



Java

Wprowadzenie



Plan prezentacji

1. Wstęp
 - Czym jest programowanie
 - Na czym polega praca programisty
 - Wiodące języki programowania
 - W jakim kierunku możemy się rozwijać
2. Java – co i dlaczego powinniśmy o niej wiedzieć
 - Jakie są etapy tworzenia programu
 - Czym są pakiety
 - Biblioteki i ich zastosowanie
 - Struktura programu
3. Słownik pojęć
4. Programujemy!



Kilka słów wstępu

- Jesteśmy jednym zespołem – pomagamy sobie nawzajem.
- Dzielimy większy problem na mniejsze, tak długo, aż dotrzemy do tak małego elementu, z którym sobie poradzimy.
- Robimy notatki.
- Ilość nowych informacji jest ogromna, ich przyswojenie wymaga pracy zarówno w trakcie zajęć, jak i pomiędzy nimi.
- Jeśli bazujemy na czyimś rozwiązaniu – próbujemy (później) rozwiązać zadanie samodzielnie, jeszcze raz.
- Próbujemy nowych rzeczy, satysfakcja z używania tego, co już potrafimy – szybko minie.
- Nie popełnia błędów tylko ten, który nic nie robi.



Wstęp

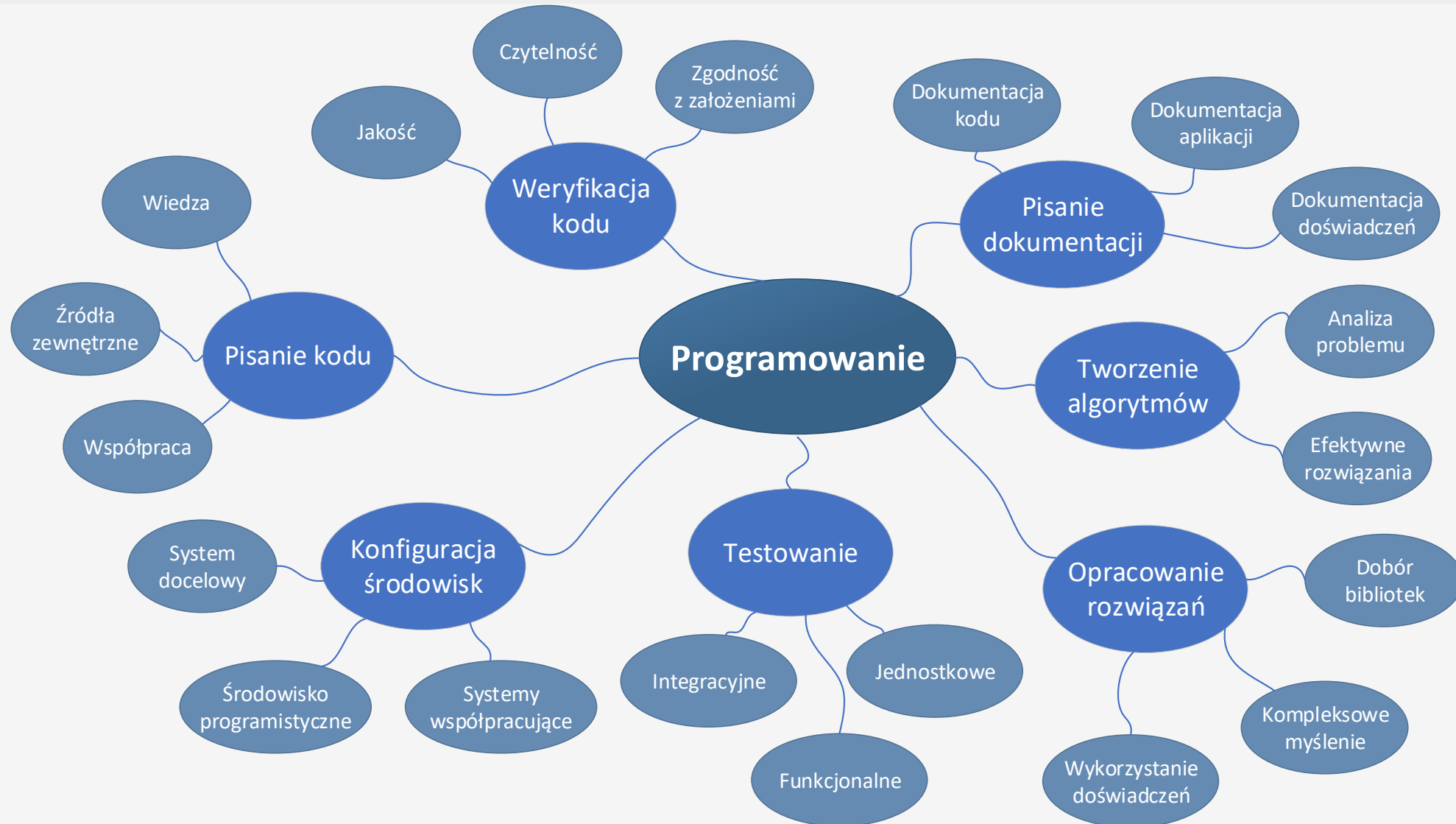
Co powinniśmy wiedzieć o programowaniu.

Autor: Marek Bobcow

Prawa do korzystania z materiałów posiada Software Development Academy



Czym jest programowanie?

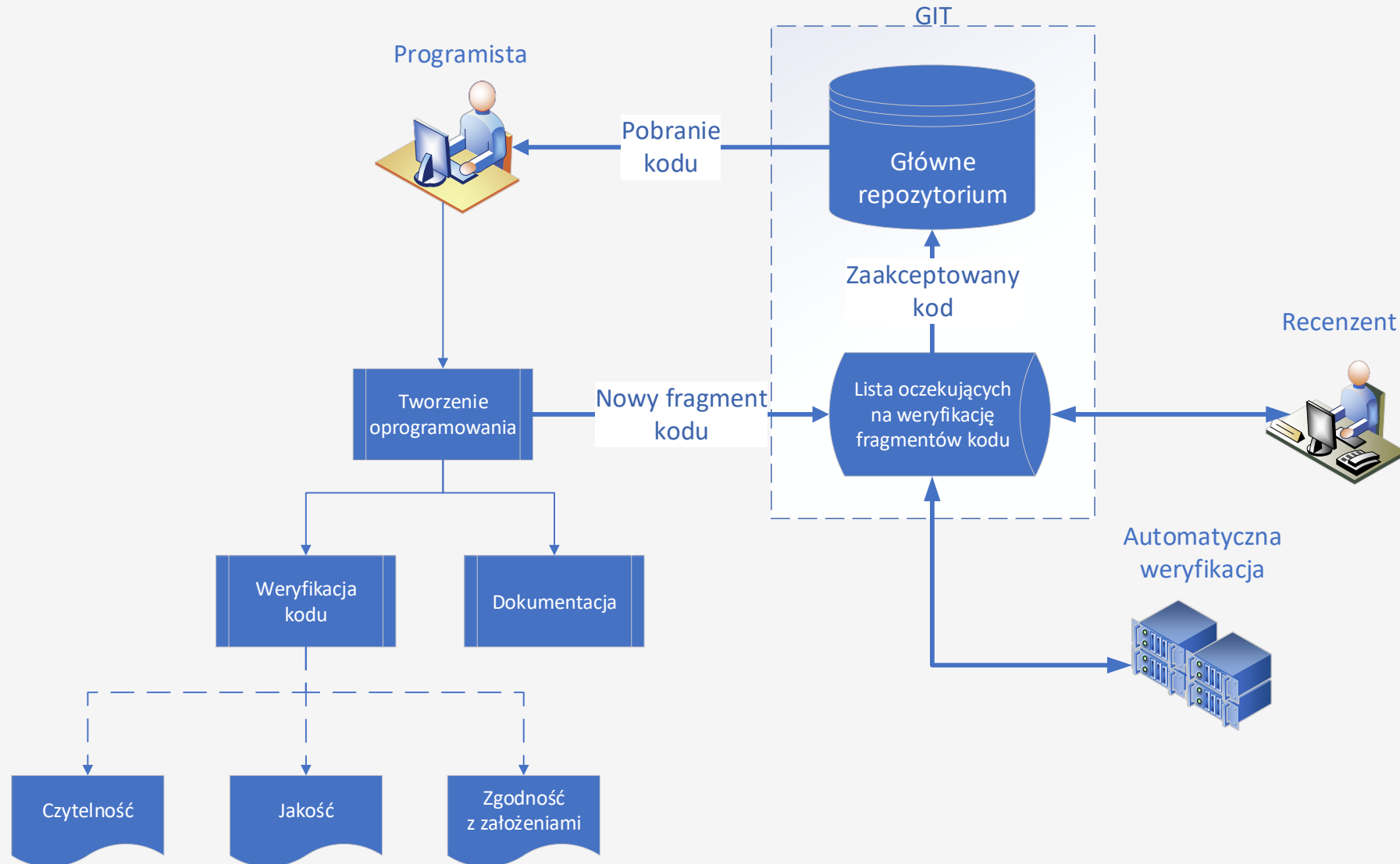


Autor: Marek Bobcow

Prawa do korzystania z materiałów posiada Software Development Academy



Praca programisty



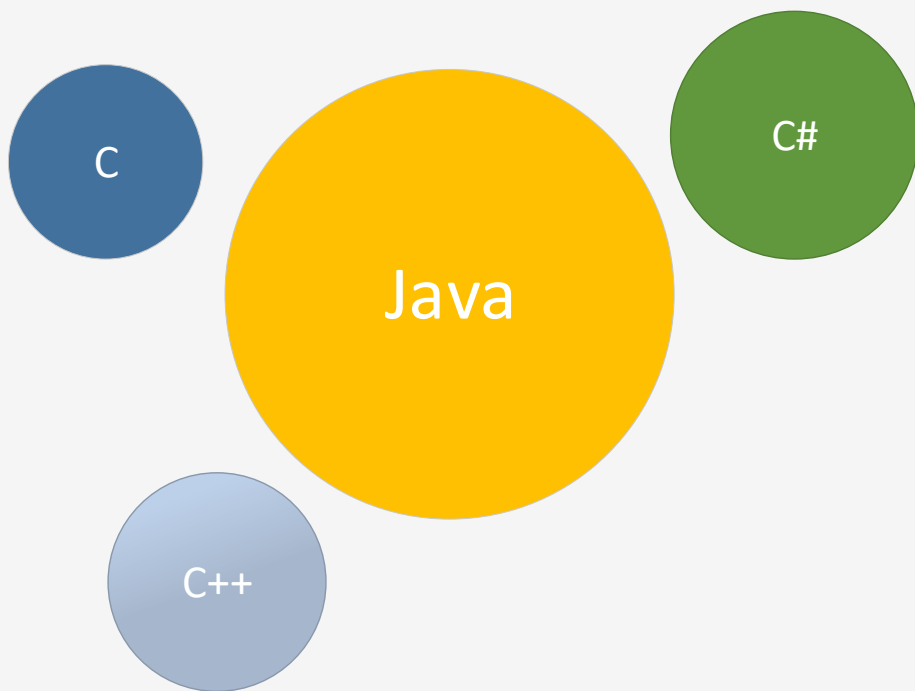
Autor: Marek Bobcow

Prawa do korzystania z materiałów posiada Software Development Academy

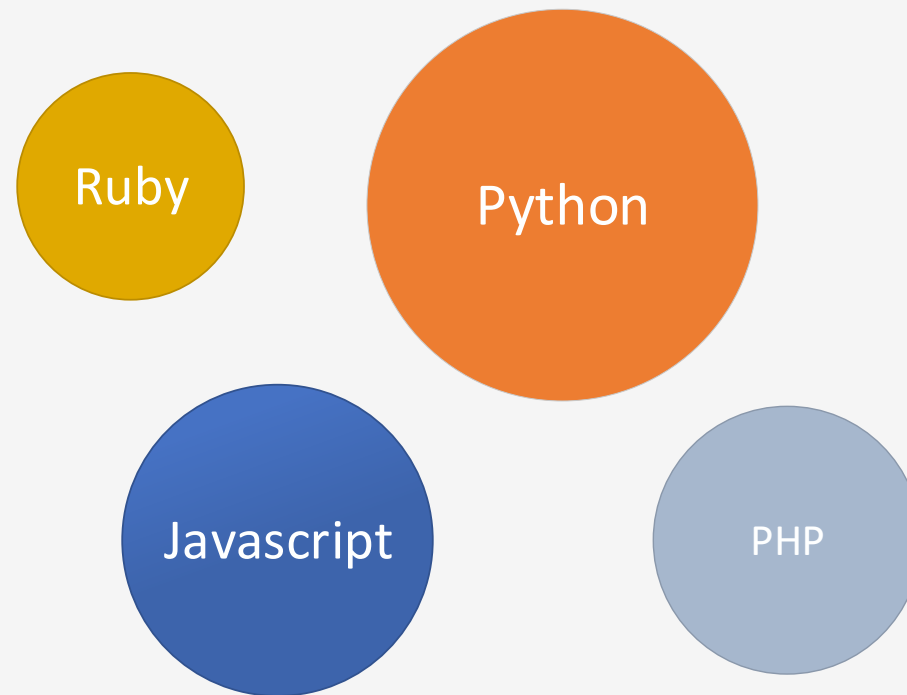


Wiodące języki programowania

Kompilowane



Skryptowe

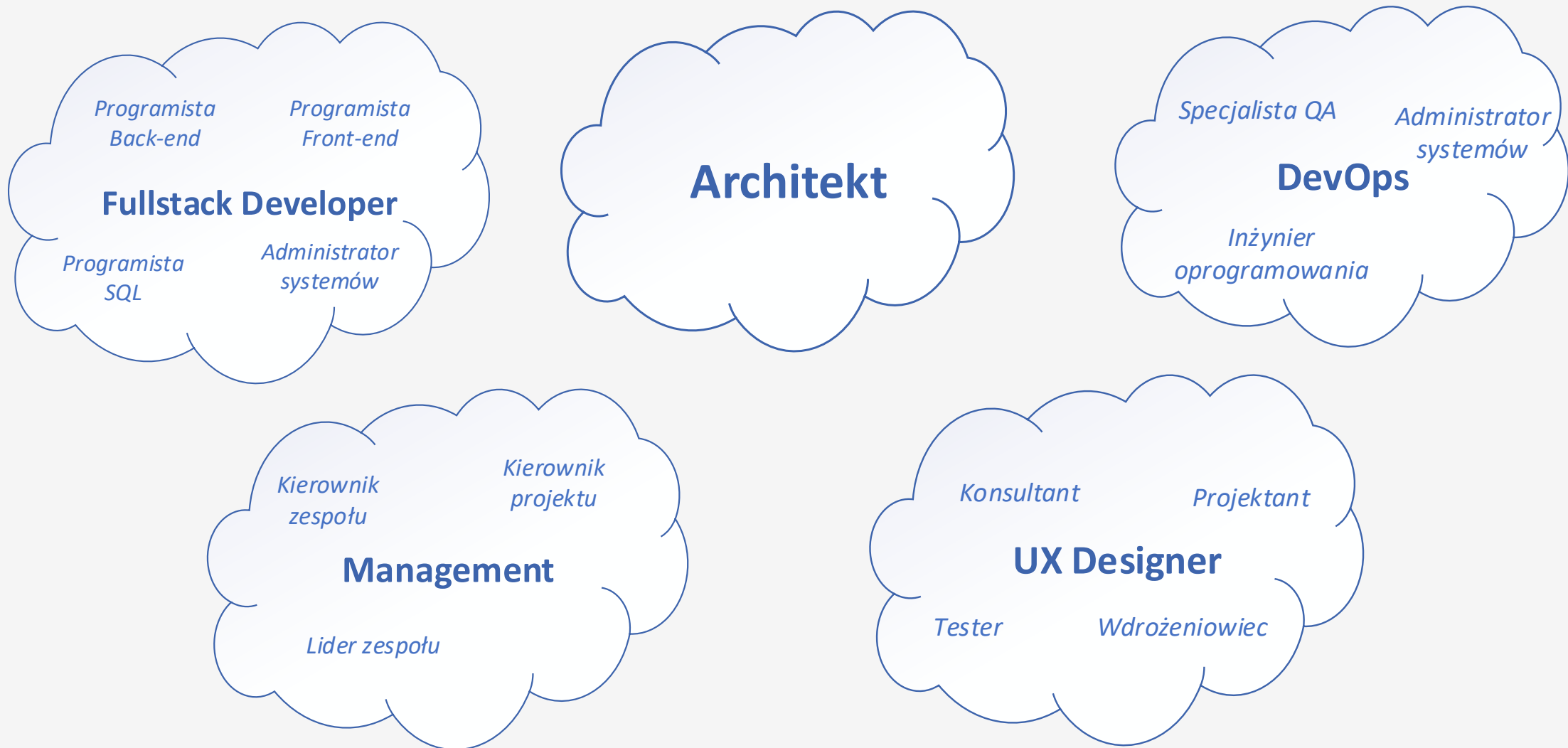


Autor: Marek Bobcow

Prawa do korzystania z materiałów posiada Software Development Academy



Kim możemy zostać?



Autor: Marek Bobcow

Prawa do korzystania z materiałów posiada Software Development Academy



Co pomaga w programowaniu?

- Cierpliwość
- Ciekawość i wnikliwość
- Planowanie przed wykonaniem (zawsze!)
- Spisywanie (pamiętanie) doświadczeń
- Analiza programów pisanych przez innych programistów
- Dokumentowanie (nikt tego nie lubi, ale każdy to docenia)*
- Współpraca
- Na etapie nauki ważne, aby działało „jakoś”, później zadbamy o „ć”
- Podczas nauki możemy korzystać z języka *polskiego*, podczas tworzenia rozwiązań komercyjnych **wyłącznie** z *angielskiego*
- *Stackoverflow*

*dobry kod jest sam w sobie komentarzem



Autor: Marek Bobcow

Prawa do korzystania z materiałów posiada Software Development Academy



Java

Przykład na to, jak powinien wyglądać dobry język programowania.

