30 - IO, new IO
14:30 - przerwa krótka
14:45 - IO, new IO cd
16:00 - koniec zajęć :)

Instrukcja aktualizacji projektu z GitHub: https://goo.gl/m3BHMr

Szybka powtórka

TEST

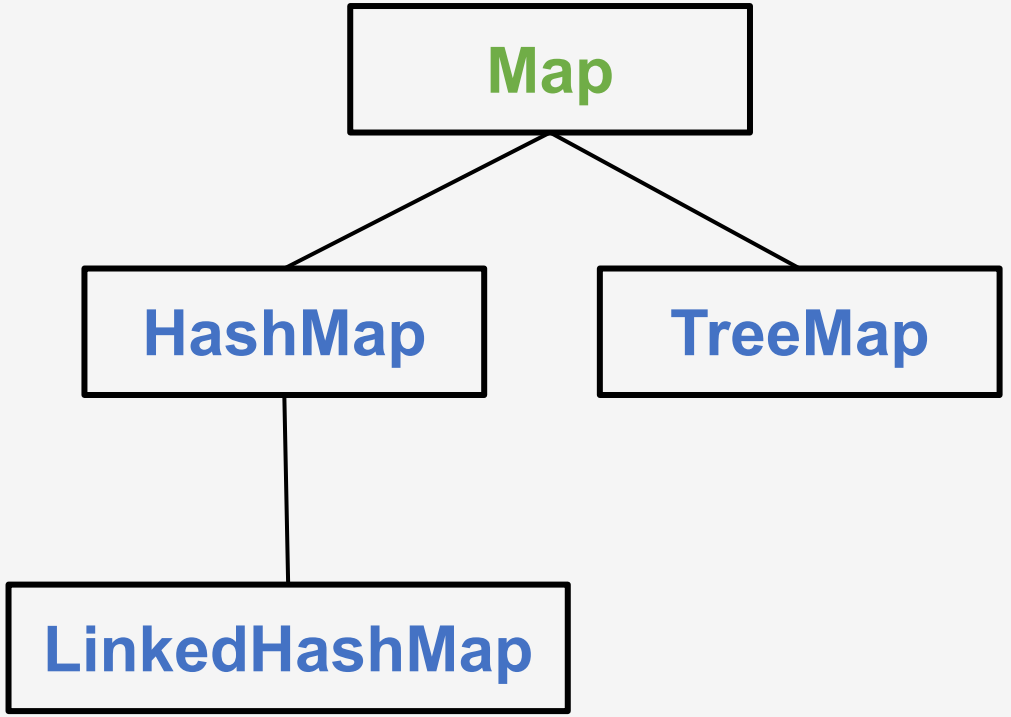
Mapy

java.util.Map

Map (słownik, tablica asocjacyjna)
kolekcja danych, w której przechowywane są pary danych: klucz i wartość. Klucze nie mogą się powtarzać. Umożliwia przypisanie do klucza konkretną wartość - jak w słowniku gdzie np. słowo po polsku to klucz a po angielsku to wartość.

O tym czy dwa klucze są takie same decyduje:

- w przypadku **HashMap** i **LinkedHashMap** metoda ***equals()***
- w przypadku **TreeMap** metoda ***compareTo()***/***compare()***



Rodzaje map

- java.util.**HashMap**
  Mapa nieposortowana, gdzie kolejność iteracji jest nieokreślona.
  Ogólna implementacja, którą możemy utożsamiać ze słownikiem.

- java.util.**LinkedHashMap**
  Dziedziczy po HashMap, ale zapewnia taką samą kolejność elementów w momencie iterowania po kolekcji.

- java.util.**TreeMap**
  Mapa posortowana według kluczy w sposób naturalny gdy elementy implementują Comparable. Sortowanie zapewnione jest już na etapie dodawania nowego elementu.

Zadania

#collections

1. Stwórz mapę, która jako klucze będzie zawierała obiekty ***Integer***, a jako wartości obiekty typu ***String***. Wypisz na ekran rozmiar mapy oraz wszystkie wartości (klucz - wartość).
2. Policz w mapie z pkt 1 (za pomocą pętli) ilość kluczy, które mają wartość mniejszą od zera oraz takich które mają wartość większą lub równą zeru. Wypisz wyniki na ekran.
3. Z mapy stworzonej w pkt 1 wyciągnij wszystkie klucze, które mają wartość mniejszą od zera i dodaj je do nowej listy obiektów ***Integer***. Następnie usuń z mapy wszystkie wartości, których klucze znajdują się w liście. Wypisz wszystkie pozostałe wartości mapy (klucz - wartość) na ekran.
4. Stwórz metodę, która jako parametr przyjmie listę obiektów ***pl.sda.collections.Product*** i zwróci mapę gdzie kluczem będzie id produktu a wartością obiekt klasy ***Product.***
   a. zmień implementację mapy tak żeby kolejność produktów w mapie odpowiadała kolejności z listy wejściowej
   b. zmień implementację mapy tak żeby kolejność produktów w mapie odpowiadała kolejności idków produktów
5. * Napisz klasę, która będzie przechowywała mapę słów (obiektów typu ***String***), gdzie kluczem będzie pierwsza litera słowa a wartością zbiór unikalnych słów podanych przez użytkownika. Klasa powinna mieć metodę do podania pojedynczego słowa do klasy, metodę do pobrania zbioru słów na podaną literę i metodę do wyciągnięcia liczności słów dla podanej litery.

Pamiętaj o wykorzystaniu repozytorium kodu Git! * Dla chętnych.

Typy generyczne

“Pudełkowy” problem

OrangeBox

AppleBox

StrawberryBox

“Pudełkowy” problem - rozwiązanie

Box<T>

```
public class Box<T> {
    private T item;

    public Box(T item) {
        this.item = item;
    }

    public T getItem() {
        return item;
    }
}
```

Typy generyczne

Typami generycznymi mogą być tylko obiekty (typy referencyjne), nie mogą nimi być typy prymitywne. Stąd:

- Integer - OK
- int - BAD

Iterable<E>
Collection<E>
List<E>
Set<E>
Queue<E>
Map<K, V>
...

Konwencja nazewnicza dla nazw parametrów:

E - Element (Java Collections)
K - Key
N - Number
T - Type
V - Value
S,U,V etc. - 2nd, 3rd, 4th types

Typy generyczne

- >= Java 1.5
- parametryzacja klas, metod, interfejsów
- podanie typu dopiero w miejscu użycia
- zwiększenie uniwersalności klasy
- unikanie niepotrzebnego rzutowania

Typy generyczne

```
public class Vehicle {
    private Object obj;

    public Object get() {
        return obj;
    }

    public void set(Object obj) {
        this.obj = obj;
    }
}
```

Typy generyczne

```
public class Vehicle<T> {
    private T obj;

    public T get() {
        return obj;
    }

    public void set(T obj) {
        this.obj = obj;
    }
}
```

Typy generyczne

```
Car car = new Car();
Vehicle<Car> vehicle = new Vehicle<>();
vehicle.set(car);
```

Typy generyczne

```
class Name<T1, T2, ..., Tn>
interface Name<T1, T2, ..., Tn>
T methodName()
<T> methodName(List<T> list)
```

Typy generyczne

ograniczenia typów (bounded type parameters)

= generyki ograniczone do konkretnych typów

- ograniczenie górne (upper bound)

public class Vehicle<T extends Car>

- ograniczenie górne wielokrotne

public class Vehicle<T extends Car & Electric & ..>

klasa interfejsy..

Istnieje jeszcze ograniczenie dolne, ale raczej nigdy się z tym nie spotkasz ;)

Zadania

#generics

Zadania

#generics

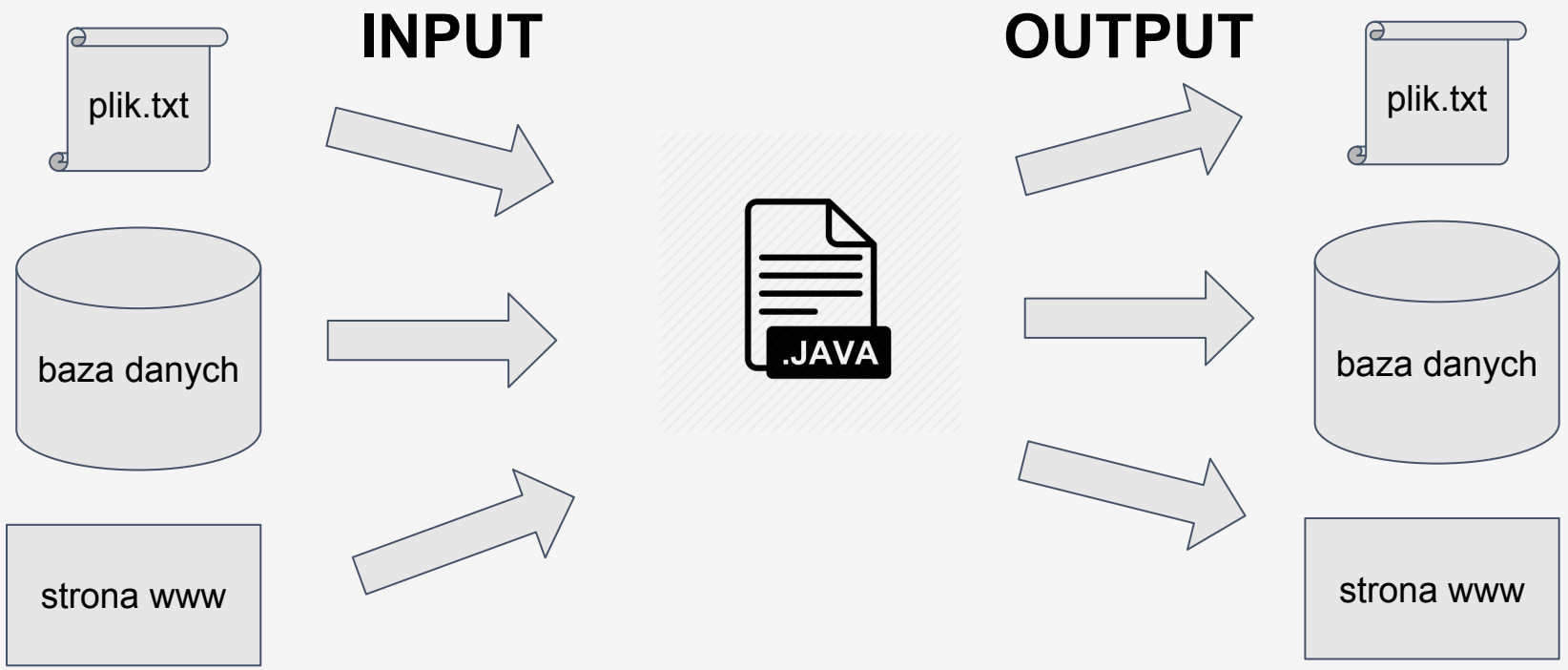
1. Stwórz metodę, która przyjmie listę obiektów typu Integer i wyświeli na ekran tylko liczby większe od 10.
2. Stwórz metodę, która jako parametry przyjmie: kolekcję (Set) zawierającą zestaw fraz oraz pojedynczy wyraz którego szukamy w kolekcji. Metoda zwróci ilość wystąpień szukanego wyrazu w podanej kolekcji. Weź pod uwagę sytuację gdy szukany wyraz jest częścią frazy z kolekcji.
3. Zmień metodę z pkt.1 tak żeby przyjmowała listę dowolnych obiektów dziedziczących po klasie Number. Sprawdź czy metoda będzie działała z: List<Double> i List<Integer>
4. Utwórz klasę, która pozwoli ustawić parę obiektów różnego typu. Sygnatura klasy powinna zawierać dwa generyki, a następnie konstruktor pozwalający zainicjalizować obiekt klasy z dwoma instancjami obiektów.
5. Stwórz klasę, która będzie zawierała mapę gdzie klucz = String, a wartość = Integer. Klasa powinna mieć metody: addWord() i getWordCount() - które dodadzą słowo i zwrócą ilość wcześniej dodanych słów lub zero jeżeli słowo nie występuje.
6. Ogranicz możliwość podania dowolnego typu obiektów dla klasy z pkt. 4.
7. * Utwórz klasę Forest która będzie zawierała zbiór różnego rodzajów drzew (liściastych i iglastych - dodaj odpowiednie klasy). Dodaj metody, które zwrócą wszystkie drzewa, tylko liściaste albo tylko iglaste. Dodaj metodę, która zwróci drzewa starsze niż podany parametr.

Pamiętaj o wykorzystaniu repozytorium kodu Git! * Dla chętnych.

I/O, new I/O

I/O = Input/Output

I/O Input/Output
oznacza programowanie operacji wejścia-wyjścia.
Wejście(ang. input) oznacza zasoby (pliki, zasoby internetowe, bazy danych itp) które dostarczają danych do programu, a **wyjście(ang. output)** oznacza zasoby do których program zapisuje dane.



Obsługa plików

do Java 7	od Java 7
Java IO API java.io.File (JDK1.0) ...	Java NIO2 API java.nio.file.Paths java.nio.file.Path java.nio.file.Files ...

Obsługa plików - java.io.File

```
File folder = new File("C:\\test");
File file = new File(folder, "hello.txt");
```

obiekt klasy File reprezentuje abstrakcyjną ścieżkę do pliku lub katalogu - nawet gdy plik/katalog jeszcze nie istnieje!

to samo co wyżej ale z podaniem dwóch parametrów: katalog i plik

```
System.out.println("file.exists() = " + file.exists());
```

metoda exists() sprawdza czy ścieżka zapisana w obiekcie klasy File wskazuje na istniejący plik

```
File parentFile = file.getParentFile();
boolean success = parentFile.mkdirs();
System.out.println("pFile.exists() = " + parentFile.exists());
```

w tym miejscu tworzone są nadkatalogi i sam katalog reprezentowany przez zmienną **parentFile**

```
try {
    boolean success = file.createNewFile();
} catch (IOException e) {...}
```

dopiero w tym miejscu tworzymy plik. Jeżeli metoda zwróci true oznacza to że plik został stworzony

Klasa java.io.File

ważne metody

- exists(): boolean; czy plik/katalog istnieje w rzeczywistości

- mkdir(): boolean; tworzy katalog na który wskazuje obiekt File
- mkdirs(): boolean; tworzy katalog i wszystkie nieistniejące nadkatalogi
- createNewFile(): boolean; tworzy nowy, pusty plik
- renameTo(): boolean; zmienia nazwę lub przenosi plik/katalog w inne miejsce
- delete(): boolean; usuwa plik lub katalog (musi być pusty!)

- isDirectory(): boolean; czy ścieżka wskazuje na katalog
- isFile(): boolean; czy ścieżka wskazuje na plik

- getPath(): String; zwraca ścieżkę podaną w konstruktorze
- getAbsolutePath(): String; zwraca bezwzględną ścieżkę do pliku lub katalogu
- getCanonicalPath(): String; zwraca bezwzględną, rzeczywistą ścieżkę (bez . i ..)

- listFiles(): File[]; zwraca listę plików lub katalogów zapisanych w katalogu

Przykłady w kodzie: pl.sda.io.OldFilesSamples

Obsługa plików - Java NIO 2

```
Path folder = Paths.get("C:\\test");
Path file = folder.resolve("hello.txt");

System.out.println("exists = " + Files.exists(file));

Path parentFile = file.getParent();
try {
    Files.createDirectories(parentFile);

    Files.createFile(file);
} catch (IOException e) {...}
```

Path reprezentuje abstrakcyjną ścieżkę do pliku lub katalogu, Paths to klasa-fabryka służąca do tworzenia obiektów klasy Path

tworzymy nową ścieżkę na bazie obiektu **folder**, dodając nazwę pliku

klasa Files składa się z zestawu metod statycznych do operacji na plikach i katalogach

w tym miejscu tworzone są nadkatalogi i sam katalog reprezentowany przez zmienną **parentFile**

dopiero w tym miejscu tworzymy plik. Jeżeli metoda zwróci true oznacza to że plik został utworzony

Klasy Path, Paths, Files

ważne metody

- Paths.get(): Paths; tworzy nową ścieżkę do pliku lub katalogu
- Files.exists(): boolean; czy plik / katalog istnieje w rzeczywistości

- Files.createDirectory(): Path; tworzy katalog na który wskazuje obiekt Path
- Files.createDirectories(): Path; tworzy katalog i wszystkie nieistniejące nadkatalogi
- Files.createFile(): Path; tworzy nowy, pusty plik
- Files.move(): Path; zmienia nazwę lub przenosi plik / katalog w inne miejsce
- Files.copy(): Path; kopiuje plik / katalog w inne miejsce
- Files.deleteIfExists(): boolean; usuwa plik lub katalog (musi być pusty!)