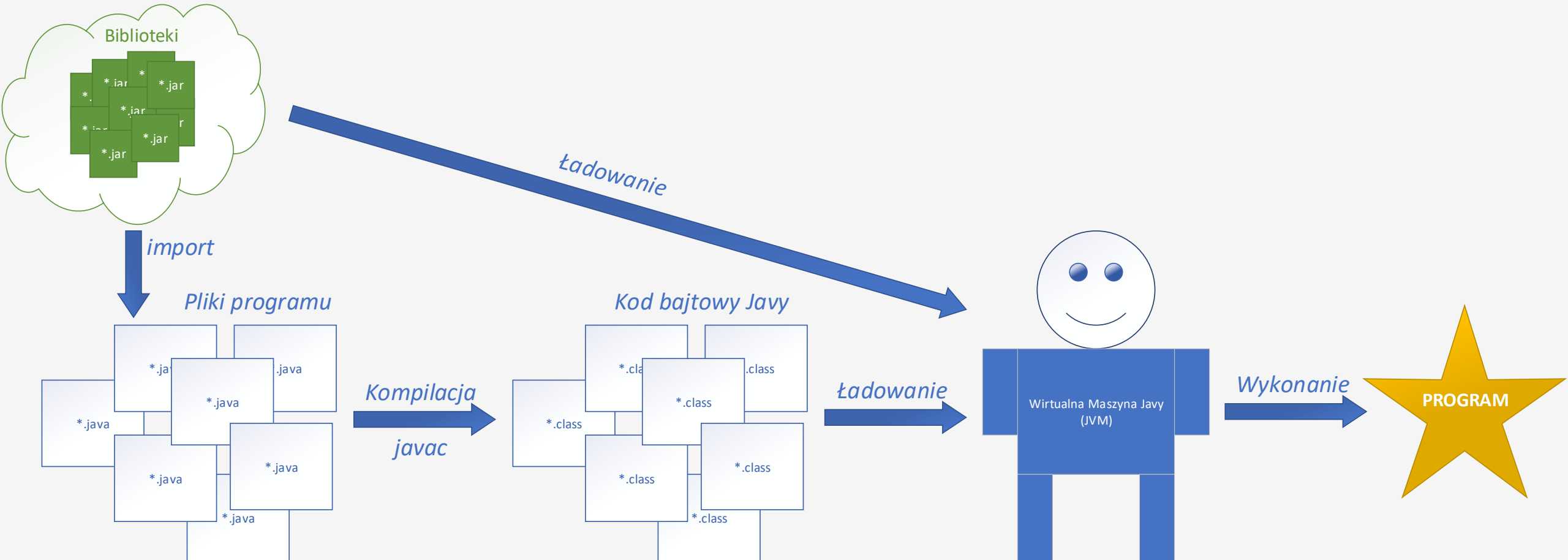


Minuta historii

- 1991 – zespół inżynierów z firmy Sun, pod przewodnictwem Jamesa Goslinga i Patricka Naughtona rozpoczynają projekt pod kryptonimem *Green*. Cel – utworzenie możliwie „lekkiego” (pod kątem koniecznych zasobów) języka programowania, który będzie niezależny od platformy sprzętowej. Oparcie projektu na języku C++.
- Tylko w początkowym etapie trwania projektu język nazywał się „*Oak*” – okazało się, że język o takiej nazwie już istnieje i podjęto decyzję (która okazała się strzałem w 10) o zmianie nazwy na „*Java*”.
- 23 maja 1995r. opublikowano w magazynie „Sun-World” kamień węgielny Javy – aplety. Od tego momentu popularność języka wyłącznie wzrastała.
- Liczba klas i interfejsów od 1996 do 2006 roku wzrosła z 211 do 3777.
- W ramach szkolenia używamy Javy w wersji 8 (Java jest zbiorczym określeniem oprogramowania i jego składników, w tym środowiska Java Runtime Environment (JRE) wirtualnej maszyny Java (JVM — Java Virtual Machine) oraz wtyczki (Plug-in)). Najnowsza wersja, wprowadzająca kilka dodatków, posiada numer 10.



Java – schemat blokowy





Czym są pakiety Javy?

Składniki języka Java, które są niezbędne, żeby móc z niego korzystać.

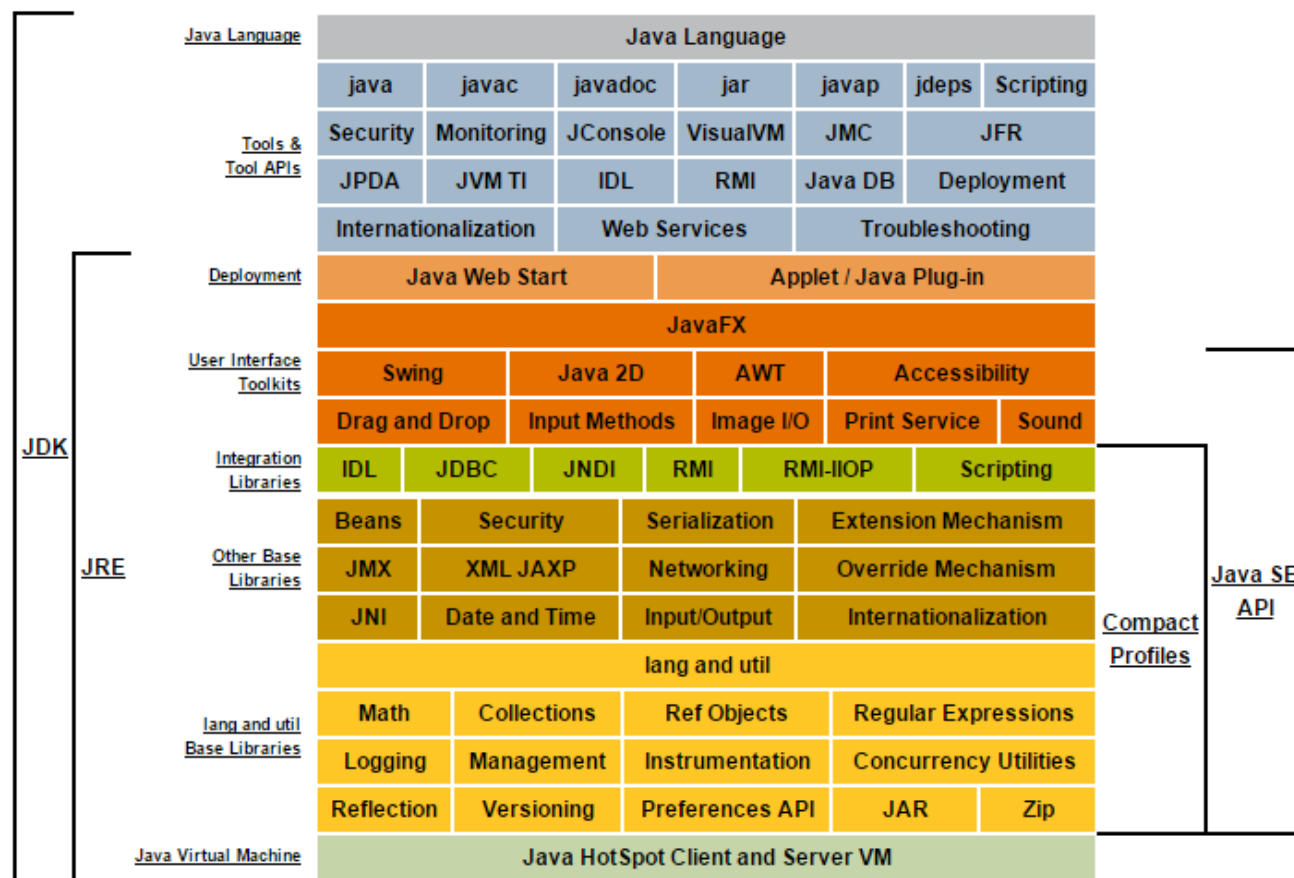
Jakiego pakietu potrzebujemy?

- Programiści (to my): JDK (Java SE Development Kit). Zawiera kompletne JRE oraz narzędzia do pisania oprogramowania, jego debuggowania oraz monitorowania.
- Administratorzy uruchamiający aplikacje serwerowe: Server JRE (Server Java Runtime Environment) zawierający narzędzia do monitorowania oraz wszelkie narzędzia niezbędne aplikacjom serwerowym. Nie zawiera integracji z przeglądarką, auto-update'u, ani instalatora.
- Użytkownicy końcowi: JRE (Java Runtime Environment). Pokrywa większość potrzeb użytkowników.

Java – pakiety c.d.

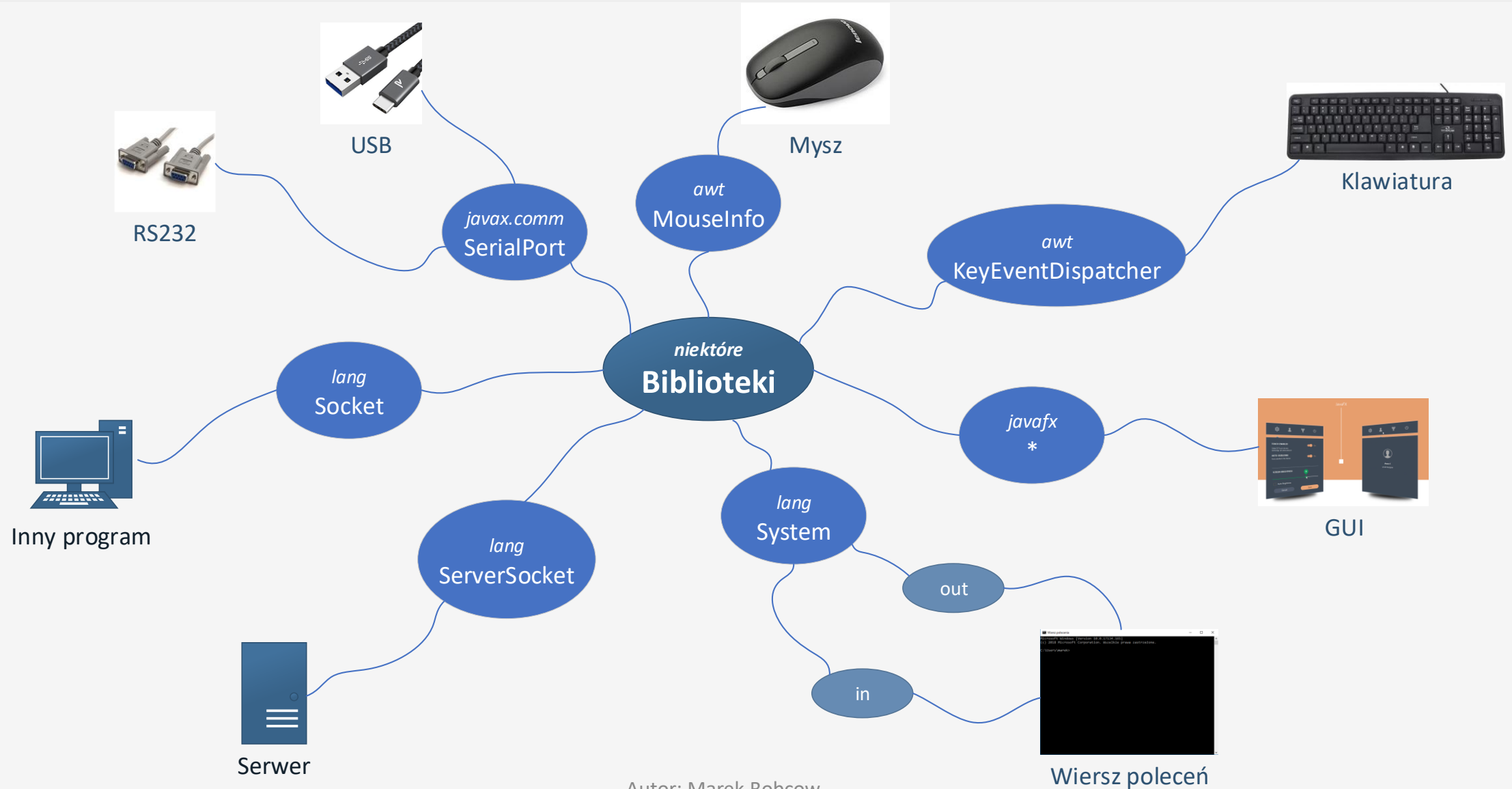


Description of Java Conceptual Diagram



Autor: Marek Bobcow

Prawa do korzystania z materiałów posiada Software Development Academy



Autor: Marek Bobcow

Prawa do korzystania z materiałów posiada Software Development Academy



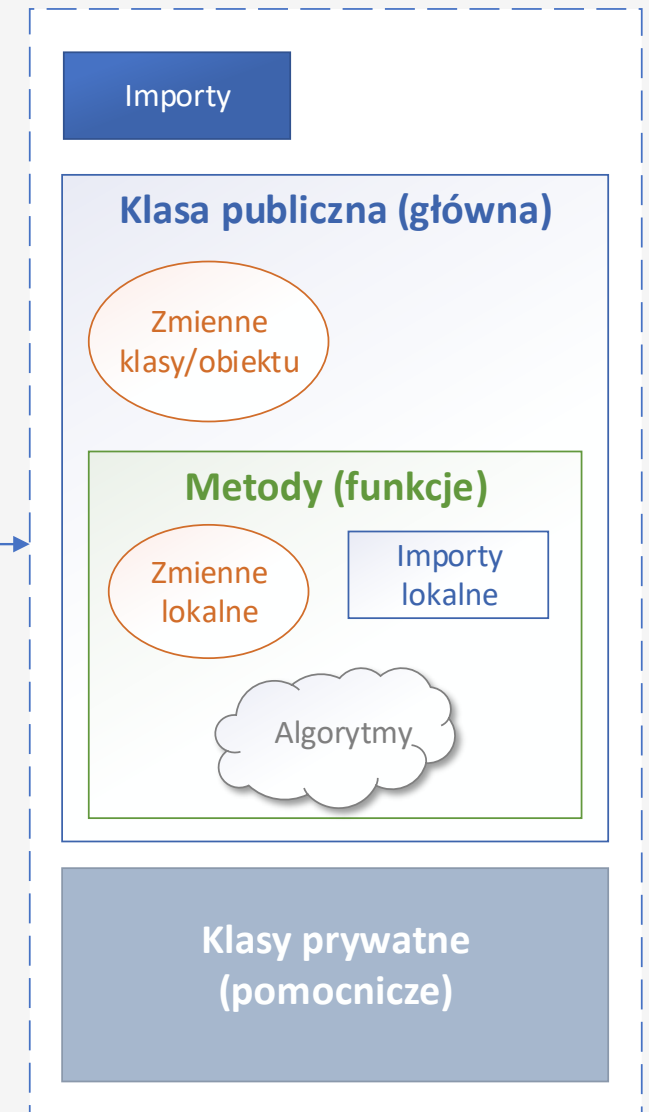
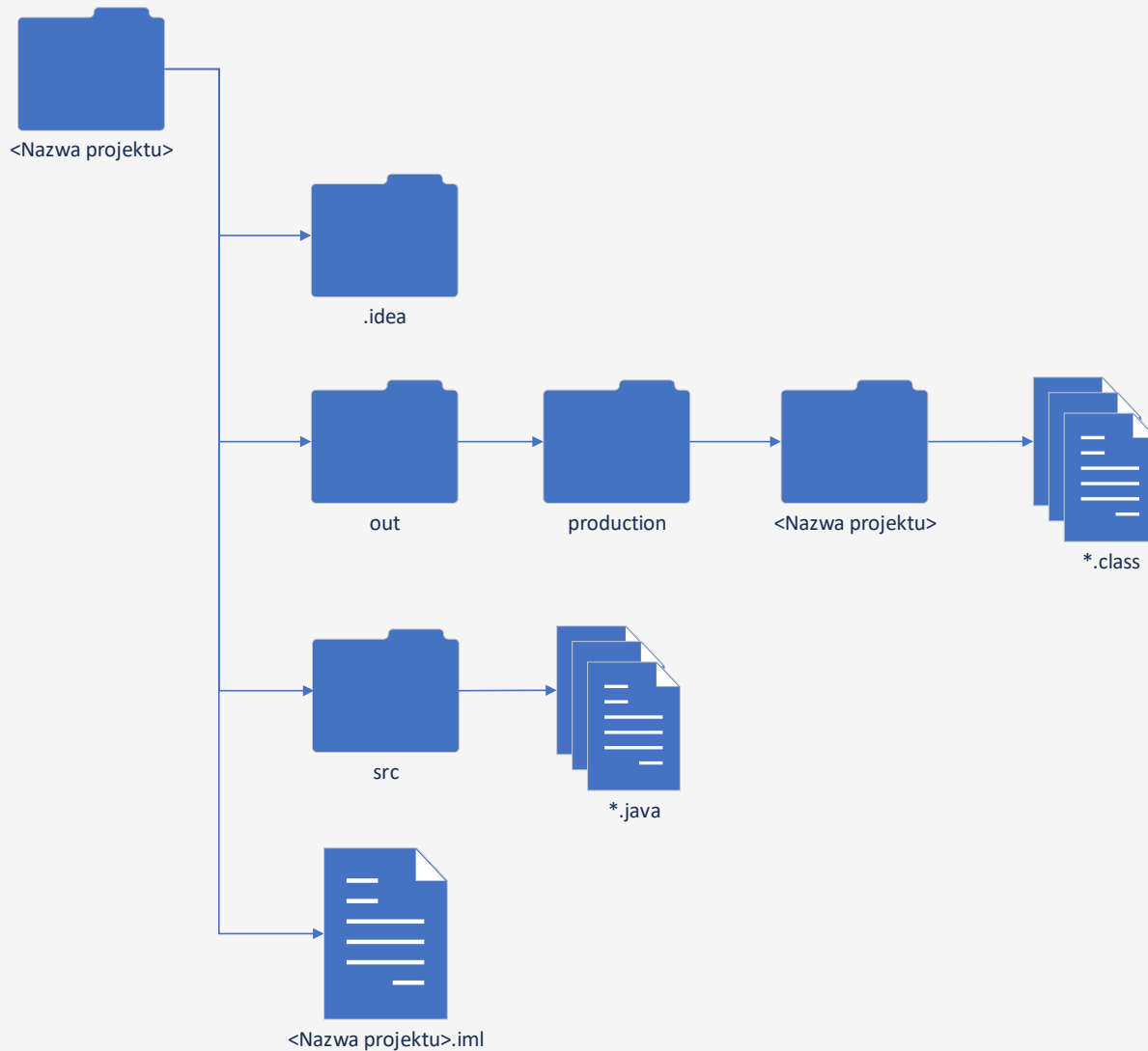
Java – główne założenia