- Files.isDirectory(): boolean; czy ścieżka wskazuje na katalog
- Files.isRegularFile(): boolean; czy ścieżka wskazuje na plik

- Path.toString(): String; zwraca ścieżkę jaka została podana przy tworzeniu obiektu
- Path.toAbsolutePath(): String; zwraca bezwzględną ścieżkę do pliku lub katalogu
- Path.toRealPath(): String; zwraca bezwzględną, rzeczywistą ścieżkę (bez . i ..)

- Files.newDirectoryStream(): DirectoryStream<Path>; zwraca listę plików / katalogów

Przykłady w kodzie: pl.sda.io.FilesExamples

Strumienie

Strumień danych

oznacza ciąg danych (“strumień”), do którego dane mogą być dodawane (OutputStream) i z którego dane mogą być pobierane (InputStream).

- Strumień związany jest ze źródłem lub odbiornikiem danych (np. plik, pamięć, URL, gniazdo itp.)
- Strumień służy do zapisywania-odczytywania informacji (dowolnych danych), dlatego mówimy o dwóch typach:
  - strumieniach wejściowych (input)
  - strumieniach wyjściowych (output)
- Strumienie dodatkowo dzielą się na dwa rozłączne typy:
  - strumienie bajtowe - gdzie “atomem” (podstawową porcją danych) jest bajt (**InputStream** i **OutputStream**)
  - strumienie znakowe - gdzie “atomem” są znaki unikodu, po dwa bajty każdy(**Reader** i **Writer**)
- Java tworzy domyślnie 3 strumienie:
  - System.in
  - System.out
  - System.err

try-with-resources

Zamiast:

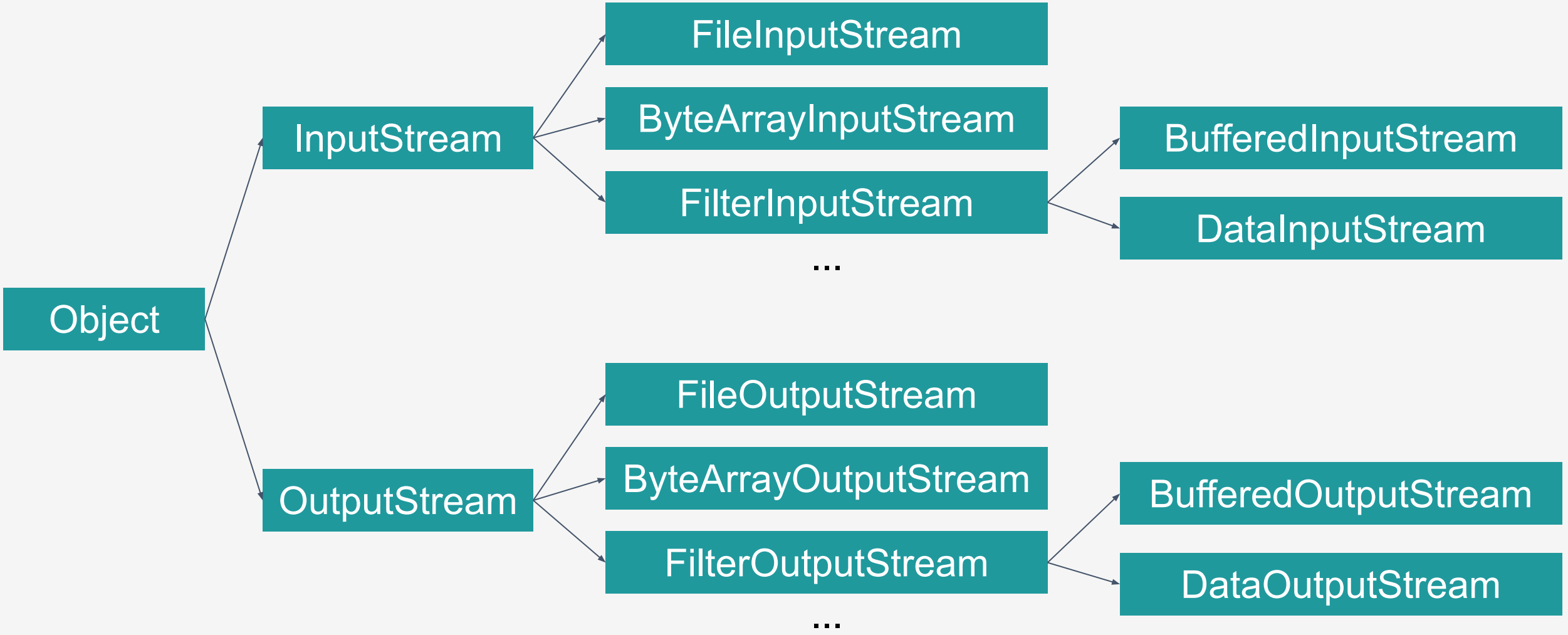
```
FileReader reader = null;
try {
      reader = new FileReader("test.txt");
      ...
} catch (IOException e) {
      e.printStackTrace();
} finally {
      try {
            if (reader != null) {
                  reader.close();
            }
      } catch (IOException e) {
            e.printStackTrace();
      }
}
```

Można:

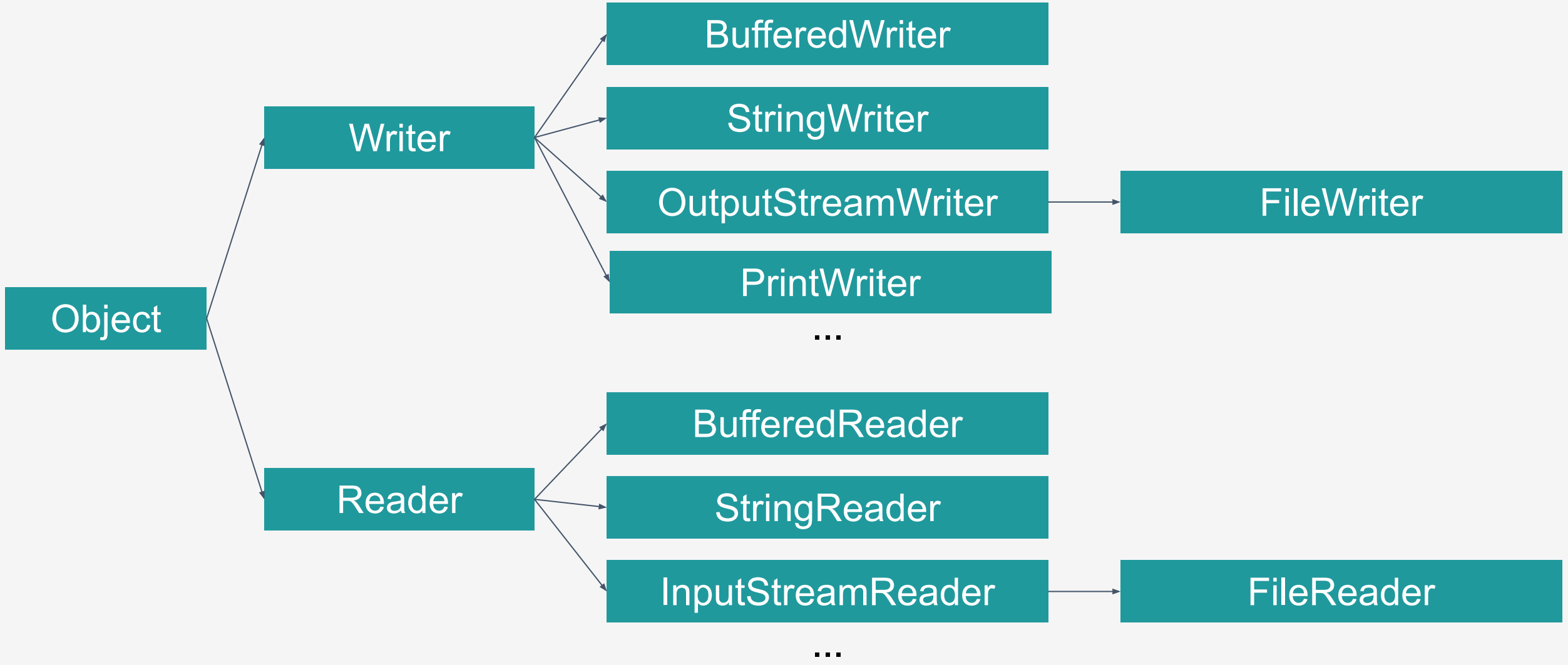
```
try(FileReader reader = new FileReader("test.txt")) {
      reader.read();
      ...
} catch (IOException e) {
      e.printStackTrace();
}
```

dzięki **try-with-resource** nie musimy zamykać sami strumienia danych w bloku finally - oszczędzamy parę linijek kodu + mamy pewność że zasób będzie zawsze zamknięty tuż po wykonaniu całego kodu z bloku try