Java jest językiem obiektowym i jako taki musi spełniać 4 główne założenia (do których będziemy jeszcze wielokrotnie wracali):

- **Abstrakcja** – prosta reprezentacja złożonych zagadnień.
 - pilot do telewizora
- **Enkapsulacja** – nadawanie dostępu tylko do tych elementów, które są niezbędne, reszta powinna być „prywatna”.
 - dostęp do naszych pieniędzy
- **Dziedziczenie** – tworzymy rozwiązania bazując na istniejących.
 - koło
- **Polimorfizm** – znaczenie zależne od kontekstu.
 - zamek, zamek i zamek



Java – struktura (prostego) projektu



Autor: Marek Bobcow

Prawa do korzystania z materiałów posiada Software Development Academy



Przygotowujemy się do pracy

- Dostęp do sieci
 - SSID i hasło
- Java
 - <http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html>
- IDE – IntelliJ Idea Community
 - <https://www.jetbrains.com/idea/download>
- Git
 - <https://git-scm.com/downloads>
- Folder do projektów
 - Java – wprowadzenie -> oprogramowanie



Słownik pojęć

Autor: Marek Bobcow

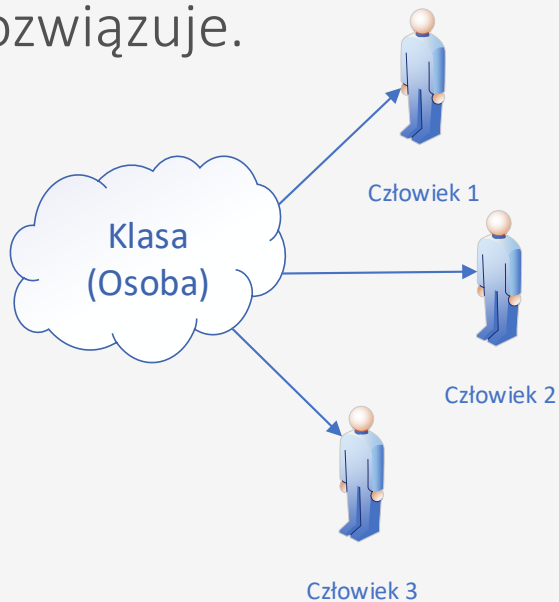
Prawa do korzystania z materiałów posiada Software Development Academy



Klasa

Klasa – szablon, z którego tworzy się obiekty

Konstruując obiekt, tworzymy **egzemplarz** Klasy. Wszystko to, co piszemy w Javie, znajduje się w jakiejś klasie. Nazwy klas tworzymy wykorzystując **rzeczowniki**, które najlepiej oddają problem, jaki dana klasa rozwiązuje.



```
public class Osoba {  
  
    private String kolorOczu;  
    private String kolorWlosow;  
    private String plec;  
    private int wzrostCm;  
  
    Osoba(String kolorOczu, String kolorWlosow, String plec, int wzrostCm) {  
        this.kolorOczu = kolorOczu;  
        this.kolorWlosow = kolorWlosow;  
        this.plec = plec;  
        this.wzrostCm = wzrostCm;  
    }  
  
    public void ustawKolorOczu(String kolor) {  
        this.kolorOczu = kolor;  
    }  
  
    public String podajKolorOczu() {  
        return this.kolorOczu;  
    }  
  
    ...  
}
```



Obiekt

Obiekt – egzemplarz klasy

Wszystkie obiekty charakteryzują się:

- **Zachowaniem** – co można z nimi zrobić, jakie metody można wywołać na ich rzecz?
- **Stanem** – jak obiekt reaguje na wywołane na jego rzecz metody?
- **Tożsamością** – jak odróżnić jeden obiekt od drugiego (tej samej klasy)?

Wszystkie obiekty tej samej klasy charakteryzują się **jednakowym zachowaniem**.

```
class Klasy {  
  
    public static void main(String[] args) {  
  
        Osoba tomasz = new Osoba("brązowe", "czarne", "mężczyzna", 180);  
        Osoba krystyna = new Osoba("zielone", "czerwone", "kobieta", 160);  
        System.out.println("Krystyna ma " + krystyna.podajWzrost() + " cm wzrostu");  
        System.out.println("Tomasz ma " + tomasz.podajKolorOczu() + " oczy");  
    }  
}
```

```
>> Krystyna ma 160 cm wzrostu  
>> Tomasz ma brązowe oczy
```



Metoda

Metoda – procedura operująca na danych

Metoda, na podstawie dostarczonych danych oraz danych zewnętrznych, do których posiada dostęp, wykonuje ściśle określone zadanie, np.:

- zwraca informację o stanie obiektu,
- przeprowadza obliczenia na dostarczonych danych,
- zapisuje dostarczone dane...

```
public void ustawKolorOczu(String kolor) {  
    this.kolorOczu = kolor;  
}  
  
public String podajKolorOczu() {  
    return this.kolorOczu;  
}  
  
public int podajWzrost() {  
    return this.wzrostCm;  
}  
  
public int obliczSume(int x, int y) {  
    return x + y;  
}
```



Zmienna

Zmienna – przechowuje dane określonego typu

Java jest językiem **silnie typowanym**, tzn., że każda zmienna musi mieć ściśle określony typ. Zadanie określenia typu danej zmiennej należy do **programisty**.

Najbardziej popularne typy danych to:

- **String** – ciąg dowolnych znaków (w tym liczb), np. imię, adres
- **int** – liczba całkowita, np. wzrost w cm,
- **double** – liczba rzeczywista, np. stan konta,
- **boolean** – typ logiczny, przyjmujący tylko dwie wartości – **true** lub **false**

```
public class Zmienne1 {  
  
    int staloprzecinkowa = 10;  
    String ciagZnakow = "dowolny ciag 12345..";  
    boolean warunek = true;  
    double stanPortfela = 8.79;  
  
}
```

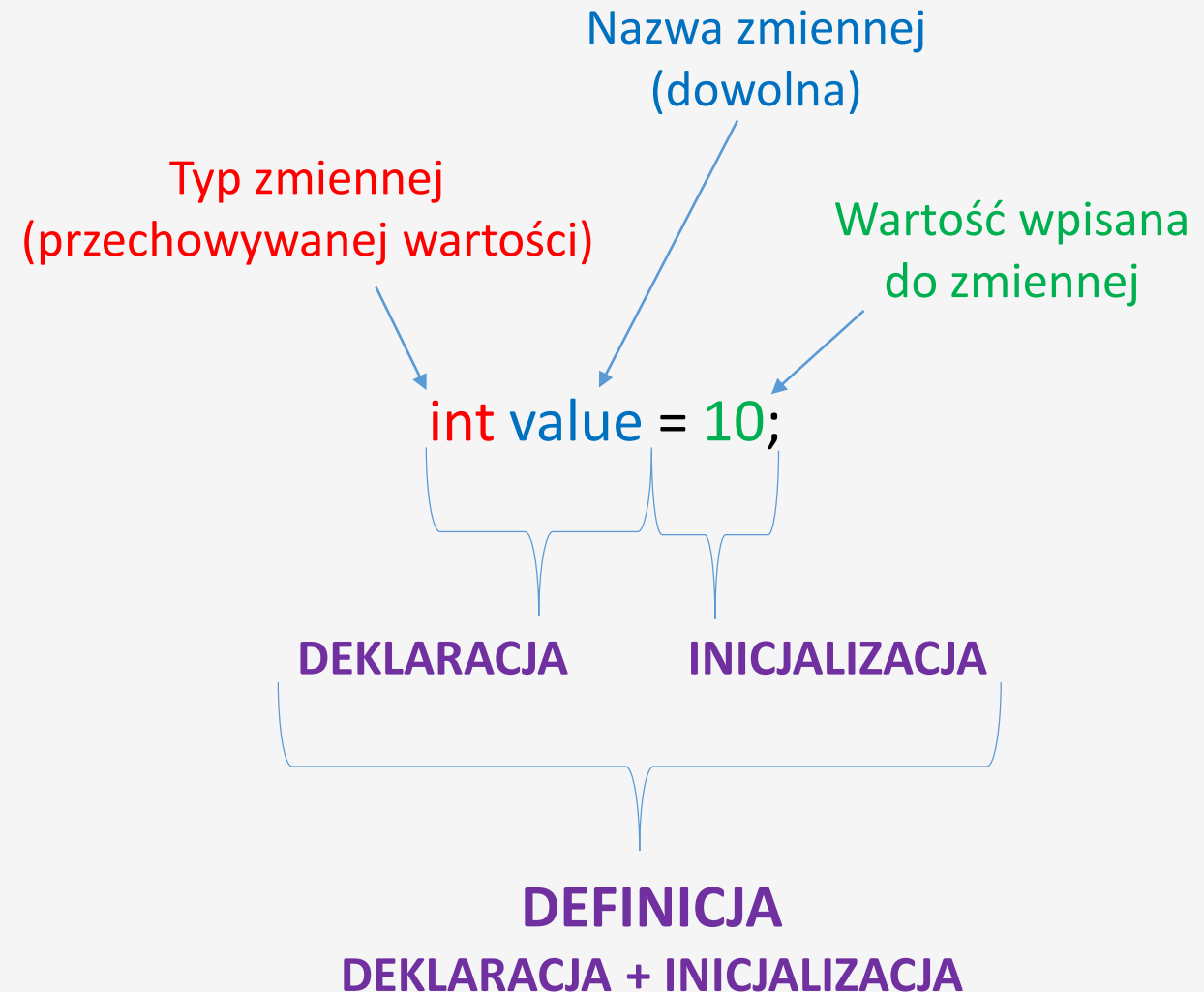


Tworzenie zmiennej

W celu utworzenia nowej zmiennej musimy:

- określić typ danych, jaki będzie przechowywała,
- wybrać nazwę,
- przypisać do niej określoną wartość (jeśli tego nie zrobimy, to zmienna otrzyma wartość domyślną).

Deklaracja i inicjalizacja zmiennej **nie musi** odbywać się w jednej linii (instrukcji).





- Dodawanie

`x = x + 10`

`x += 10`

- Odejmowanie

`x = x - 10`

`x -= 10`

- Mnożenie

`x = x * 10`

`x *= 10`

- Dzielenie

`x = x / 10`

`x /= 10`

- Inkrementacja

post -> `x++`

pre -> `++x`

- Dekrementacja

post -> `x--`

pre -> `--x`

- Jest równe

`x == y`

- Jest różne

`x != y`

- Oraz

`x && y`

- Lub

`x || y`

- Większy

`x > y`

- Mniejszy

`x < y`

- Większy lub równy

`x >= y`

- Mniejszy lub równy

`x <= y`



```
/* Komentarz wieloliniowy  
* zawartość tego komentarza będzie zignorowana  
* - Opis zadania realizowanego przez dany plik/klasę/metodę  
* - Przykład danych trafiających do danej klasy...  
* */  
  
// Komentarz jednoliniowy - zawartość tego komentarza również zostanie zignorowana
```



Zaczynamy

Autor: Marek Bobcow

Prawa do korzystania z materiałów posiada Software Development Academy



1. Witaj Świecie!
2. Wypisywanie danych na konsolę (ekran)
3. Czytanie danych z konsoli
4. Instrukcje warunkowe: if-else, switch-case
5. Pętle: for, while, do-while, for-each
6. Tablice jedno i wielowymiarowe

Nasz pierwszy program



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



Wypisywanie danych na ekran

```
public class Wyjscie1 {  
  
    public static void main(String[] args) {  
  
        System.out.println("Standardowy tekst");  
        System.out.printf("Sformatowany tekst int : %d, double : %.2f, string : %s\n", 10, 7.52423, "przyklad");  
  
        String tekstA = "Przykładowy tekst.";  
        String tekstB = "Inny przykładowy tekst.";  
        System.out.printf("%s %s\n", tekstA, tekstB);  
        System.out.println("Lub inaczej... " + tekstA + " " + tekstB);  
  
        String tekst = String.format("Jak wcześniej int : %d, double : %f, string : %s", 25, 7.232, "test");  
        System.out.println(tekst);  
  
        System.out.println("Wypisane w nowej linii");  
        System.out.print("I to również!");  
        System.out.print("Ale to już nie...");  
  
    }  
  
}
```



1. Wprowadź swoje imię do zmiennej typu String. Wypisz na konsolę „Witaj, <imię>” wykorzystując funkcję println.
2. Wypisz dwa dowolne łańcuchy znaków jeden pod drugim z wykorzystaniem funkcji print.
3. Wprowadź dowolną wartość z kilkoma cyframi po przecinku do zmiennej typu double. Wyświetl podaną wartość z dokładnością do dwóch miejsc po przecinku.
4. *Wyświetl trzy dowolne ciągi znaków w jednej linii tak, aby były wyrównane do prawej krawędzi 15 znakowych bloków. Np. „ test t wprowadzenie”.
5. **Wyświetl wartości 192, 168, 1, 10 heksadecymalnie w formacie XX:XX:XX:XX, dla podanych wartości powinniśmy otrzymać: „C0:A8:01:0A”



```
import java.util.Scanner;

public class Wejscie1 {

    public static void main(String[] args) {
        Scanner wejscie = new Scanner(System.in);
        String napisanaLinia = wejscie.nextLine();
    }
}
```




1. Wprowadź swoje imię z poziomu konsoli i zapisz je do zmiennej typu String. Wypisz na konsolę „Witaj, <imię>”.
2. Wprowadź z poziomu konsoli dwie wartości, dodaj je do siebie i wypisz ich sumę.



Instrukcje warunkowe – if - else

```
Jeśli (warunek lub warunki) {  
    Instrukcje do wykonania  
} W przeciwnym wypadku jeśli (warunek lub warunki) {  
    Instrukcje do wykonania  
} W pozostałych wypadkach {  
    Instrukcje do wykonania  
}
```

- Warunki sprawdzamy od góry do dołu.
- Jeśli którykolwiek z warunków będzie spełniony, to wykonujemy instrukcje dla tego warunku. **Pozostałych warunków nie sprawdzamy.**
- Warunek może być pojedynczy, np. „x == 10” lub możemy sprawdzić kilka warunków jednocześnie, np. „x == 10 && y >= 4”.
- Liczba bloków „W przeciwnym wypadku jeśli ...” może być dowolna.



Instrukcje warunkowe – if - else

```
public class Warunki1 {  
  
    public static void main(String[] args) {  
  
        int naszaLiczba = 140;  
  
        if(naszaLiczba < 10){  
            System.out.println("Nasza liczba jest mniejsza od 10");  
        } else if (naszaLiczba == 10) {  
            System.out.println("Nasza liczba jest równa 10");  
        } else {  
            System.out.println("Nasza liczba jest większa od 10");  
        }  
  
    }  
  
}
```



1. Zmodyfikuj przykładowy program tak, aby pobierana liczba pochodziła z konsoli. Przetestuj program dla każdego przypadku (liczba mniejsza, równa lub większa od 10). Podpowiedź: skorzystaj z metody Scannera „nextInt”.
2. Pobierz z konsoli wartość od 0 do 9. Na podstawie otrzymanej wartości wyświetl dowolny znak. Na przykład dla numeru 0 wyświetl „*”, dla 1 wyświetl „\$” (lub dowolny inny).
3. *Jak wyżej, ale zamiast wartości operuj na łańcuchach znaków (Stringach). Na przykład dla słowa „gwiazdka” wyświetl „*”.



Instrukcje warunkowe switch – case

```
Porównaj zmienną (zmienna) {  
    wartość zmiennej to ...:  
        Instrukcje do wykonania  
    przerwij  
    wartość zmiennej to ...:  
        Instrukcje do wykonania  
    przerwij  
    domyślnie wykonaj:  
        Instrukcje do wykonania  
    przerwij  
}
```

- Porównujemy zmienną do określonych z góry wartości (w powyższym przykładzie nasze wartości oznaczone są przez „...”).
- Musimy uwzględnić wszystkie możliwe wartości, jakie przyjmie nasza zmienna.
- Jeśli nasza zmienna nie jest zgodna z żadną ze zdefiniowanych wartości, to wykonają się instrukcje z bloku „domyślnego”.
- Każdy blok instrukcji musi zostać na końcu **przerwany**. W innym wypadku zostaną wykonane instrukcje dla **każdej kolejnej wartości zmiennej**.



Instrukcje warunkowe switch – case

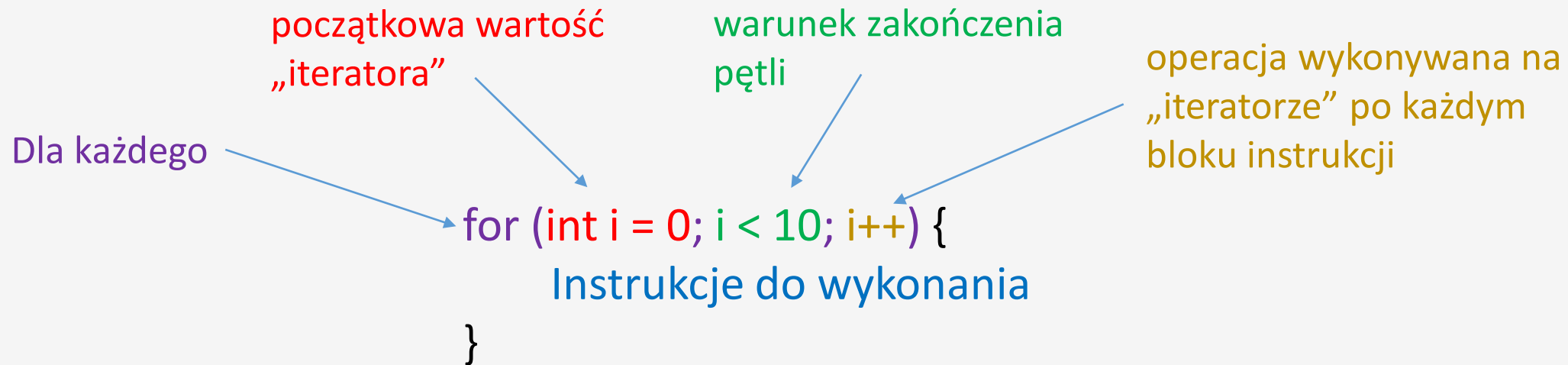
```
public class Warunki2 {  
  
    public static void main(String[] args) {  
  
        int naszaLiczba = 140;  
  
        switch(naszaLiczba) {  
            case 10:  
                System.out.println("Nasza liczba jest rowna 10");  
                break;  
            case 100:  
                System.out.println("Nasza liczba jest rowna 100");  
                break;  
            default:  
                System.out.println("Nasza liczba jest rozna od 10 i 100");  
                break;  
        }  
    }  
}
```



1. Zmodyfikuj przykładowy program tak, aby pobierana liczba pochodziła z konsoli. Przetestuj program dla każdego przypadku (liczba mniejsza, równa lub większa od 10). Podpowiedź: skorzystaj z metody Scannera „nextInt”.
2. Pobierz z konsoli wartość od 0 do 9. Na podstawie otrzymanej wartości wyświetl dowolny znak. Na przykład dla numeru 0 wyświetl „*”, dla 1 wyświetl „\$” (lub dowolny inny).
3. Jak wyżej, ale zamiast wartości operuj na łańcuchach znaków (Stringach). Na przykład dla słowa „gwiazdka” wyświetl „*”.
4. *Jak wyżej, ale po wyświetleniu danego znaku wyświetl wszystkie kolejne możliwości (jeśli posiadamy case „gwiazdka”, a kolejny to „dolar” to po wpisaniu gwiazdki powinniśmy otrzymać zarówno gwiazdkę, jak i dolar).



1. Napisz prosty kalkulator, który:
 - a) Przyjmuje dwie wartości x i y , zwraca ich sumę oraz różnicę. Wykorzystaj funkcję `printf` albo `String.format` tak, aby znak „=„ i „+” oraz „-” były w tym samym miejscu jeden pod drugim. Nie używaj metody `nextInt()`.
 - b) j/w + podawanie informacji, czy chcemy, aby dokonać dodawania, czy odejmowania poprzez napisanie „suma” lub „różnica”. Wykorzystaj instrukcję warunkową `if-else`
 - c) *j/w + użytkownik może zdecydować jaką operację chce wykonać uwzględniając mnożenie i dzielenie. Wykorzystaj instrukcję warunkową `switch-case`
 - d) **j/w + skorzystaj z „ternary operator” zamiast standardowej instrukcji `if-else`
 - e) **obsłuż sytuację, w której użytkownik poda wartość 0 podczas dzielenia.
2. Napisz program, który pozwoli określić, czy wprowadzona przez użytkownika liczba:
 - a) Jest parzysta
 - b) Jest podzielna przez drugą podaną liczbę



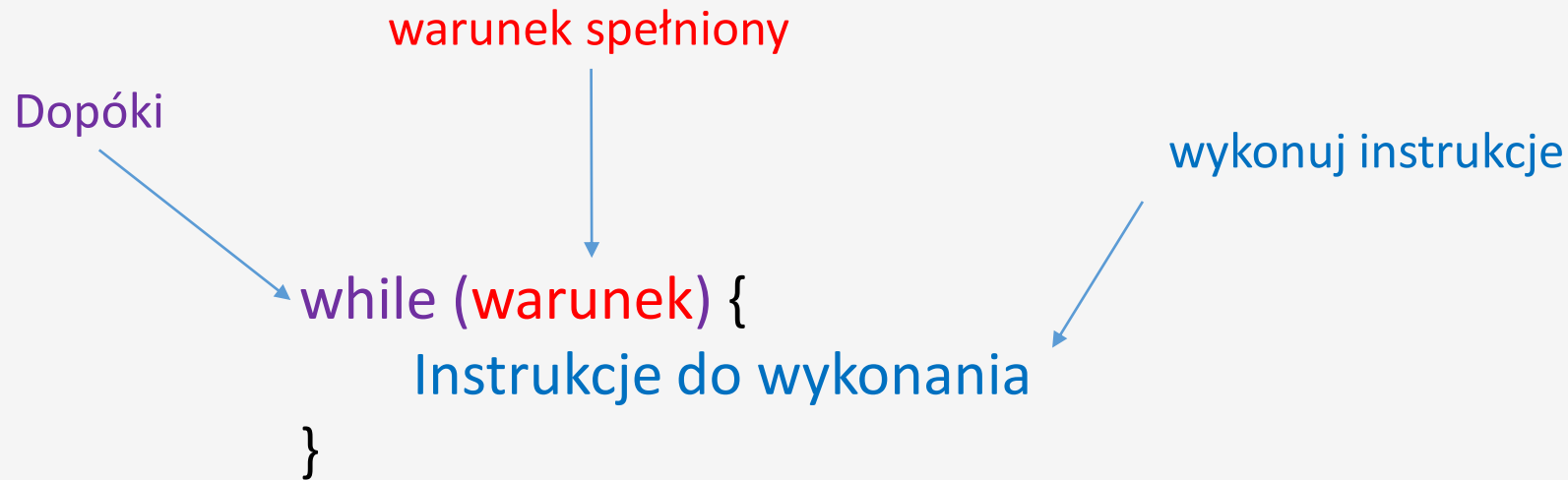
- Pętla **for** pozwala na wykonanie „instrukcji do wykonania” określoną liczbę razy.
- Zarówno początkowa wartość iteratora, jak i warunek zakończenia pętli może być dowolny.
- Jeśli zmienną iterującą stworzymy na potrzeby danej pętli („int i = ...”), to po wyjściu z pętli zmienna iterująca jest usuwana z pamięci.
- **Zastosowanie:** operacje na zbiorach danych (tablicach, obiektach), wypisywanie wartości



```
public class For1 {  
  
    public static void main(String args[]) {  
  
        for(int i = 0; i < 5; i++){  
  
            System.out.println("Aktualna wartość i : " + i);  
  
        }  
        System.out.println("");  
        for(int i = 0; i <= 5; i++){  
  
            System.out.println("Aktualna wartość i : " + i);  
  
        }  
        System.out.println("");  
        for(int i = 0; i <= 5; ++i){  
  
            System.out.println("Aktualna wartość i : " + i);  
  
        }  
  
    }  
  
}
```



1. Wewnątrz pętli for (np. 10 razy) pobieraj produkt do kupienia i wyświetlaj go na ekran w postaci : „Dodałem do koszyka <nazwa>, to nasz <numer iteracji> produkt!”
2. Stwórz pętle for, która wykona 5 iteracji. Wewnątrz pętli pobieraj z konsoli dowolną wartość typu int. Po wyjściu z pętli zwróć sumę tych wartości.
3. *Stwórz pętle wewnątrz pętli (pamiętaj o różnych nazwach zmiennych iterujących!). Wyświetl wartości iteratorów w postaci: „<iterator nr 1> : <iterator nr 2>”.
4. **Wyświetl kwadrat składający się z samych gwiazdek „*”, których liczba (długość boku kwadratu) będzie równa podanej z konsoli wartości.
5. **Jak wyżej, ale znak, z którego będzie składał się kwadrat, również pobierz z konsoli.



- Pętla **while** pozwala na wykonywanie określonych instrukcji tak długo, jak długo warunek pętli jest spełniony.
- W przeciwieństwie do pętli `for` nie podajemy danych dotyczących iteratora (wartości początkowej, instrukcji wykonywanej po każdym bloku)
- **Zastosowanie:** pobieranie instrukcji od użytkownika aplikacji, do momentu otrzymania określonego rozkazu; czytanie zawartości pliku; usypianie programu do momentu spełnienia określonego warunku

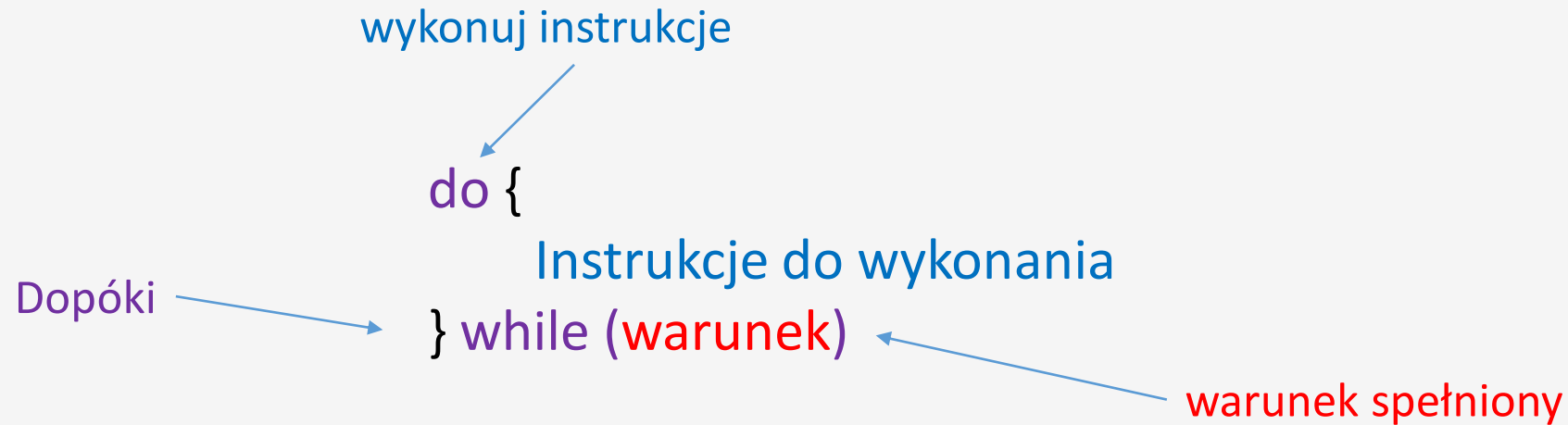
Pętle – while



```
public class While1 {  
  
    public static void main(String args[]){  
        int i=0;  
  
        while(i < 5) {  
            System.out.printf("Aktualna wartość i : %d\n", i);  
            i++;  
        }  
        System.out.println("Zerujemy wartość licznika");  
        i = 0;  
  
        while(i++ < 5) {  
            System.out.printf("Aktualna wartość i : %d\n", i);  
        }  
        System.out.println("Zerujemy wartość licznika");  
        i = 0;  
  
        while(++i < 5) {  
            System.out.printf("Aktualna wartość i : %d\n", i);  
        }  
    }  
}
```



Pętle – do while



- Pętla **do while** pozwala na wykonywanie określonych instrukcji tak długo, jak długo warunek pętli jest spełniony. W przeciwieństwie do pętli `while` wykona się **zawsze chociaż jeden raz, nawet, jeśli warunek nie będzie spełniony**.
- **Zastosowanie:** próba połączenia z inną aplikacją tak długo, aż nie otrzymamy informacji o prawidłowym połączeniu; zastosowania podobne do pętli `while`

Pętle – do while



```
public class DoWhile {  
  
    public static void main(String args[]){  
  
        int i = 16;  
  
        do {  
            System.out.println("Ta informacja wyświetli się zawsze chociaż jeden raz! Wartość i : " + i);  
        }  
        while(i < 15);  
    }  
  
}
```



1. Wewnątrz pętli while (np. 10 razy) pobieraj produkt do kupienia i wyświetlaj go na ekran w postaci : „Dodałem do koszyka <nazwa>, to nasz <numer iteracji> produkt!”
2. Stwórz pętle while, która wykona 5 iteracji. Wewnątrz pętli pobieraj z konsoli dowolną wartość typu int. Po wyjściu z pętli zwróć sumę tych wartości.
3. Jak wyżej, ale zwróć sumę tylko tych wartości, które były większe od 10;
4. *Pobieraj i wyświetlaj dowolny ciąg znaków od użytkownika tak długo, aż nie napisze „koniec”.
5. *Jak wyżej, z i bez wyświetlania słowa „koniec”.

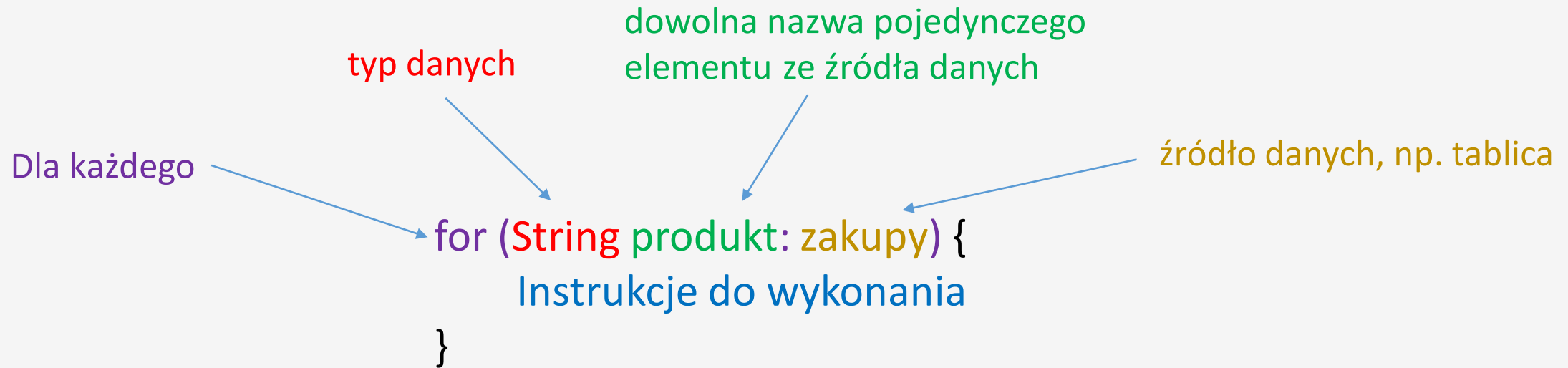


Tablice jednowymiarowe

```
public class Tablice1 {  
  
    public static void main(String args[]) {  
  
        int[] tab = new int[10];  
        System.out.println("Tablica tab ma długość : " + tab.length);  
  
        for(int i = 0; i < 10; i++){  
            System.out.println("Wartość w tablicy przed wpisaniem tab[" + i + "] : " + tab[i]);  
        }  
  
        for(int i = 0; i < 10; i++){  
            tab[i] = i + 50;  
        }  
  
        for(int i = 0; i < 10; i++){  
            System.out.println("Wartość w tablicy po wpisaniu tab[" + i + "] : " + tab[i]);  
        }  
    }  
}
```



Pętle – for each



- Pętla **for each** pozwala na wykonanie jednakowych instrukcji dla każdego elementu z danego zbioru danych. Jest to uproszczona forma pętli `for(int i=0; i <)` umożliwiająca ominięcie zmiennej iterującej.
- Źródło danych musi przechowywać elementy tego samego typu (np. tablica przechowująca zmienne typu `String`).

Pętle – for each



```
String[] zakupy = {"jajka", "mleko", "masło", "coś na chleb"};

System.out.println("Lista zakupów : ");
for (String produkt: zakupy) {
    System.out.println("  - " + produkt);
}
```



1. Wewnątrz pętli while (np. 10 razy) pobieraj produkt do kupienia i wprowadzaj go do utworzonej tablicy typu String. Wyświetl wszystkie produkty z wykorzystaniem pętli for oraz for each.
2. Utwórz tablicę przechowującą wartości typu int o rozmiarze zadanym z konsoli. Wypełnij ją wartościami wewnątrz pętli for. Zwróć sumę tych wartości.
3. Stwórz tablicę zawierającą 5 imion. Wewnątrz pętli for-each dopisuj imiona do zmiennej typu String, oddzielając je przecinkami. Wyświetl utworzony łańcuch znaków. Np. „Małgorzata, Jan, Jakub”.
4. *Jak wyżej, ale dopisuj tylko imiona, które składają się z mniej, niż 5 znaków.