I/O, new I/O


Object
InputStream
FileInputStream
ByteArrayInputStream
FilterInputStream
...
BufferedInputStream
DataInputStream
OutputStream
FileOutputStream
ByteArrayOutputStream
FilterOutputStream
...
BufferedOutputStream
DataOutputStream

I/O, new I/O



Przykłady w kodzie: pl.sda.io.IoExamples

Zadania

#io

Zadania

#io

1. * W dowolnym wybranym przez Ciebie katalogu stwórz za pomocą klasy java.io.File dwa podkatalogi i dwa pliki. W każdym z podkatalogów stwórz kolejne dwa podkatalogi i dwa kolejne pliki. Poćwicz operacje na plikach: przenoszenie, usuwanie, wyświetlanie, kopiowanie.
2. Wykonaj zadanie z pkt 1 za pomocą Java NIO2 API.
3. Napisz kod, który zapisze do pliku Twoje imię i nazwisko.
4. Napisz kod, który odczyta plik tekstowy i wyświetli jego zawartość na konsoli.
5. Napisz kod, który skopiuje zawartość dowolnego pliku (tekstowego lub binarnego) do drugiego pliku .
6. Napisz kod, który zapisze do pliku kolekcję obiektów typu Student. Zadbaj o własny format zapisu danych w postaci tekstowej. Każdy obiekt powinien być zapisany w nowej linii.
7. Napisz kod, który wczyta dane z pliku utworzonego w punkcie 6 oraz utworzy z nich kolekcję obiektów typu Student.
8. * Napisz kod, który przeanalizuje podany plik tekstowy oraz utworzy statystykę występowania poszczególnych słów w odczytanym tekście. Zadbaj o odpowiednią obsługę wyjątków. Daj możliwość podania separatora poszczególnych słów.
9. * Napisz kod, który stworzy:
   a. 10 katalogów o nazwach 0, 100, 200, …, 900.
   b. w każdym z katalogów stwórz 10 plików z nazwami odpowiadającymi poszczególnym dziesiątkom liczb - czyli np. w katalogu 0 będą się znajdować pliki: 1-10.txt, 11-20.txt, … a w katalogu 200: 200-210.txt, 211-220.txt.
   c. w każdym pliku zapisz sumę liczb z zakresu który wskazuje nazwa pliku, np. w pliku 0-10.txt powinna się pojawić liczba: 55.

Pamiętaj o wykorzystaniu repozytorium kodu Git! * Dla chętnych.

Spotkanie #8

Wprowadzenie do języka Java

Szybka powtórka

- kolekcja Map
- typy generyczne
- I/O, new I/O

Rozkład jazdy

- programowanie funkcyjne: Optional, Lambda Expression, Streams
- elementy biblioteki Swing
- JavaFX
- programowanie współbieżne i równoległe
- adnotacje

Zanim zaczniemy proszę zaktualizować projekt: **java24gda_intro !**

Instrukcja aktualizacji projektu z GitHub: https://goo.gl/m3BHMr

Programowanie funkcyjne: Optional, Lambda Expression, Streams

Programowanie funkcyjne?

Programowanie funkcyjne?

Zamiast operować na stanach obiektów, definiujemy co faktycznie chcemy zrobić = co ma być osiągnięte.

Programowanie funkcyjne: Optional, Lambda Expression, Streams

Lambda Expression (->) - obiekty, które zawierają fragment kodu (**funkcję bez nazwy**), a także atrybuty i parametry pozwalające na operowanie funkcji.

Poszczególne fragmenty kodu (funkcje = lambdy) możemy traktować jak pełnoprawne obiekty, które mogą być przekazywane innym funkcjom i zwracane z innych funkcji.

Programowanie funkcyjne: Optional, Lambda Expression, Streams

```
List<String> NAMES = Arrays.asList(
    "Jan", "Emilia", "Tomasz", "Michał", "Anna");

for (String name : NAMES) {
  System.out.println(name);
}
```

vs

```
NAMES.stream()
  .forEach(name -> System.out.println(name));
```

Programowanie funkcyjne:
Optional, Lambda Expression, Streams

java.util.stream.Stream - strumień reprezentuje sekwencję elementów i pozwala wykonywać na nich różne operacje. Operacje te mogą być pośrednie (możemy je układać w łańcuchy metod) oraz końcowe (zwracające wynik lub nie).

```
collection.stream()
    .filter(...)
    .map(...)
    .sorted()
```

Programowanie funkcyjne: Optional, Lambda Expression, Streams

Niektóre operacje (np. .filter, .map) na strumieniach mogą przyjmować wyrażenia lambda.

Funkcje muszą być jednak:

- nieinterferujce (**non-interfering**) - funkcja nie modyfikuje podstawowego źródła danych do strumienia (nie usuwa, nie edytuje i nie zmienia elementów ze strumieniowanej kolekcji)
- bezstanowe (**stateless**) - wynik funkcji jest deterministyczny = nie zależy od żadnej zmiennej, którą można zmieniać (mutable) albo stanu z zewnętrznego źródła, który może się zmieniać w trakcie jej wykonywania

Programowanie funkcyjne:
Optional, Lambda Expression, Streams

```
List<String> NAMES = Arrays.asList(
    "Jan", "Emilia", "Tomasz", "Michał", "Anna");

Optional<String> jan = NAMES.stream()
    .filter(name -> "Jan".equals(name))
    .findAny();

System.out.println(jan.isPresent());
```

Zadania

#functional

Zadania

#functional

1. Zapoznaj się z dostępnymi metodami w interfejsie **Stream**.
2. Utwórz kolekcję i wypełnij ją losowymi liczbami całkowitymi. Zsumuj wszystkie wartości znajdujące się w kolekcji za pomocą użycia interfejsu stream(). * Znajdź co najmniej dwa sposoby na realizację tego zadania.
3. Na podstawie kolekcji z liczbami całkowitymi napisz kod z wykorzystaniem interfejsu .stream(), który po wykonaniu zwróci nam kolekcję z liczbami pomnożonymi przez 2.
4. Utwórz kolekcję zawierającą obiekty typu **Person**. Napisz metodę do przeszukiwania kolekcji (po nazwie) z wykorzystaniem interfejsu stream().
   * Napisz kod umożliwiający wprowadzanie danych do przeszukiwania kolekcji z poziomu konsoli. Obsłuż możliwe do wystąpienia wyjątki oraz zadbaj by Twój program zawsze wyświetlał odpowiednie komunikaty dla użytkownika.
5. Wykorzystaj kolekcję z obiektami typu **Person** i używając interfejsu stream(), wyciągnij wszystkie nazwy, posortuj, zmień wszystkie litery na wielkie, ogranicz do 5 elementów i na koniec utwórz jeden String zawierający przetworzone nazwy scalone ze sobą (za pomocą przecinka).

Pamiętaj o wykorzystaniu repozytorium kodu Git! * Dla chętnych.

Elementy biblioteki Swing

Elementy biblioteki Swing

Swing to zestaw klas umożliwiający tworzenie aplikacji posiadających **graficzny interfejs użytkownika** (GUI).