1. Napisz program, który dla zadanej tablicy typu `int` zwraca:
 - a) Największą wartość
 - b) Najmniejszą wartość
 - c) Sumę wszystkich wartości
 - d) *Wartość średnią
 - e) *Medianę
2. Napisz program, który zwraca w postaci tablicy zbiór wszystkich liczb mniejszych od zadanej przez użytkownika liczby oraz:
 - a) Podzielnych przez 2
 - b) Podzielnych przez 3
 - c) *Podzielnych przez zadaną przez użytkownika liczbę



```
public class TabliceWielol {

    public static void main(String[] args) {

        int[][] tablicaMnozenia = new int[10][10];

        // Wpisujemy wartości do tablicy
        for (int wiersz = 1; wiersz <= tablicaMnozenia.length; wiersz++) {
            for (int kolumna = 1; kolumna <= tablicaMnozenia.length; kolumna++) {
                tablicaMnozenia[wiersz - 1][kolumna - 1] = wiersz * kolumna;
            }
        }

        System.out.println("Wypisujemy wartości tabliczki mnożenia : ");
        for (int wiersz = 0; wiersz < tablicaMnozenia.length; wiersz++) {
            for (int kolumna = 0; kolumna < tablicaMnozenia.length; kolumna++) {
                System.out.printf("%4d", tablicaMnozenia[wiersz][kolumna]);
            }
            System.out.println("");
        }

    }

}
```



```
public class Varargs {  
  
    public static void main(String[] args) {  
  
        System.out.println("Otrzymane argumenty jeden po drugim : ");  
  
        if (args.length > 0) {  
            for(String arg : args) {  
                System.out.println(arg);  
            }  
        } else {  
            System.out.println("Lista argumentow jest pusta...");  
        }  
  
    }  
  
}
```



ZADANIA Podsumowujące

Na podstawie zadanych z konsoli wartości oraz znaku (*, \$, #...) narysuj:

1. Kwadrat pusty w środku (wyłącznie krawędzie).
2. Prostokąt pusty w środku.
3. Literę „L”, o krawędziach równej długości.
4. *Trójkąt prostokątny.
5. *Jak punkt 1, ale najpierw wprowadź wszystkie elementy do tablicy dwuwymiarowej, a dopiero później je wyświetl.
6. *Kwadrat pusty w środku z jedną przekątną
7. **Kwadrat pusty w środku z dwiema przekątnymi.



1. Napisz aplikację, która:
 - a) Będzie posiadała tablicę jednowymiarową składającą się z 3 elementów typu String: kombinerek, nożyczek i śrubokrętu.
 - b) Pobierze imię użytkownika.
 - c) Wypisze na ekran: „Witaj <imię>! W naszej ofercie mamy: ” + lista elementów z tablicy produktów zdefiniowanych w podpunkcie a, każdy w nowej linii zaczynający się od myślnika + „Co chciałbyś kupić?”
 - d) Za pośrednictwem pętli switch – case przeanalizuj wybór użytkownika, dla opcji default wypisz „Takiego produktu nie mamy w ofercie”
 - e) Dla prawidłowego wyboru usuń dany element z tablicy i potwierdź użytkownikowi wybrany produkt.
 - f) Potwierdź usunięcie elementu z listy poprzez jej ponowne wyświetlenie.
 - g) *Imię użytkownika przekaz przez Varargs jako pierwszy parametr.
 - h) *Produkty do kupienia również.
 - i) W przypadku błędnego wyboru produktu pozwól na ponowny wybór tak długo, aż użytkownik nie napisze „do widzenia”



Programowanie obiektowe

OOP

Autor: Marek Bobcow

Prawa do korzystania z materiałów posiada Software Development Academy



Plan bloku

1. Modyfikatory dostępu
2. Dziedziczenie
3. Kompozycja
4. Abstrakcja
5. Statyczne: pola, metody, klasy
6. Polimorfizm
7. Enum
8. Wyjątki
9. Data i czas
10. String
11. Wyrażenia regularne
12. Operacje na plikach
13. Adnotacje
14. Typy generyczne



Modyfikatory dostępu

Modyfikator	Klasa	Moduł	Subklasa	Świat
public	Tak	Tak	Tak	Tak
protected	Tak	Tak	Tak	Nie
<i>brak modyfikatora</i>	Tak	Tak	Nie	Nie
private	Tak	Nie	Nie	Nie

Modyfikatory dostępu określają czy inne klasy mogą używać poszczególnych pól lub wywoływać określone metody. Wyróżniamy dwa poziomy dostępu:

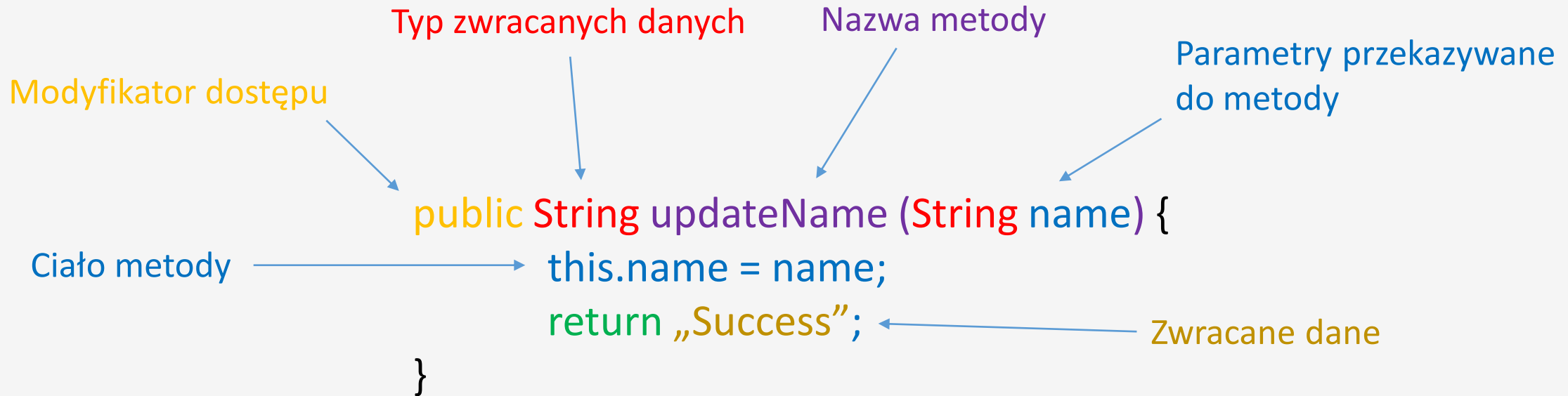
- top level – public lub package-private (brak modyfikatora)
- member level – public, private, protected lub package-private (brak modyfikatora)

Podczas wyboru poziomu dostępu kierujemy się dwiema zasadami:

- używamy modyfikatora **private** chyba, że mamy naprawdę dobry powód, aby tego nie robić,
- unikaj pól **public** za wyjątkiem **stałych**,

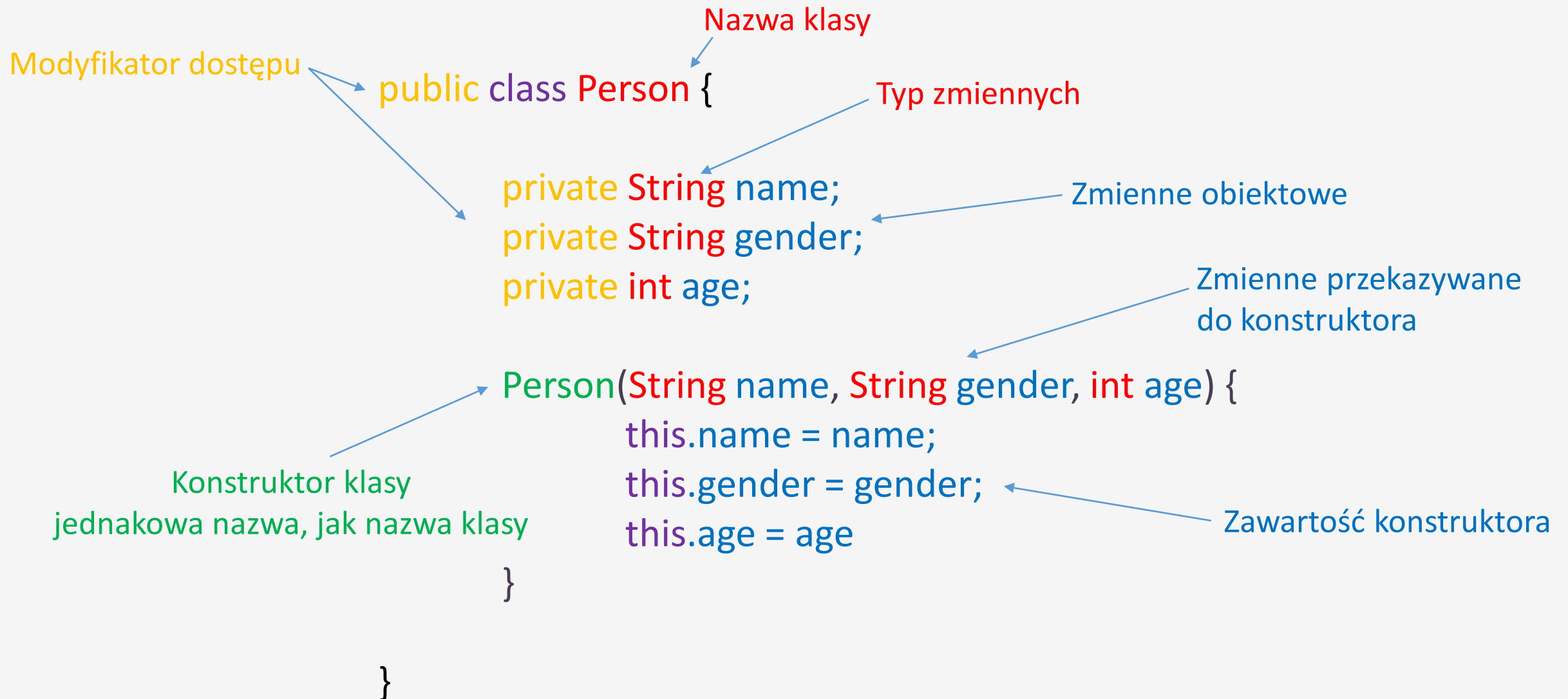


Metody – uzupełnienie informacji





Klasy – uzupełnienie informacji





Obiekt danej klasy – uzupełnienie informacji

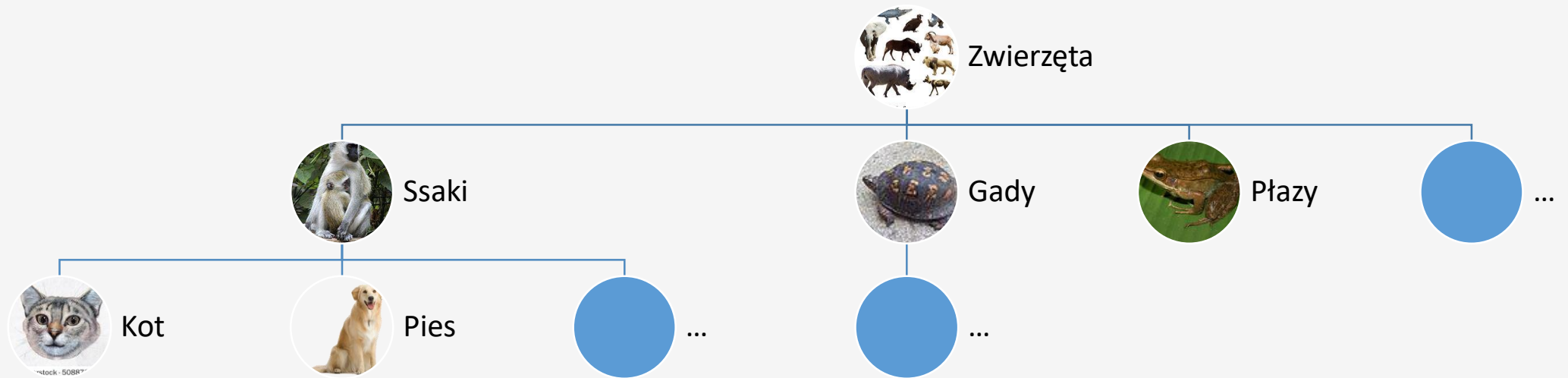




1. Stwórz klasę Pies.
 - a) Dodaj atrybuty rasa, wiek, płeć
 - b) Dodaj metody umożliwiające ustawienie wieku oraz pobranie wieku (getAge, setAge)
 - c) Dodaj konstruktor, który przyjmie wartości dla rasy i płci, a wiek ustawi na 0
 - d) Dodaj metodę powodującą wypisanie na ekran dźwięku wydawanego przez psa „Woof!”
 - e) Z poziomu metody main zaprezentuj działanie poszczególnych metody – utwórz obiekt klasy Pies, zmodyfikuj wiek, wyświetl parametry psa.
 - f) *Stwórz listę składającą się z 2 różnych psów, wypisz wartości ich atrybutów.




1. Stwórz klasę Pokoj.
 - a) Dodaj atrybuty wysokość, szerokość, długość (typu double).
 - b) Dodaj konstruktor, który przyjmie wszystkie wartości.
 - c) Dodaj drugi konstruktor, który przyjmie szerokość i długość a wysokość ustawi na 2,4.
 - d) Dodaj metody obliczające oraz zwracające pole powierzchni oraz objętość pokoju.
 - e) Dodaj metody wyświetlające pole powierzchni oraz objętość pokoju.
 - f) Z poziomu metody main zaprezentuj działanie poszczególnych metod.
 - g) Stwórz listę pokoi. Wyświetl ich parametry.






Dziedziczenie - jest

Nazwa klasy (nadrzędnej)


`class A {`
Ciało klasy
`}`

Dziedziczy po (rozszerza) A


`class B extends A {`
Ciało klasy
`}`

- Klasa dziedzicząca (podrzędna) „dziedziczy” wszystkie elementy klasy nadrzędnej. Nie ma jednak dostępu do pól i metod **prywatnych** (patrz slajd poświęcony modyfikatorom dostępu).
- W celu wywołania konstruktora klasy nadrzędnej z poziomu klasy dziedziczącej należy skorzystać z metody **super()**. Pozwoli ona na przekazanie parametrów do konstruktora klasy nadrzędnej.
- W celu zabezpieczenia klasy przed jej dziedziczeniem możemy skorzystać ze słowa kluczowego **final** przed słowem **class**.
- Dziedziczenie informuje o tym, że element dziedziczący **jest** elementem, po którym dziedziczy. Np. Kot jest Ssakiem, Ssak jest Zwierzęciem.



Dziedziczenie – *jest*

```
class Owoc {

    String witaminy;
    String rodzaj;

    Owoc(String rodzaj, String witaminy, boolean soczysty, boolean moznaJescZeSkorka) {
        this.rodzaj = rodzaj;
        this.witaminy = witaminy;
    }

    void przerobNaSok() {
        System.out.printf("Przerabiamy %s na sok bogaty w witaminy %s", this.rodzaj, this.witaminy);
    }

}

class Jablko extends Owoc {

    Jablko() {
        super("Jablko", "A,D,E,K,C,B6", true, true);
    }

    void zetrzyjNaMus() {
        System.out.println("Jablko starte na mus");
    }

}
```



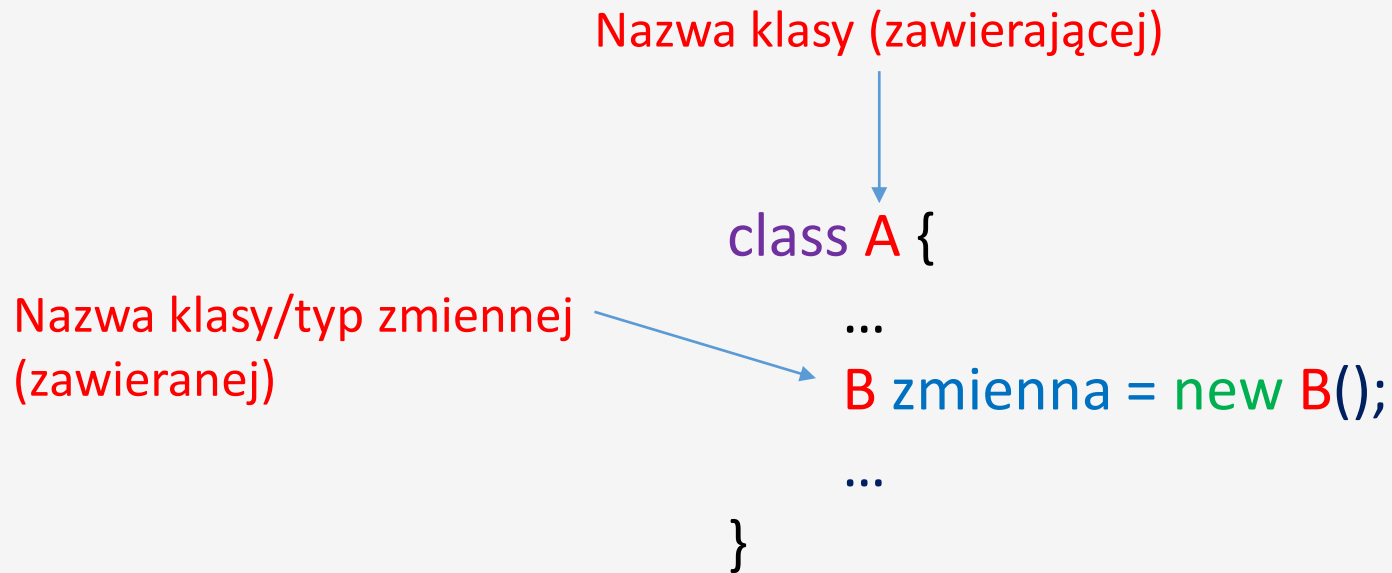
1. Analogicznie do klasy Pies stwórz klasę Kot, zaprezentuj jej działanie.
2. Stwórz klasę nadrzędną, np. ZwierzeDomowe, po której będzie dziedziczył Kot i Pies. Przenieś do klasy ZwierzeDomowe wszystkie wspólne metody i atrybuty.
3. Stwórz tablicę przechowującą obiekt klasy Pies i obiekt klasy Kot. Wypisz wydawane przez nie dźwięki.