Swing bazuje na architekturze **Model-View-Controller** (MVC), gdzie:

- **model** - określa dane związane z komponentem lub stany komponentu
- **widok** (view) - określa wizualną reprezentację danych lub stanów
- **sterownik** (controller) - zapewnia interakcję użytkownika z widokiem i wynikające stąd modyfikacje modelu

Elementy biblioteki Swing

javax.swing.*

- **JFrame** (służy do tworzenia okien)
- **JMenuBar** (służy do tworzenia menu w oknach)
- **JMenuItem** (tworzy element menu)
- **JTextField** (pole do wpisywania tekstu)
- **JList** (tabelka z listą elementów)
- **WindowConstant** (zbiór pewnych stałych potrzebnych do ustawienia właściwości tworzonych obiektów)
- ...

Elementy biblioteki Swing

Małe demo - #swing

- https://beginnersbook.com/2015/07/java-swing-tutorial/
- http://zetcode.com/tutorials/javaswingtutorial/
- https://www.javatpoint.com/java-swing
- https://www.udemy.com/java-swing-complete/
- https://www.jetbrains.com/help/idea/swing-designing-gui.html

JavaFX

JavaFX - Java (7?) 8++

- (F)XML
- CSS
- MVC (Model - View - Controller)
- Scene Builder
- wsparcie multimediów

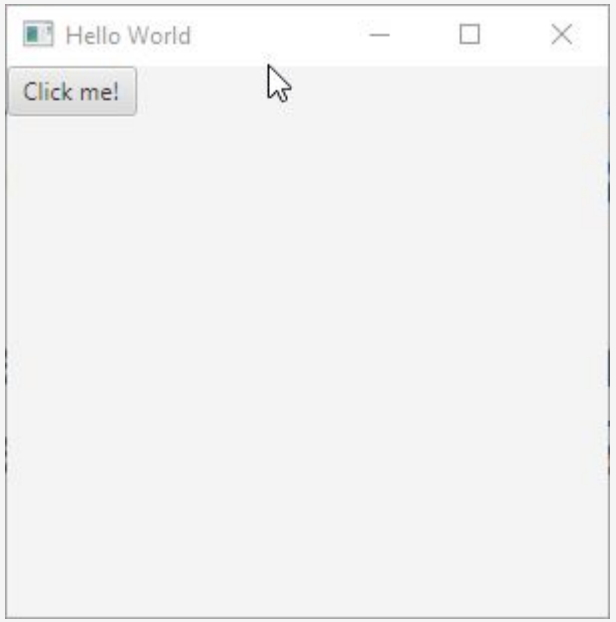
JavaFX - od Java8

- http://code.makery.ch/library/javafx-8-tutorial/
- http://www.tutorialspoint.com/javafx/
- https://www.oracle.com/technetwork/java/javase/overview/javafx-samples-2158687.html

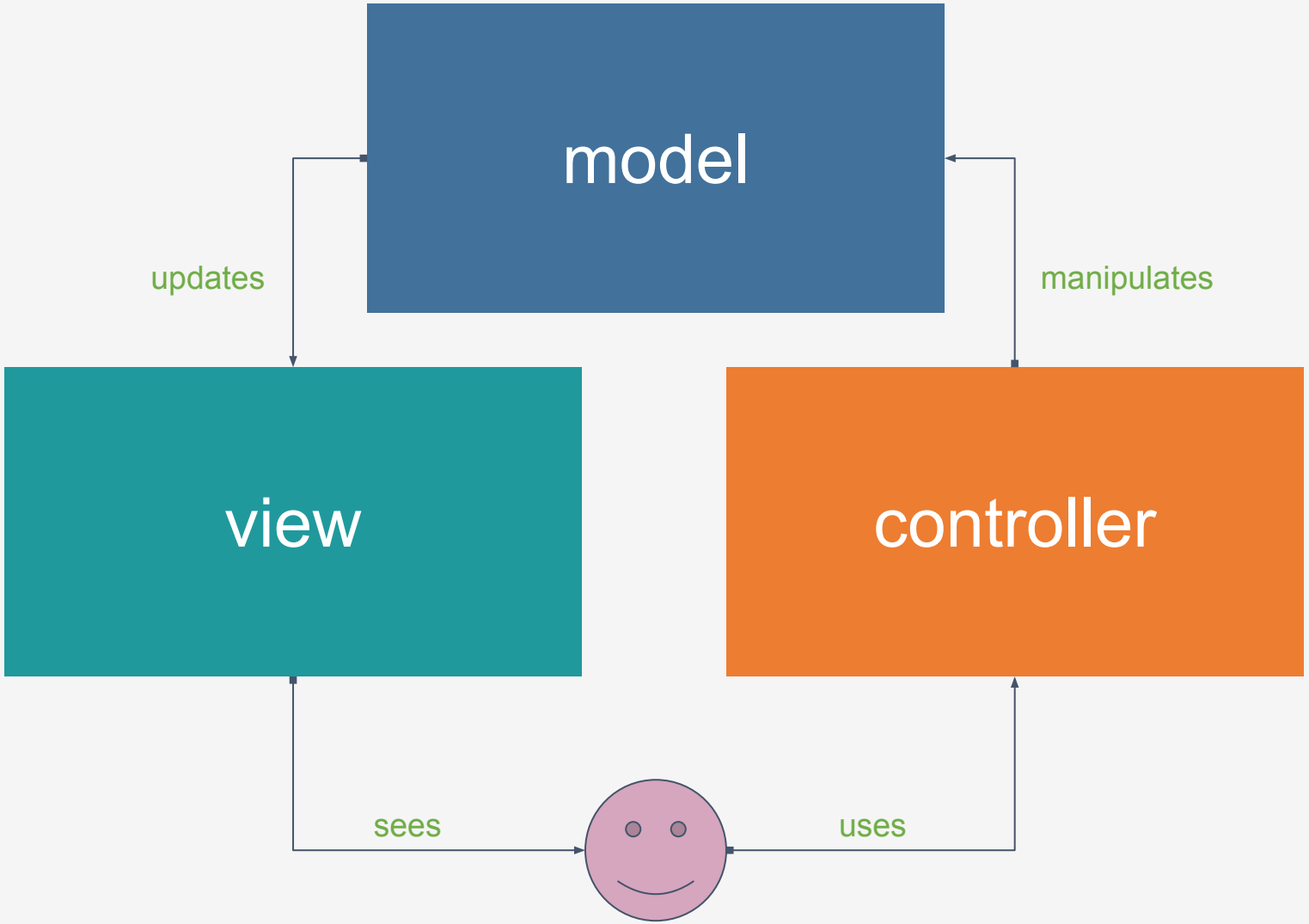
- https://www.youtube.com/watch?v=FLkOX4Eez6o&list=PL6gx4Cwl9DGBzfXLWLSYVy8EbTdpGbUIG

JavaFX - FXML

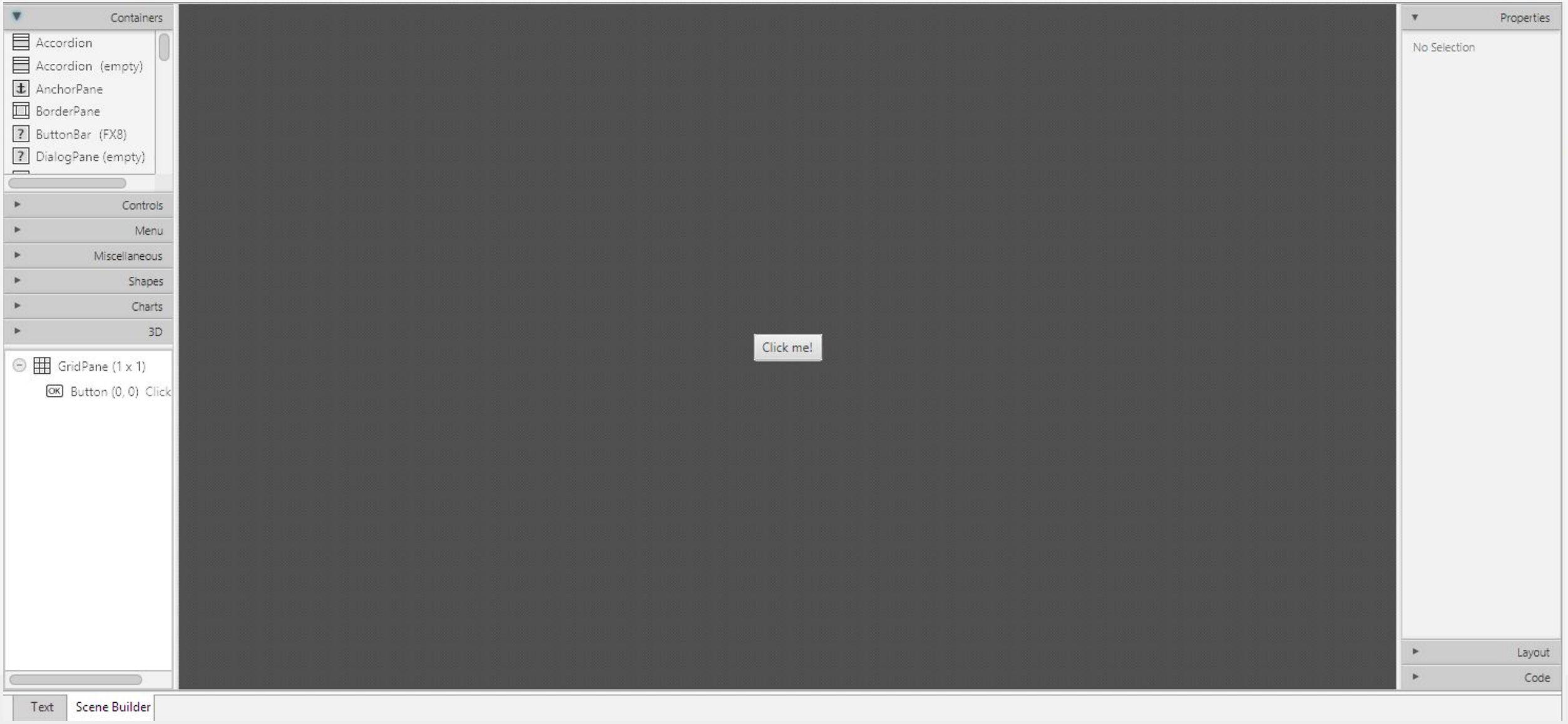
```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.layout.GridPane?>
<?import javafx.scene.control.Button?>

<GridPane>
    <Button>Click me!</Button>
</GridPane>
```



JavaFX - MVC -> Model & View & Controller



JavaFX - Scene Builder



Zadania

#javafx

Zadania

#javafx

1. Utwórz aplikację, która zaszyfruje podany ciąg znaków algorytmem base64 oraz wyświetli wynik w postaci łatwej do skopiowania. Wykorzystaj klasę Base64 do zakodowania tekstu.

2. Utwórz w pełni działający kalkulator dla działań: +, -, * oraz /.

3. Utwórz program, który wymaga zalogowania się po starcie za pomocą nazwy użytkownika i hasła zdefiniowanych po stronie aplikacji. Po pomyślnym zalogowaniu aplikacja wyświetli komunikat.

4. * Spójrz na Aleksandrię ;)

Pamiętaj o wykorzystaniu repozytorium kodu Git! * Dla chętnych.

Programowanie współbieżne i równoległe

Programowanie współbieżne i równoległe

Programowanie **współbieżne (concurrent)** – jeden proces rozpoczyna się przed zakończeniem drugiego, co oznacza wykonywanie kilku zadań przez procesor w tym samym czasie poprzez przeplatanie wątków.

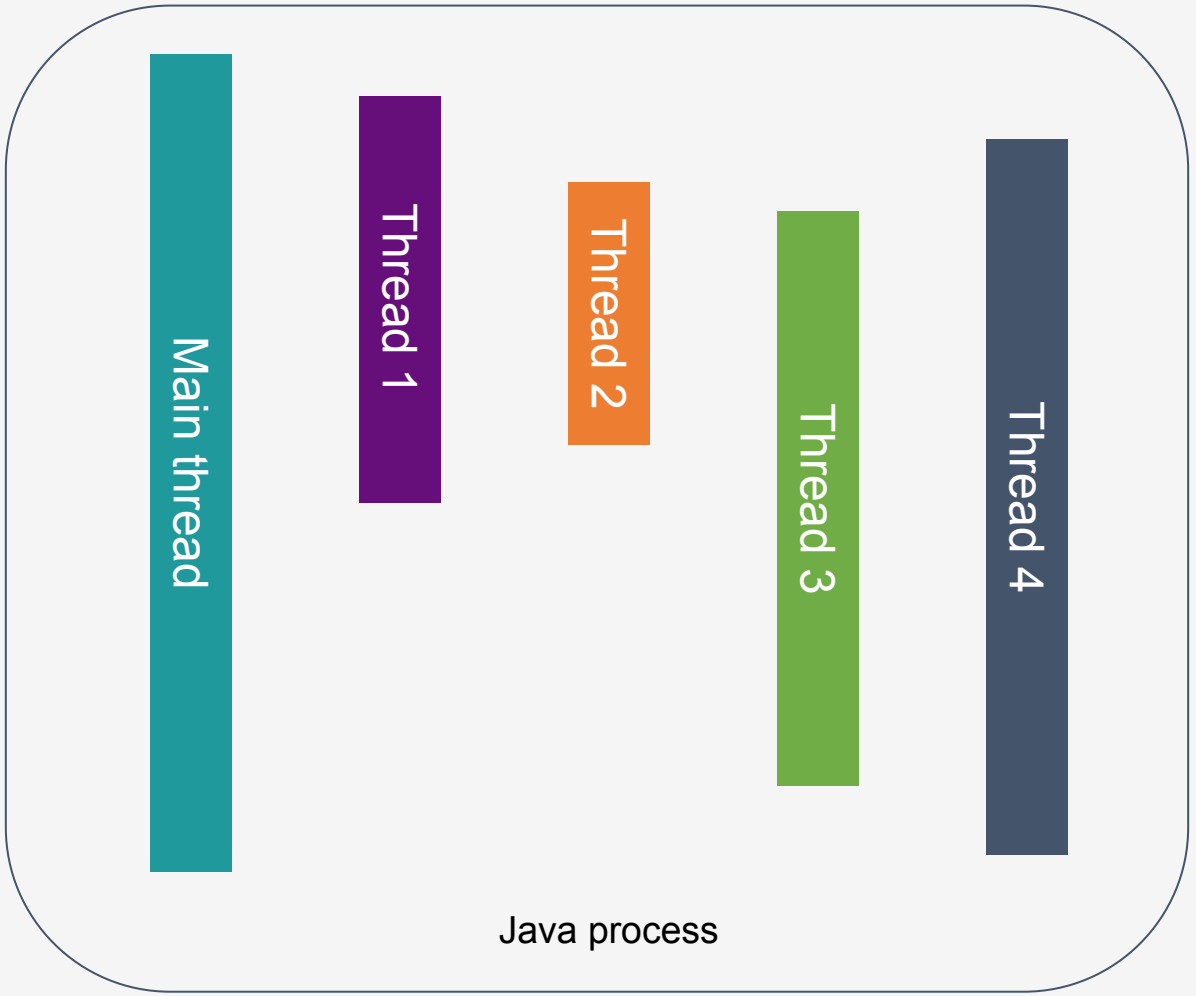
Programowanie **równoległe (parallel)** – jednoczesne wykonywanie wielu operacji w trakcie rozwiązywania jednego problemu.