1. Stwórz 3 klasy: Czlowiek, Programista, JavaDeveloper.
 - a) Powiąż klasy:
 - Czlowiek jest nadrzędny
 - Programista dziedziczy po Czlowiek
 - JavaDeveloper dziedziczy po Programista
 - b) Dla każdej z klas stwórz konstruktor, który wywoła konstruktor nadrzędny (jeśli istnieje) oraz wypisze informację o swoim wywołaniu, np. dla klasy Czlowiek powinniśmy otrzymać komunikat „Wywołanie konstruktora Czlowiek”.
 - c) Utwórz obiekt klasy JavaDeveloper
 - d) *Za pośrednictwem obiektu JavaDeveloper wywołaj dowolną metodę utworzoną w klasie Czlowiek, która przyjmie jeden parametr. Jaki powinna posiadać modyfikator dostępu?
 - e) **Przeciąż metodę z klasy Człowiek wewnątrz klasy JavaDeveloper tak, aby mogła przyjmować dodatkowe parametry.



Kompozycja - zawiera



- Kompozycja informuje nas, że dana klasa ***zawiera*** obiekty innej klasy.
- Do składowych (metod, pól) podanego obiektu odwołujemy się za pośrednictwem klasy, która obiekt ten zawiera (poprzez odpowiednie metody).
- Np. Człowiek zawiera portfel, kluczyki, dokumenty... (które mogą być innymi klasami).



Kompozycja - zawiera

```
public class Kompozycja1 {  
  
    public static void main(String[] args) {  
        DzialPrawo dzialPrawo = new DzialPrawo();  
        Osoba autor = new Osoba(...);  
        dzialPrawo.dodajKsiazke(new Ksiazka(autor, "Kodeks Pracy", 25.0));  
        int pozycjaKsiazki = dzialPrawo.podajPozycjeKsiazki("Kodeks Pracy");  
        if (pozycjaKsiazki != -1) {  
            System.out.println("Kodeks Pracy znajduje się na : " + pozycjaKsiazki + " pozycji");  
        } else {  
            System.out.println("Kodeksu Pracy nie ma w Dziale Prawo!");  
        }  
    }  
}
```




Kompozycja – zawiera c.d.

```
class DzialPrawo {  
  
    private final int POJEMNOSC_REGALU = 30;  
    private Ksiazka[] ksiazki = new Ksiazka[POJEMNOSC_REGALU];  
    private int id = 0;  
    private String opis;  
  
    public void dodajKsiazke(Ksiazka ksiazka) {  
        this.ksiazki[this.id++] = ksiazka;  
    }  
  
    public int podajPozycjeKsiazki(String tytul) {  
        int pozycja = -1;  
        for (int i = 0; i < this.POJEMNOSC_REGALU; i++) {  
            if (this.ksiazki[i].podajTytul().equals(tytul)) {  
                pozycja = i;  
                break;  
            }  
        }  
        return pozycja;  
    }  
}
```

```
class Ksiazka {  
  
    private Osoba autor;  
    private String tytul;  
    private double cena;  
  
    Ksiazka(Osoba autor, String tytul, double cena) {  
        this.autor = autor;  
        this.tytul = tytul;  
        this.cena = cena;  
    }  
  
    public String podajTytul() {  
        return this.tytul;  
    }  
}
```



Nazwa klasy abstrakcyjnej

↓

```
abstract class A {  
    ...  
    abstract void doSomething();  
    ...  
}
```

Deklaracja metody abstrakcyjnej →

- Klasa abstrakcyjna definiowana jest przez słowo kluczowe **abstract**.
- Pozwala ona na **deklarację** metod bez ich **implementacji** (ciała metody).
- Klasa abstrakcyjna może zawierać zarówno metody abstrakcyjne, jak i „zwykłe” – posiadające implementację.
- W celu skorzystania z klasy abstrakcyjnej należy utworzyć klasę, która po niej **dziedziczy** oraz **implementuje metody abstrakcyjne**.



```
public class Abstrakcje1 {

    public static void main(String[] args) {
        Czlowiek[] ludzie = new Czlowiek[2];
        Nurek nurek = new Nurek(Czlowiek.charakter.RYZYKANT);
        if (nurek.podajTypCharakteru() == Czlowiek.charakter.RYZYKANT) {
            System.out.println("Masz skłonność do ryzyka!");
        }
    }
}

abstract class Czlowiek {

    public enum charakter {
        RYZYKANT, CICHAWODA, WYWAZONY
    }

    abstract charakter podajTypCharakteru();
}
```

```
class Nurek extends Czlowiek {

    private charakter typCharakteru;

    Nurek(charakter typCharakteru) {
        this.typCharakteru = typCharakteru;
    }

    public charakter podajTypCharakteru() {
        return this.typCharakteru;
    }
}
```



1. Zmodyfikuj aplikację dotyczącą zwierząt domowych tak, aby klasa `ZwierzeDomowe` była abstrakcyjna.
2. Utwórz klasę `Kaganiec` (rozmiar, materiał wykonania...), której obiekt będzie zawierała klasa `Pies` (kompozycja). Wyświetl dane dotyczące kagańca za pośrednictwem obiektu `psa`.



Pola statyczne

```
class A {  
    ...  
    Deklaracja zmiennej statycznej → static int value;  
    ...  
}
```

- Zmienna statyczna jest **atrybutem klasy**, NIE atrybutem obiektu.
- Wszystkie obiekty danej klasy **współdzielą zmienne statyczne** – modyfikacja zmiennej statycznej w jednym obiekcie będzie widoczna w pozostałych obiektach danej klasy.
- W celu skorzystania ze zmiennej statycznej nie musimy tworzyć obiektu klasy, która ją zawiera.
- Do zmiennej statycznej odnosimy się poprzez **nazwę klasy**, nie nazwę obiektu.
Np. **A.value = ...;** *zamiast* ~~A a = new A(); a.value = ...;~~.



Pola statyczne

```
public class Statyczne1 {  
  
    public static void main(String[] args) {  
        Towar dalmierz = new Towar("dalmierz");  
        Towar wiertarka = new Towar("wiertarka");  
        Towar bruzdownica = new Towar("bruzdownica");  
        Towar[] listaTowarow = {dalmierz, wiertarka, bruzdownica};  
        for(Towar towar: listaTowarow) {  
            System.out.printf("Towar : %s ma id : %d\n",  
                               towar.podajNazwe(), towar.podajId());  
        }  
    }  
}
```

```
class Towar {  
    private static int liczbaTowarow;  
    private int id = liczbaTowarow++;  
    private String nazwa;  
  
    Towar(String nazwa) {  
        this.nazwa = nazwa;  
    }  
  
    public int podajId() {  
        return this.id;  
    }  
  
    public String podajNazwe() {  
        return this.nazwa;  
    }  
}
```



Metody statyczne

```
class A {  
    ...  
    Definicja metody statycznej → static void sumator() {  
        ...  
    }  
    ...  
}
```

- Metoda statyczna jest **metodą klasy**, NIE metodą obiektu.
- W celu skorzystania z metody statycznej nie musimy tworzyć obiektu klasy, która ją zawiera.
- Metoda statyczna może odwoływać się wyłącznie do innych metod statycznych i korzystać wyłącznie z atrybutów statycznych.
- Do metody statycznej odnosimy się poprzez **nazwę klasy**, nie nazwę obiektu.
Np. **A.sumator()**; *zamiast ~~A a = new A(); a.sumator();~~*.



```
public class MetodyStatyczne1 {  
  
    public static void main(String[] args) {  
  
        System.out.println("Użycie metody klasy bez tworzenia obiektu 5 + 10 = " + Funkcje.sumator(5, 10));  
  
    }  
}  
  
class Funkcje {  
  
    public static int sumator(int a, int b) {  
        return a + b;  
    }  
  
}
```




1. Stwórz kalkulator korzystając z metod statycznych:
 - a) Stwórz najbardziej popularne metody: dodawanie, odejmowanie, mnożenie, dzielenie, reszta z dzielenia wewnątrz klasy „Kalkulator”
 - b) Stwórz mechanizm wyboru przez użytkownika, która metoda zostanie wykonana.
 - c) Zaprezentuj działanie poszczególnych metod
2. Zaprezentuj wykorzystanie zmiennej statycznej, która umożliwi zliczanie utworzonych obiektów danej klasy (np. Pies).



Klasy statyczne

Definicja klasy statycznej

```
class A {  
    ...  
    static class B() {  
        ...  
    }  
    ...  
}
```

- W celu utworzenia obiektu klasy statycznej nie musimy tworzyć obiektu klasy, która ją zawiera.
- Obiekt klasy statycznej tworzymy poprzez **nazwę klasy która ją zawiera**, nie nazwę obiektu.
Np. ... **new A.B()**; *zamiast ~~A a = new A(); new a.B();~~*.
- Klasa statyczna może być zdefiniowana **wyłącznie wewnątrz innej (niestatycznej) klasy**.
- Klasa statyczna może się odwoływać **wyłącznie do pól statycznych** klasy nadrzędnej.
- Klasa statyczna może posiadać zarówno **metody i pola statyczne**, jak i **niestatyczne**.



Klasy statyczne

```
public class KlasyStatyczne1 {  
  
    public static void main(String[] args) {  
        KlasaNadrzedna klasaGlowna = new KlasaNadrzedna();  
        KlasaNadrzedna.KlasaNieStatyczna klasaNieStatyczna =  
            klasaGlowna.new KlasaNieStatyczna();  
        klasaNieStatyczna.wypiszTekst();  
  
        KlasaNadrzedna.KlasaStatyczna klasaStatyczna =  
            new KlasaNadrzedna.KlasaStatyczna();  
        klasaStatyczna.wypiszTekst();  
    }  
}
```

```
class KlasaNadrzedna {  
  
    public static class KlasaStatyczna {  
  
        public void wypiszTekst() {  
            System.out.println("Klasa podrzędna statyczna");  
        }  
    }  
  
    public class KlasaNieStatyczna {  
  
        public void wypiszTekst() {  
            System.out.println("Klasa podrzędna niestatyczna");  
        }  
    }  
}
```



Nazwa interfejsu

interface **A** {

...

Deklaracja metody → void doSomething();

...

}

- Interfejsy pozwalają wyłącznie na **deklarację metod** bez ich **implementacji** (za wyjątkiem metod domyślnych).
- Pozwalają również na definicję **zmiennych**, które są **stałymi** (poprzedzone słowem kluczowym **final**).
- W celu skorzystania z danego interfejsu konieczne jest jego **zaimplementowanie** za pośrednictwem danej klasy – **class X implements A**. „X” to klasa implementująca interfejs, „A” to implementowany interfejs.



```
public class Polimorfizm {

    public static void main(String[] args) {
        Kolo kolo = new Kolo(9.5);
        Prostokat prostokat = new Prostokat(10, 7);
        Ksztalt[] ksztalty = {kolo, prostokat};
        for(Ksztalt ksztalt: ksztalty) {
            System.out.printf("%s ma pole rowne : %.2f i obwod rowny : %.2f\n",
                               ksztalt.getClass().getName(), ksztalt.obliczPole(),
                               ksztalt.obliczObwod());
        }
    }
}
```

```
interface Ksztalt {

    public double obliczPole();
    public double obliczObwod();

}

class Kolo implements Ksztalt {
    private double promien;

    Kolo(double promien) {
        this.promien = promien;
    }

    public double obliczPole() {
        return Math.PI*Math.pow(this.promien, 2);
    }

    public double obliczObwod() {
        return 2*Math.PI*this.promien;
    }

}
```



ZADANIA dla chętnych

1. Bazując na napisanej wcześniej aplikacji:
 - a) Stwórz aplikację – magazyn produktów
 - b) Obsłuż wybór zadania do wykonania:
 - dodawanie nowego produktu: nazwa, cena, liczba dostępnych sztuk
 - wyświetlanie stanu magazynu
 - c) *skorzystaj z kompozycji (Magazyn zawierający Produkt[y])
 - d) **suma wartości poszczególnych/wszystkich produktów w magazynie
 - e) ***aktualizacja stanu magazynu, np.:
 - „kombinerki:30”
 - „kombinerki:+4”

Wymagania:

Rozbicie aplikacji na klasy i metody. Użycie pól statycznych (id produktu). Najpierw jakoś potem dodać „ć”.



```
public class Enum {  
  
    public static void main(String[] args) {  
  
        EnumKolor.Kolor kolor = EnumKolor.Kolor.CZARNY;  
  
        if (kolor == EnumKolor.Kolor.CZARNY) {  
            System.out.println("Tak, nasz kolor to czarny");  
        } else {  
            System.out.println("Nasz kolor jest inny niz czarny");  
        }  
  
    }  
  
}  
  
class EnumKolor {  
  
    public enum Kolor{  
        CZARNY, BIALY, NIEBIESKI;  
    }  
  
}
```



```
try {  
    instrukcja, która może wywołać wyjątek  
} catch (typ pierwszego wyjątku ex) {  
    obsługa pierwszego wyjątku  
} catch (typ drugiego wyjątku ex) {  
    obsługa drugiego wyjątku  
} finally {  
    blok, który wykona się zawsze  
}
```

- Wyjątki służą do **zgłaszania nieprawidłowości w pracy programu**. Brak obsłużenia wyjątku, który wystąpił, doprowadzi do **wyłączenia wątku**, w ramach którego się pojawił. W najprostszej sytuacji spowoduje to wyłączenie programu.
- Należy obsłużyć **wszystkie wyjątki**, które **mają szansę pojawić się** w trakcie pracy programu. Dążymy do tego, aby program mógł zostać **wyłączony wyłącznie w przewidziany przez nas sposób**.



```
import java.util.InputMismatchException;
import java.util.Scanner;

public class Wyjatki {

    public static void main(String[] args) {

        Scanner wejscie = new Scanner(System.in);
        System.out.println("Podaj dowolna wartosc typu int");
        int wprowadzonaWartosc;
        try {
            wprowadzonaWartosc = wejscie.nextInt();
        } catch (InputMismatchException ex) {
            System.out.println("Wprowadzona wartosc nie jest typu int!");
            return;
        }
        System.out.println("Wprowadzona wartosc to : " + wprowadzonaWartosc);
    }

}
```



```
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;

public class Data {

    public static void main(String[] args) {

        LocalDateTime aktualnaDataCzas = LocalDateTime.now();
        System.out.println("Data i czas przed formatowaniem : " + aktualnaDataCzas);

        DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss");
        String dataCzasPoFormacie = aktualnaDataCzas.format(formatter);
        System.out.println("Data i czas po formatowaniu : " + dataCzasPoFormacie);

        String dataCzasString = "2018-08-19 23:32:00";
        LocalDateTime formatDateTime = LocalDateTime.parse(dataCzasString, formatter);

        System.out.println("Otrzymana data i czas : " + dataCzasString);
        System.out.println("Data i czas obiekt : " + formatDateTime);
        System.out.println("Obiekt po formacie : " + formatDateTime.format(formatter));

    }

}
```



```
public class KlasaString {

    public static void main(String[] args) {

        String ciagZnakow = "Lorem ipsum dolor sit amet, consectetur adipiscing elit, " +
            "sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.";
        System.out.println("Nasz ciag znakow : " + ciagZnakow);
        System.out.println("Male litery : " + ciagZnakow.toLowerCase());
        System.out.println("Wielkie litery : " + ciagZnakow.toUpperCase());
        System.out.println("Zamiana Lorem na Tworem : " + ciagZnakow.replace("Lorem", "Tworem"));
        System.out.println("Zamiana litery a na x : " + ciagZnakow.replace('a', 'x'));

        String a = "test";
        String b = "test";
        String c = new String("test");
        if (a == b) {
            System.out.println("Ciagi znakow a i b sa rowne (wskazuja na ten sam obiekt!)");
        }
        if (a.equals(c)) {
            System.out.println("a i c zawieraja jednakowe ciagi znakow!");
        }

    }

}
```



Wyrażenia regularne – Regex

<https://regex101.com/>

```
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class StringRegex {

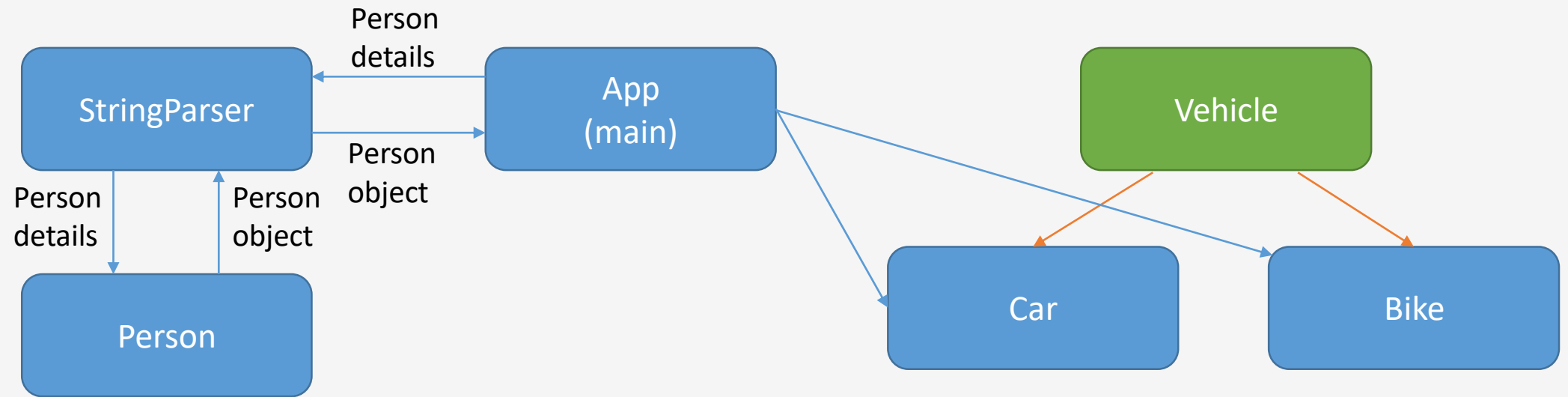
    public static void main(String[] args) {

        String informacje = "Marian Kowalski 600-625-420 80-422 Sopot";
        String pattern = "([a-z,A-Z]+)\\s+([a-z,A-Z]+)\\s+([0-9,\\-]{9,12})\\s+([0-9,\\-]{6})\\s+([a-z,A-Z]{2,})";

        Pattern r = Pattern.compile(pattern);
        Matcher m = r.matcher(informacje);
        if (m.find( )) {
            System.out.println("Imie : " + m.group(1) );
            System.out.println("Nazwisko : " + m.group(2) );
            System.out.println("Telefon : " + m.group(3) );
            System.out.println("Kod pocztowy : " + m.group(4) );
            System.out.println("Miasto : " + m.group(5) );
        } else {
            System.out.println("Nie udało się znaleźć poszukiwanych informacji");
        }

    }

}
```





ZADANIE c.d.