Programowanie współbieżne i równoległe

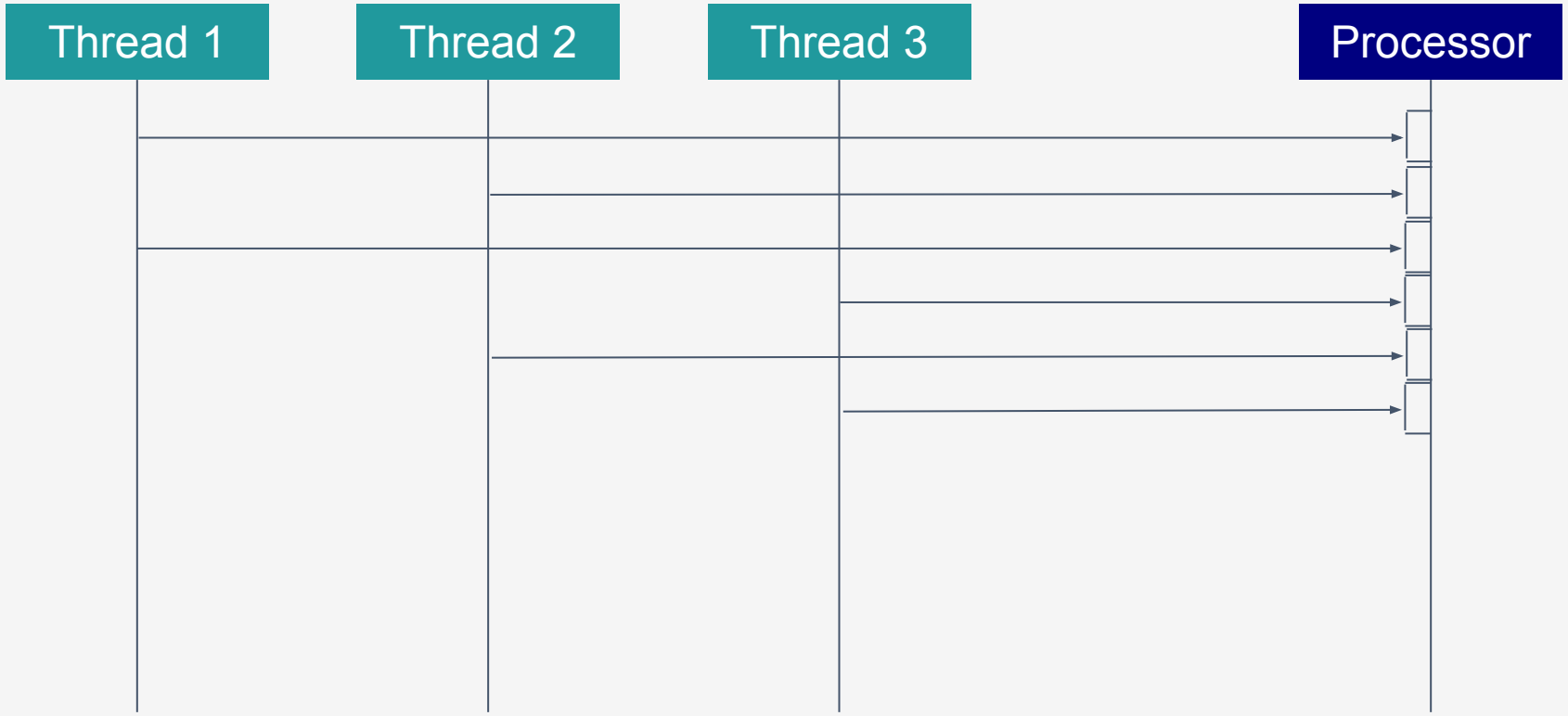
Proces - wykonujący się program wraz z zasobami dynamicznie przydzielanymi mu przez system (np. pamięcią operacyjną, zasobami plikowymi). Każdy proces ma własną przestrzeń adresową.

Wątek - sekwencja działań, która wykonuje się w kontekście danego procesu (programu).

Programowanie współbieżne i równoległe



Programowanie współbieżne i równoległe



Programowanie współbieżne i równoległe

Równoległość działania wątków osiągana jest przez mechanizm przydzielania czasu procesora poszczególnym wykonującym się wątkom. Każdy wątek uzyskuje dostęp do procesora na krótki czas (kwant czasu), po czym „oddaje procesor" innemu wątkowi.

Zmiana wątku wykonywanego przez procesor może dokonywać się na zasadzie:

współpracy (cooperative multitasking) - wątek sam decyduje, kiedy oddać czas procesora innym wątkom

wywłaszczania (pre-emptive multitasking) - o dostępie wątków do procesora decyduje systemowy zarządca wątków, który przydziela wątkowi kwant czasu, po upływie którego odsuwa wątek i przydziela kolejny kwant czasu innemu wątkowi

Programowanie współbieżne i równoległe

java.lang.Thread - klasa odpowiedzialna za uruchamianie i zarządzanie wątkami

java.lang.Runnable - interfejs zawierający metodę **run()**, która definiuje co powinien wykonać dany wątek implementujący

Najbardziej popularne metody tworzenia nowego wątku:

- poprzez **dziedziczenie klasy Thread** (implementuje Runnable)
- poprzez **implementację interfejsu Runnable** (lepsze rozwiązanie niż dziedziczenie klasy Thread) oraz uruchomienie za pomocą klasy Thread
- Runnable / Callable + **ExecutorService**

java.util.concurrent.ExecutorService

Interfejs ten (a właściwie jego implementacje) pozwala nam tworzyć pule wątków, delegować im zadania, tworzyć zadania cykliczne, kończyć pracę wątków w kontrolowany sposób i wiele więcej.

Zadania

#concurrent

Zadania

#concurrent

1. Zapoznaj się z metodami klas Thread oraz ExecutorService. Co możemy dzięki nim osiągnąć?
2. Utwórz własne implementacje klas CustomThread oraz CustomRunnable. Następnie przetestuj ich działanie.
3. Napisz klasę implementującą interfejs Runnable. Klasa powinna wykonywać w metodzie run() kod, którego wynikiem będzie lista wszystkich całkowitych dzielników podanej liczby naturalnej. Uruchom klasę z wykorzystaniem ExecutorService i przetestuj jej działanie. Sprawdź czas wykonania swojego kodu.
4. * Napisz klasę **Counter**, która zwiększać będzie swój licznik przy każdym wykonaniu metody **increase()**. Do klasy dodaj również metodę **get()**, która zwróci aktualny stan licznika. Napisz klasę **CounterRunnable** implementującą interfejs Runnable, dla której konstruktor jako argument przyjmować będzie instancję klasy **Counter**, a metoda **run()** wywoła metody increase() oraz get() i wyświetli na konsoli rezultat. Uruchom pulę wątków i obserwuj jakie otrzymujesz wartości. Przetestuj działanie dla różnej puli wątków. Czy rezultaty są takie jak oczekiwane przez Ciebie?

Pamiętaj o wykorzystaniu repozytorium kodu Git! * Dla chętnych.

Adnotacje

Adnotacje

Jakie poznaliście?

Adnotacje

Java 5+

Dodatkowe dane (**metadane** - dane do danych) w kodzie, które służą kompilatorowi lub dodatkowym narzędziom (np. IDE) do analizy naszego kodu.

Adnotacje nie wpływają na wykonanie naszego kodu!

Adnotacje
zastosowania

- poprawa niezawodności programowania
- generacja dodatkowych składowych programu
- określenie sposobu funkcjonowania programu w fazie wykonania
- cele konfiguracyjne
- generowanie plików pomocniczych

Adnotacje
rodzaje

predefiniowane	użytkownika
@Deprecated @Override @SuppressWarnings @Target @Retention @Inherited @Documented ...	• stosowane na etapie kompilacji (przetwarzane przez narzędzia) • stosowane w fazie wykonania (za pomocą mechanizmów refleksji)

Adnotacje
@Target

- ElementType.ANNOTATION_TYPE
- ElementType.CONSTRUCTOR
- ElementType.FIELD
- ElementType.LOCAL_VARIABLE
- ElementType.METHOD
- ElementType.PACKAGE
- ElementType.PARAMETER
- ElementType.TYPE

Adnotacje bez @Target mają zastosowanie wszędzie!

Adnotacje
@Retention

- RetentionPolicy.SOURCE
- RetentionPolicy.CLASS
- RetentionPolicy.RUNTIME

Adnotacje

@Documented

- dokumentacja działania adnotacji ma być włączona do dokumentacji wszystkich oznaczanych przez nią elementów

@Inherited

- oznaczana przez nią adnotacja ma być dziedziczona przez podklasy

Adnotacje

```
public @interface Annotation {
  String name();
  Type type() default Type.Custom;
}
```

Java Reflection (refleksja)

Proces, który pozwala m.in. na zdobywanie informacji o klasach w trakcie wykonania programu.