1. Tworzymy aplikację składającą się z 6 klas lub 5 klas i 1 interfejsu.
 - a) Klasa App zawiera metodę main i nic więcej
 - b) Klasa StringParser zawiera statyczną metodę pozwalającą na przetworzenie za pośrednictwem Regex wejściowego tekstu na zmienne niezbędne do utworzenia obiektu klasy Person
 - c) Klasa Vehicle jest klasą abstrakcyjną (*lub interfejsem – dla chętnych), którą dziedziczą (*implementują) klasy Car oraz Bike
2. Przekazujemy do aplikacji tekst ze wszystkimi danymi wymaganymi do realizacji punktu 1b. Za pośrednictwem konsoli lub poprzez wpisanie na sztywno danych do zmiennej typu String.
3. Przetwarzamy podany w pkt.2 tekst na obiekt klasy Person (dalej „osoba”). Osoba kupuje samochód oraz rower.
4. Za pośrednictwem pętli for-each demonstrujemy efekt zakupu, np. „<imię> kupił <co> marki <jakiej> <kiedy>”. „imię” pochodzi z obiektu klasy Person, dane dotyczące kupionych rzeczy pochodzą z samochodu oraz roweru. Moment zakupu powinien zostać zapisany wewnątrz obiektu, którego dotyczy (samochodu, roweru).
5. *Markę i model auta/roweru przechowujemy w Enumie (do wyświetlenia informacji skorzystajcie z metody toString() dostępnej dla wartości Enuma).



```
import java.io.FileNotFoundException;
import java.io.PrintWriter;

public class Zapis1 {

    public static void main (String args[]) throws FileNotFoundException{
        PrintWriter zapis = new PrintWriter("dok_tekstowy.txt");
        zapis.println("Przykładowe zdanie");
        zapis.close();
    }
}
```



```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class Odczyt1 {

    public static void main(String args[]) throws FileNotFoundException{
        File plik = new File("dok_tekstowy.txt");
        Scanner in = new Scanner(plik);

        String tekst = in.nextLine();
        System.out.println(tekst);
    }

}
```




New I/O – zapis do pliku

```
import java.io.IOException;
import java.nio.charset.Charset;
import java.nio.charset.StandardCharsets;
import java.nio.file.Files;
import java.nio.file.Paths;
import java.nio.file.StandardOpenOption;

public class Zapis2 {

    public static void main(String[] args) {

        Charset utf8 = StandardCharsets.UTF_8;

        try {
            Files.write(Paths.get("Plik0.txt"), "Brak informacji o kodowaniu aęźź".getBytes());
            Files.write(Paths.get("Plik1.txt"), "Kodowanie utf-8 aęźź".getBytes(utf8),
                StandardOpenOption.CREATE, StandardOpenOption.APPEND);
        } catch (IOException e) {
            e.printStackTrace();
        }

    }

}
```



ZADANIE

1. Stwórz plik (ręcznie) o zawartości pochodzącej z generatora Lorem Ipsum.
2. Odczytaj zawartość pliku za pośrednictwem java.io.
3. Zmodyfikuj tekst tak aby:
 - a) nie zawierał kropek i przecinków
 - b) był cały zapisany małymi literami
4. Policz wystąpienie dowolnego wyrazu w tekście. Zweryfikuj poprawność.
5. Zapisz przetworzony tekst do dowolnego pliku.

**Osoby, które wykonały zadanie domowe mogą zamiast powyższego zadania:*

- a) dane, które pochodziły z konsoli zastąpić danymi z pliku*
- b) zapisać wszystko to, co program wypisze na ekran – do pliku*



Plik konfiguracyjny - zapis

```
public static void main(String[] args) {

    Properties ustawienia = new Properties();
    OutputStream wyjście = null;

    try {
        wyjście = new FileOutputStream("app.cfg");
        ustawienia.setProperty("uzytkownik", "admin");
        ustawienia.setProperty("haslo", "123456");
        ustawienia.setProperty("port", "4000");
        ustawienia.store(wyjście, null);
    } catch (IOException io) {
        io.printStackTrace();
    } finally {
        if (wyjście != null) {
            try {
                wyjście.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

}
```



Plik konfiguracyjny - odczyt

```
public static void main(String[] args) {

    Properties ustawienia = new Properties();
    InputStream wejscie = null;

    try {
        wejscie = new FileInputStream("app.cfg");
        ustawienia.load(wejscie);
        System.out.println(ustawienia.getProperty("uzytkownik"));
        System.out.println(ustawienia.getProperty("haslo"));
        System.out.println(ustawienia.getProperty("port"));
    } catch (IOException ex) {
        ex.printStackTrace();
    } finally {
        if (wejscie != null) {
            try {
                wejscie.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }

}
```



Adnotacje pozwalają na dodanie dodatkowych danych „*metadanych*” do kodu źródłowego programu.

Wyróżniamy trzy główne grupy (typy) adnotacji:

1. **Instrukcje dla kompilatora** - @Deprecated, @Override, @SuppressWarnings umożliwiające przekazanie do kompilatora dodatkowych informacji.
2. **Instrukcje realizowane w trakcie kompilacji** – umożliwiające współpracę, między innymi, z aplikacjami budującymi oprogramowanie, pliki XML, itp.
3. **Instrukcje realizowane w trakcie pracy programu** – dostępne za pośrednictwem „*Java reflections*” i umożliwiające realizację zadań, które bez ich wykorzystania nie byłyby możliwe.



```
import java.lang.annotation.Inherited;

public class Adnotacje1 {

    public static void main(String[] args) {

    }

}

@Inherited
@interface NaszaDziedziczonaAdnotacja {

}

@NaszaDziedziczonaAdnotacja
class KlasaGlowna {

}

class KlasaPodrzedna extends KlasaGlowna {

}
```



```
public class TypyGeneryczne {

    public static void main(String[] args) {
        Generyczny typ1 = new Generyczny<>();
        Generyczny typ2 = new Generyczny<>();
        typ1.set(25.5);
        typ2.set("Witam!");
        Generyczny[] dowolneElementy = {typ1, typ2};
        for(Generyczny dowolnyElement: dowolneElementy){
            System.out.println("Nasz element ma wartość : " + dowolnyElement.get());
        }
    }
}

class Generyczny<DowolnyTyp> {

    private DowolnyTyp t;

    public DowolnyTyp get(){
        return this.t;
    }

    public void set(DowolnyTyp wartosc){
        this.t=wartosc;
    }
}
```



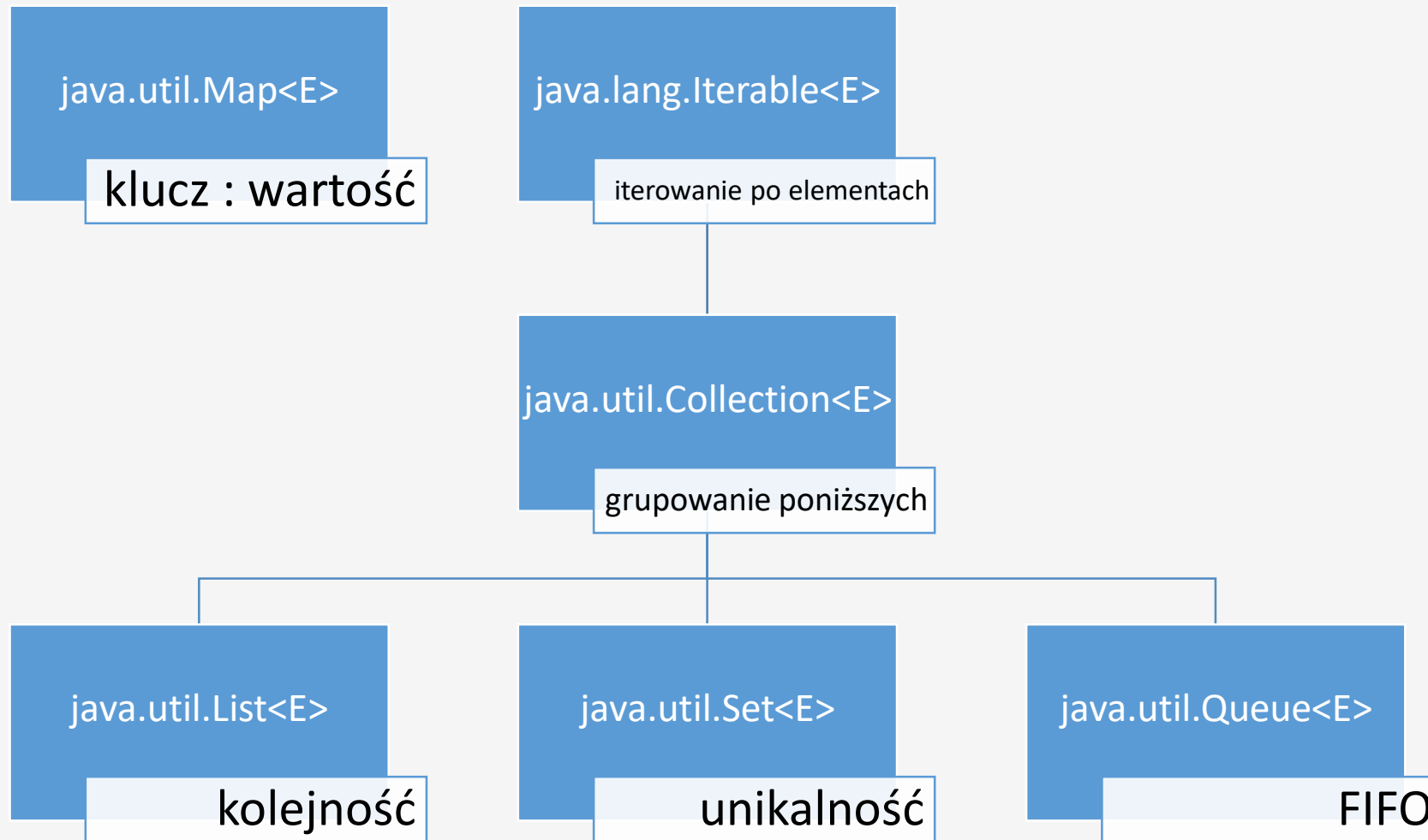
Kolekcje

Autor: Marek Bobcow

Prawa do korzystania z materiałów posiada Software Development Academy



1. Hierarchia kolekcji
2. Map
3. List
4. Set





O czym pamiętać?

1. Definicja zmiennej obiektowej powinna być typu **z warstwy interfejsów** (np. List, Set, Queue, Map..). Implementacja powinna być **typu implementującego dany interfejs** (np. ArrayList, LinkedList, HashMap, TreeSet, PriorityQueue..)

```
Map<String, List<String>> groupsMembers = new HashMap<>()
```

2. Każda z kolekcji może przechowywać elementy **dowolnego typu**. W przypadku **typów prostych** (np. int, double) należy pamiętać o konieczności użycia ich **klas osłonowych** (Integer, Double).
3. Większość struktur danych jest już zaimplementowana w ramach **kolekcji**. Wystarczy znaleźć pasującą do naszych potrzeb.



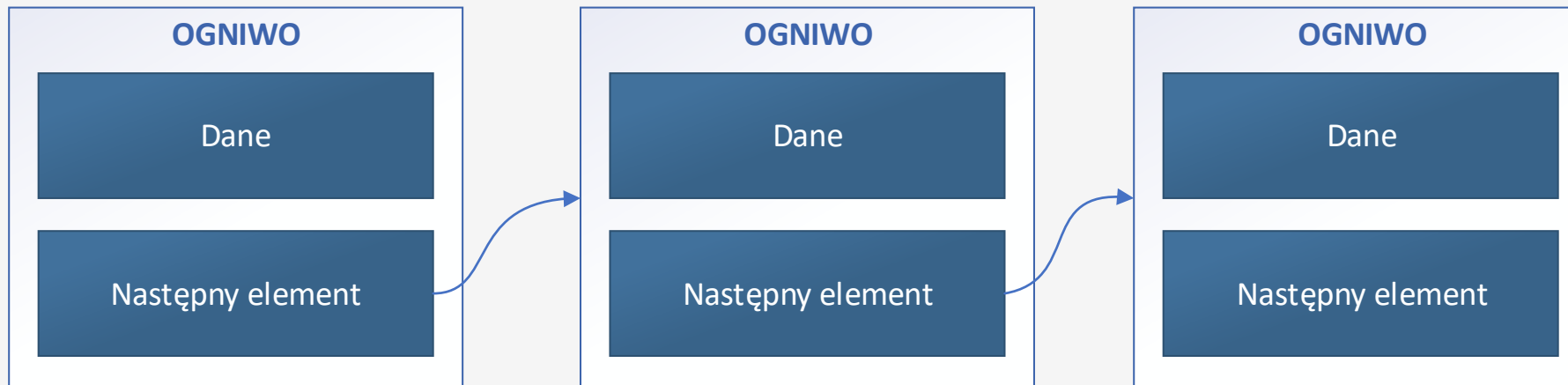
Lista cykliczna

1. **Listy cykliczne** (*ArrayList*) są konstrukcyjnie zbliżone do tablic. W większości sytuacji zaleca się stosowanie list cyklicznych, za wyjątkiem zastosowań, w których często modyfikujemy jej zawartość (dodajemy, usuwamy elementy).
2. **Lista cykliczna** podczas tworzenia ma określoną **pojemność**. Warto przekazać do konstruktora listy cyklicznej docelową pojemność, której nasza lista *najprawdopodobniej* nie przekroczy.
3. **Lista cykliczna** pozwala na odwoływanie się do jej elementów zarówno za pośrednictwem indeksu, jak i konkretnego elementu wchodzącego w jej skład.



Lista wiązana

1. Elementy przechowywane wewnątrz **listy wiązanej** (*LinkedList*) są zgrupowane w formie obiektów, które **poza wartością** (referencją do elementu) posiadają również informację o pozycji elementu **następnego**.
2. Pozwala na szybkie wstawianie i usuwanie elementów w dowolnej lokalizacji (wewnątrz listy).
3. **Listy dwukierunkowe** - implementacja list, której ogniwa poza wskazywaniem na **element następny** posiadają również informację o elemencie **poprzednim**.



Autor: Marek Bobcow

Prawa do korzystania z materiałów posiada Software Development Academy



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

public class Lista {

    public static void main(String[] args) {

        List<String> listaObrazow = new LinkedList<>();
        listaObrazow.add("Krzyk");
        listaObrazow.add("Mona Lisa");
        System.out.println("W naszej galerii posiadamy : ");
        for (String obraz: listaObrazow) {
            System.out.println("  - " + obraz);
        }

    }

}
```



1. Stwórz Listę oraz wyświetl jej rezultat (dane pobieraj z konsoli lub wprowadź na stałe do kodu):
 - a) Zakupów do zrobienia. Jeśli dany element już występuje na liście, to nie powinien zostać dodany.
 - i. Dodaj do powyższego przykładu możliwość „wykreślenia” zakupów, które zostały zrealizowane.
 - ii. Wyświetl tylko te zakupy, które zaczynają się na „m”
 - iii. *Wyświetl tylko te zakupy, których następny produkt na liście zaczyna się na „m”
 - b) Otrzymanych ocen. Wyświetl ich średnią. Liczby nie mogą być mniejsze niż 1 i większe niż 6.
 - c) *Listę list – tabliczka mnożenia (wykorzystując program z prezentacji)
 - d) *Listę Map – gdzie kluczem mapy będzie nazwa budynku, a wartością jej adres (np. „Neptun” : „Grunwaldzka ...”)



```
import java.util.HashMap;
import java.util.Map;

public class Mapa {

    public static void main(String[] args) {

        Map<String, String> panstwa = new HashMap<>();
        panstwa.put("Polska", "Warszawa");
        panstwa.put("Niemcy", "Berlin");

        for (Map.Entry<String, String> slownik: panstwa.entrySet()) {
            String panstwo = slownik.getKey();
            String stolica = slownik.getValue();
            System.out.printf("%s : %s\n", panstwo, stolica);
        }

    }

}
```




1. Stwórz mapę oraz wyświetl jej rezultat (dane pobieraj z konsoli lub wprowadź na stałe do kodu):
 - a) Imion i nazwisk
 - b) Imion i wieku.
 - c) Imion i list znajomych (innych imion).
 - d) *Imion i danych szczegółowych (mapę map), np.
„Marek” :
 „Pesel” : „...”,
 „Nazwisko” : „...”.