Zadania

#annotations

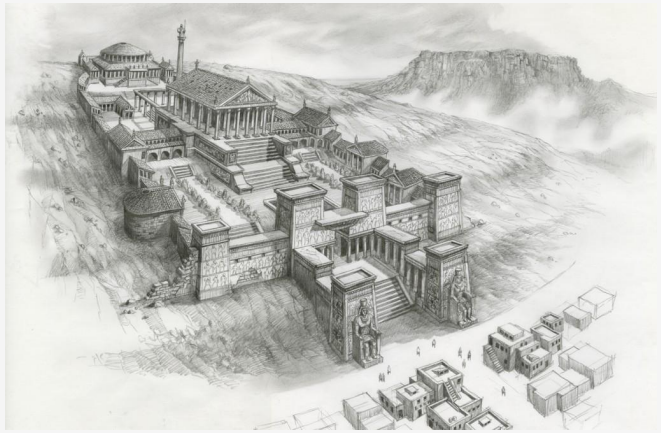
Zadania

#annotations

1. Napisz własną adnotację o nazwie **@Info**, która:
   a. będzie mogła być dodawana nad klasą i metodą
   b. powinna działać razem z aplikacją
   c. powinna przyjmować dodatkowe dane opisujące klasę / metodę - autor, data, opis
2. Utwórz klasy **School** oraz **Student** i dodaj do nich kilka pól oraz metod.
3. Utworzoną adnotację dodaj do klas School i Student - wykorzystaj pola adnotacji.
4. Napisz kod, który wyświetli na ekranie wszystkie informacje o klasach School i Student oraz ich metodach.
5. * Rozszerz działanie adnotacji na pola klas School i Student oraz dopisz kod wyświetlający informacje o dodanych adnotacjach dla pól.
6. * Napisz nową adnotację **@DefaultParams**, która będzie przyjmować domyślne parametry wywołania metod. Następnie użyj adnotacji w klasach School i Student oraz napisz kod wywołujący metody z daną adnotacją i przekazujący im podane parametry.

Pamiętaj o wykorzystaniu repozytorium kodu Git! * Dla chętnych.

Aleksandria



Projekt Aleksandria

Aplikacja do zarządzania biblioteką posiadającą w swoich zasobach:

- książki
- czasopisma
- audiobooki
- filmy
- płyty winylowe

Dwa rodzaje użytkowników:

- administrator
- zwykły user - czytelnik

Możliwości (czytelnik):

- wypożyczenia zasobu (limity miesięczne!)
- wyszukania zasobu
- zwroty zasobu
- historia wypożyczeń
- rezerwacja jak zasób już wypożyczony przez kogoś innego

Możliwości (administrator):

- zarządzanie zasobami
- zarządzanie czytelnikami

Projekt Aleksandria - zadanie #1

Przed rozpoczęciem - utwórz i zaimportuj w Intellij nowe repozytorium kodu (GitHub) o nazwie:
javagda24_alexandria

Twoim pierwszym zadaniem będzie przygotowanie klas:

- User
  - każdy użytkownik ma imię i nazwisko
  - przy rejestracji użytkownik podaje również adres e-mail i nr telefonu
  - istnieje podział na typy: administrator / czytelnik
- Book
  - każdy książka ma swój tytuł, opis, autora, liczbę stron oraz kategorię
  - ile pozycji jest aktualnie dostępnych oraz ile jest ich w sumie
- Vinyl
  - każda płyta ma swój tytuł, wykonawcę oraz kategorię
  - ile pozycji jest aktualnie dostępnych oraz ile jest ich w sumie
- AlexandriaLibrary
  - klasa zarządzająca biblioteką
  - pozwala tworzyć nowych użytkowników
  - pozwala tworzyć nowe pozycje

Pamiętaj o wykorzystaniu repozytorium kodu Git! Dodaj link do repozytorium na Slack!

Projekt Aleksandria - zadanie #2

Przykładowe pomysły:

- stwórz pakiety i uporządkuj klasy. Możesz podzielić klasy względem rodzaju: osobno przedmioty do wypożyczenia, osobno osoby, osobno zarządzanie biblioteką.
- dodaj do klas odpowiednie modyfikatory dostępu na poziomie pól, konstruktorów i metod
- w klasach reprezentujących rzeczy do wypożyczenia (książki, płyty itp) dodaj metodę do wyszukiwania po tytule. Metoda zwróci **true** jeżeli szukana fraza znajdzie się w tytule przedmiotu.
- do klas przedmiotów dodaj pole **description**, które zawierać będzie długi opis przedmiotu. Dodaj metodę która zwróci krótką wersję (np.: 50 pierwszych znaków zaokrąglając do pełnego wyrazu)
- stwórz enum dla typu każdego z rodzajów przedmiotów, np.: dla płyt oznaczających rodzaj muzyki, dla książek rodzaj literatury
- wykorzystaj własne pomysły na podstawie tego co się dzisiaj nauczyłeś do rozbudowy projektu Aleksandria

Good Luck!

Pamiętaj o wykorzystaniu repozytorium kodu Git!

Projekt Aleksandria - zadanie #3

Pomysły:

- zadbaj o to, by książki i winyle dodawane były do tablicy o maksymalnym rozmiarze 100
  - * rozszerz działanie o możliwość dodawania elementów do tablicy o nieograniczonym rozmiarze
- zadbaj o utworzenie metod zarządzających kolekcją dostępnych pozycji w Twojej bibliotece: dodanie, usunięcie, wypożyczenie, edycja, wypisanie wszystkich pozycji, wszystkich dostępnych itd.
- utwórz klasę bazową dla klas: **Book** oraz **Vinyl** - **Item**
- wykorzystaj mechanizm dziedziczenia w przypadku klas **Book** i **Vinyl**
- dodaj kolejne klasy: **Movie, Newspaper** itp., które będą stanowiły zbiór Twojej biblioteki
- ...

Good Luck!

Pamiętaj o wykorzystaniu repozytorium kodu Git!

Projekt Aleksandria - zadanie #4

Pomysły:

- Zrób z klas bazowych (np. Item, Person) klasy abstrakcyjne
- Dodaj do projektu interfejsy - np. oddzielny interfejs z metodami do wypożyczenia przedmiotu (książki, płyty CD) oraz oddzielny interfejs do przedmiotów które można użyć tylko na miejscu (vinyle, czasopisma, gry...)
- Do przedmiotów, które można wypożyczyć dodaj daty (z godzinami): wypożyczenia i zwrotu
- Do kodu zarządzania biblioteką dodaj metodę która sprawdzi czy któryś z wypożyczonych przedmiotów nie ma przekroczonego okresu wypożyczenia (np. 1 miesiąca).
- Do określenia maksymalnego okresu przedmiotu zastosuj pole statyczne - może być różne dla różnych przedmiotów.

Good Luck!

Pamiętaj o wykorzystaniu repozytorium kodu Git!

Projekt Aleksandria - zadanie #5

Pomysły:

- zadbaj o odpowiednią obsługę sytuacji wyjątkowych w aplikacji - zarówno przy wprowadzaniu danych, jak i w odpowiedzi na niewłaściwe polecenia
- wykorzystując wiedzę o kolekcjach rozszerz swój kod o możliwość przechowywania danych w odpowiednich strukturach, które przetrzymywać będą dane w sposób uporządkowany
- ...

Good Luck!

Pamiętaj o wykorzystaniu repozytorium kodu Git!

Projekt Aleksandria - zadanie #6

Pomysły:

- wykorzystaj typy generyczne do refaktoryzacji swojego kodu, by był on bardziej uniwersalny i gotowy na kolejne typy danych
- dodaj do projektu możliwość zapisania danych (obiektów) do pliku, a także odczytu danych z pliku
- za pomocą klasy Scanner dodaj prosty interfejs do komunikacji z użytkownikiem (poprzez konsolę)
- spróbuj wykorzystać Stream API do przetwarzania danych w projekcie
- ….

Good Luck!

Pamiętaj o wykorzystaniu repozytorium kodu Git!

Projekt Aleksandria - zadanie #7

Pomysły:

- przeprojektuj interfejs użytkownika tekstowy na graficzny przy użyciu JavaFX
- pokuś się o wykonywanie niektórych zadań biblioteki w osobnych wątkach - pomyśl, które operacje mogą być dzięki temu bardziej wydajne i niezależne

Good Luck!

Pamiętaj o wykorzystaniu repozytorium kodu Git!