```
import java.util.ArrayList;
import java.util.HashSet;
import java.util.Set;
import java.util.TreeSet;

public class Zbiory {

    public static void main(String[] args) {

        Set<String> odwiedzoneMiasta = new HashSet<>();
        odwiedzoneMiasta.add("Zgierz");
        odwiedzoneMiasta.add("Gdynia");
        odwiedzoneMiasta.add("Gdańsk");
        odwiedzoneMiasta.add("Sopot");
        odwiedzoneMiasta.add("Gdańsk");
        System.out.println("Odwiedzone miasta: ");
        for (String miasto: odwiedzoneMiasta) {
            System.out.println("  - " + miasto);
        }
        System.out.println("Odwiedzone miasta (posortowane): ");
        for (String miasto: new ArrayList<String>(new TreeSet(odwiedzoneMiasta))) {
            System.out.println("  - " + miasto);
        }
    }
}
```



1. Jedziemy na wakacje.
 - a) Wewnątrz pętli pobieramy informacje od użytkownika jaki kraj odwiedza, a następnie jakie miasta w danym kraju odwiedził – korzystamy z kolekcji HashMap jako kolekcji nadrzędnej. Kluczem jest String (państwo), a wartością Lista (miasta).
 - b) Pobieramy miasta tak długo, aż nie otrzymamy od użytkownika „jadę dalej”, następnie pobieramy informację o nazwie kolejnego odwiedzanego państwa, itd.
 - c) Po otrzymaniu komunikatu „wracam do domu” kończymy pobieranie danych i wyświetlamy odwiedzone państwa i miasta za pośrednictwem zagnieżdżonych pętli for each.
 - d) *Pozwalamy na maksymalnie dwukrotne odwiedziny danego miasta (na przykład jadąc tam i z powrotem).
 - e) *Jeśli dane miasto znajduje się na liście dwukrotnie, to wyświetlamy je tylko jeden raz z informacją „(tam i powrót)”. Skorzystaj ze zbioru Set.
2. Planujemy wakacje (i robimy korektę po powrocie).
 - a) Tworzymy słownik (j/w), który zawiera państwa i miasta, które chcemy odwiedzić oraz ją wyświetlamy.
 - b) Następnie pobieramy w pętli od użytkownika informacje, których państw/miast nie udało mu się odwiedzić – usuwamy je z naszego słownika.
 - c) Wyświetlamy odwiedzone państwa i miasta.
 - d) *Wyświetlamy, czego nie udało nam się odwiedzić.

Zaprojektuj aplikację tak, aby wyświetlanie wyników (i inne powtarzalne bloki kodu) było wykonywane za pośrednictwem odpowiednio nazwanych funkcji.



ZADANIA – powtórka (dom)

1. List
 - a) Wprowadź z linii komend 10 dowolnych wyrazów do listy
 - b) Wyświetl jej zawartość
 - c) Wewnątrz pętli (while) pobieraj od użytkownika informację, który element usunąć. Wyświetlaj efekt po każdym usunięciu elementu listy.
 - d) Wyświetlanie zawartości listy przenieś do zewnętrznej funkcji, która jako parametr będzie otrzymywała listę, np. **public void showList(List<String> list) {...}**
2. Map
 - a) Stwórz mapę postaci z bajek, gdzie kluczem będzie nazwa bajki, a wartością jej główna postać. Niezbędne dane pobieraj z konsoli.
 - b) Wyświetl utworzoną mapę.
 - c) Wewnątrz pętli (while) pobieraj od użytkownika informację, który element usunąć. Wyświetlaj efekt po każdym usunięciu elementu mapy.
 - d) *Rozwiń punkt „a” tak, aby możliwe było wprowadzanie wielu postaci z bajek. (Map<String, List<String>>)
 - e) *j/w + postacie z bajek powinny być obiektami
3. Set
 - a) Stwórz zbiór składający się z kolorów – podawanych od użytkownika.
 - b) Zaprezentuj usuwanie poszczególnych elementów ze zbioru.
 - c) Wyświetl zbiór przed i po sortowaniu.



ZADANIA – powtórka dla chętnych (dom)

1. *Dane osobowe
 - a) Stwórz plik zawierający dowolne dane osobowe (imię, nazwisko, numer telefonu). Dane poszczególnych osób powinny znajdować się w kolejnych liniach.
 - b) Pobierz dane z pliku i utwórz na ich podstawie obiekty osób (w dowolny sposób – Regex, String.split...).
 - c) Wprowadź utworzone obiekty do ArrayList lub Map (<numer linii> : <Osoba>).
 - d) Zaprezentuj uzyskane dane.
2. **Statystyka
 - a) Stwórz plik, który będzie zawierał dowolny tekst.
 - b) Stwórz statystykę wyrazów zawartych w tekście.
 - c) Wyświetl liczbę wystąpień poszczególnych wyrazów w formie <wyraz> : <liczba wystąpień>
 - d) *j/w posortowane



ZADANIA – powtórka (zajęcia)

1. Stwórz mapę, gdzie kluczem będzie pracownik, a wartością jego manager.
 - a) Klucz i wartość są typu String
 - b) Klucz i wartość są typu „Employee” oraz „Manager”
 - c) *Klucz jest typu „Manager”, wartość jest listą przechowującą typ „Employee”
 - d) *Pozwól na zwolnienie danego pracownika, wyświetl rezultat
 - e) *Pozwól na zatrudnienie nowego pracownika, wyświetl rezultat
2. Stwórz listę przechowującą dany znak, np. „*”.
 - a) Korzystając z listy narysuj linię poziomą.
 - b) Narysuj linię pionową
 - c) Narysuj kwadrat pełen gwiazdek.
 - d) *Wewnątrz pętli pozwól użytkownikowi na wybór „dodaj”/”usuń” „wiersz”/”kolumnę” – wyświetlaj efekt po każdym wyborze.



Programowanie współbieżne i równoległe

Optymalne zagospodarowanie czasu procesora

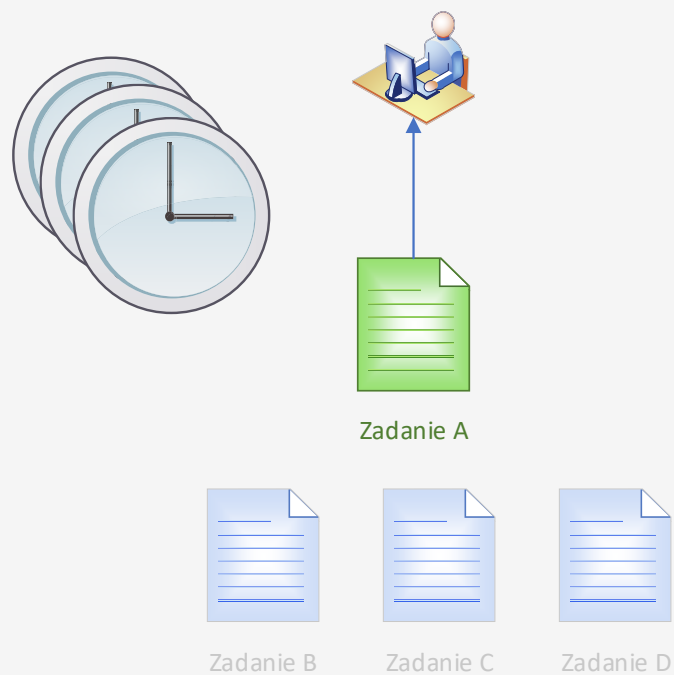


1. Programowanie sekwencyjne i równoległe
2. Runnable
3. Callable

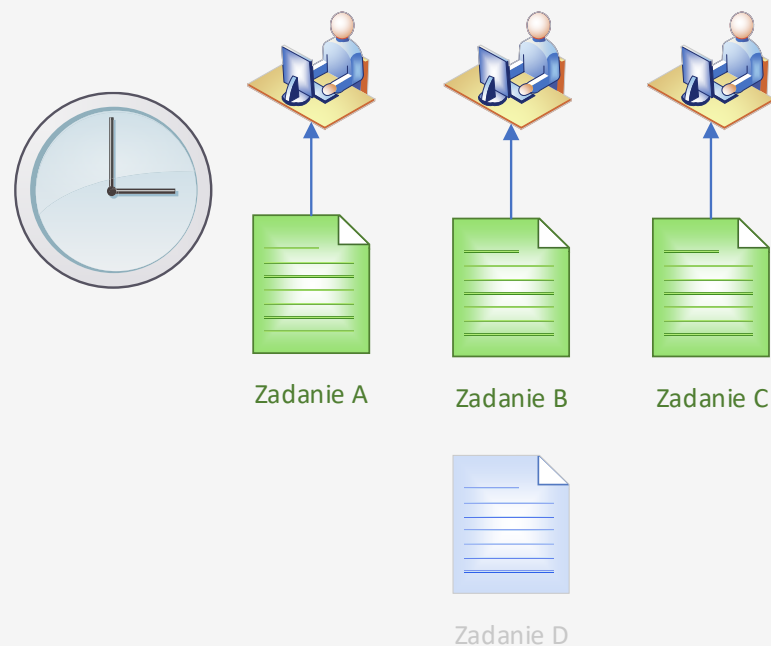


Wstęp

Sekwencyjnie



Czy równolegle?





Runnable

```
class Zadanie implements Runnable {

    private String nazwa;
    private boolean aktywny = true;

    public Zadanie(String nazwa) {
        this.nazwa = nazwa;
    }

    public void wylacz() {
        System.out.println(this.nazwa + " : 16:00 - idę do domu!");
        this.aktywny = false;
    }

    @Override
    public void run() {
        while(this.aktywny) {
            System.out.println(this.nazwa + " : pracuje!");
            try {
                Thread.sleep(1000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```
import java.util.concurrent.TimeUnit;

public class Watki1 {

    public static void main(String[] args) {
        Zadanie[] zadania = new Zadanie[3];
        zadania[0] = new Zadanie("Pracownik A");
        zadania[1] = new Zadanie("Pracownik B");
        zadania[2] = new Zadanie("Pracownik C");
        for (Zadanie zadanie: zadania) {
            new Thread(zadanie).start();
        }
        try {
            TimeUnit.SECONDS.sleep(10);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        for (Zadanie zadanie: zadania) {
            zadanie.wylacz();
        }
    }
}
```



1. Stwórz klasę implementującą interfejs Runnable, która jako parametr (do konstruktora) przyjmie tekst. Wewnątrz metody run wyświetl wprowadzony tekst.
 - a) Zaprezentuj działanie wątku dla pojedynczego obiektu.
 - b) Dla kilku obiektów.
 - c) Dodaj różne wartości uśpienia (jako parametr do konstruktora) dla poszczególnych obiektów. Zaprezentuj działanie.
2. Zmodyfikuj powyższą klasę tak, aby zawartość metody run była realizowana w pętli while.
 - a) Co kilka sekund wyświetl dowolną informację.
 - b) W dowolnym momencie zatrzymaj wątek – wyświetl informację o jego zatrzymaniu.
 - c) W trakcie pracy wątku zmodyfikuj wyświetlany tekst (za pośrednictwem dodatkowej metody).
3. **Do konstruktora każdego wątku podaj uchwyt do wspólnej kolejki.
 - a) Zaprezentuj wymianę danych pomiędzy wątkami w myśl zasady – kto pierwszy, ten lepszy.



1. Stwórz klasę implementującą interfejs Runnable (implementującą metodę run):
 - a) Wewnątrz metody run wyświetl „Hello!”
 - b) Utwórz obiekt klasy.
 - c) Uruchom wątek otrzymujący utworzony obiekt jako parametr (**new Thread(<obiekt>).start()**)
 - d) Stwórz kilka obiektów, uruchom dla każdego z nich osobny wątek.
 - e) Dodaj konstruktor do stworzonej klasy, który przyjmuje wartość typu int.
 - f) Do wyświetlanych danych wewnątrz metody run dodaj otrzymaną wartość (Hello + wartość).
 - g) Dodaj metodę do klasy, która zmodyfikuje wartość typu int.
 - h) Do metody run dodaj pętlę while, wewnątrz której będzie wypisywana zmodyfikowana wartość typu int co kilka sekund.
 - i) Dodaj możliwość wyłączenia wątku (zmiany warunku sprawdzanego przez pętlę while).



```
import java.util.concurrent.*;

public class Watki2 {

    public static void main(String[] args) throws InterruptedException, ExecutionException {
        ExecutorService executorService = Executors.newSingleThreadExecutor();

        System.out.println("Zabieram się do pracy!");
        Future<String> future = executorService.submit(() -> {
            Thread.sleep(5000);
            return "Praca skończona!";
        });

        while(!future.isDone()) {
            System.out.println("Pracuję..");
            Thread.sleep(1000);
        }

        String wynik = future.get();
        System.out.println(wynik);

        executorService.shutdown();
    }
}
```



ZADANIE dla chętnych

1. Jesteś kierownikiem zmiany.
 - a) Posiadasz 5 pracowników.
 - b) Zasymuluj sytuację, w której każdy z nich przychodzi o innej godzinie do pracy.
 - c) Każdy pracownik, po przyjściu do pracy, powoduje wyświetlenie informacji „<imię> : zabieram się do pracy. Przyszedłem o <czas HH:MM>”.
 - d) Co 10 sekund pracownik wyświetla „<imię> : wciąż pracuje!”.
 - e) Wewnątrz pętli, co 30 sekund, wypuszczamy jednego z pracowników do domu (pamiętajcie o zatrzymaniu wątku!) i usuwamy pracownika z listy „aktywnych”.
 - f) Podczas wypuszczenia pracownika do domu wyświetlamy „<imię> : już <czas HH:MM>, pora iść do domu!”
 - g) *Podczas wypuszczania danego pracownika do domu pośpiesz pozostałych. Od tego momentu wyświetlają informację o pracy o 2 sekundy szybciej.
 - h) *Kierownik decyduje, w jakiej kolejności wypuści pracowników (np. za pośrednictwem wcześniej zdefiniowanej listy)



Programowanie funkcyjne

Autor: Marek Bobcow

Prawa do korzystania z materiałów posiada Software Development Academy



1. Optional
2. Lambda Expression
3. Streams
4. Podsumowanie *Wprowadzenia do języka Java*



```
import java.util.Optional;

public class Optional1 {

    public static void main(String[] args) {
        Programista programista = new Programista();
        System.out.println("Nie podaliśmy języka programisty...");
        if(programista.podajGlownyJezyk() != null) {
            System.out.println("Główny język programisty to : "
                               + programista.podajGlownyJezyk());
        }
        System.out.println("... więc nic się nie wyświetli");
        System.out.println("Chyba, że zastosujemy Optional!");
        System.out.println("Główny język programisty to : "
                           + programista.podajGlownyJezykOpt().orElse("Java!"));
    }
}
```

```
class Programista {

    private String jezyk;

    public String podajGlownyJezyk() {
        return this.jezyk;
    }

    public Optional podajGlownyJezykOpt() {
        return Optional.ofNullable(this.jezyk);
    }

    public void ustawGlownyJezyk(String jezyk) {
        this.jezyk = jezyk;
    }

}
```



Lambda expression

Wyrażenia lambda:

1. To blok kodu przekazywany pomiędzy fragmentami programu.
2. Wykorzystywane są w celu **opóźnionego wykonywania procedur**.
 - a) wykonanie kodu w oddzielnym wątku
 - b) wielokrotne wykonanie kodu
 - c) wykonanie kodu w odpowiednim momencie
 - d) **wykonanie kodu w odpowiedzi na zdarzenie**
3. Składają się z:
 - a) bloku kodu
 - b) parametrów
 - c) wartości wolnych zmiennych
4. Nie mogą:
 - a) modyfikować zmiennych spoza ich zakresu
 - b) wykorzystywać zmiennych spoza ich zakresu
5. Wykorzystują wyłącznie **faktycznie finalne zmienne**.

*Cay S. Horstmann – Java Podstawy

Autor: Marek Bobcow

Prawa do korzystania z materiałów posiada Software Development Academy



Lambda Expression

```
public class Lambda1 {  
  
    interface PrzykladLambda1 {  
        String podajPowitanie(String imie);  
    }  
  
    interface PrzykladLambda2 {  
        int wynikMnozenia(int x, int y);  
    }  
  
    public static void main(String[] args) {  
        PrzykladLambda1 powitanie = (imie) -> "Witaj, " + imie + "!";  
        System.out.println(powitanie.podajPowitanie("Marek"));  
  
        PrzykladLambda2 mnozenie = (x, y) -> x * y;  
        System.out.println("Wynik mnozenia 10 * 5 = " + mnozenie.wynikMnozenia(10, 5));  
    }  
}
```



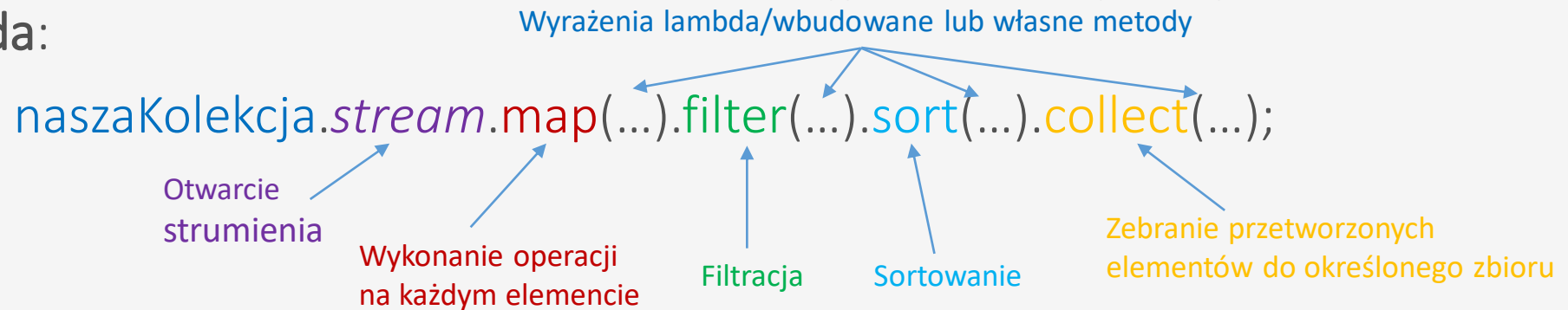
Stwórz i zaprezentuj działanie wyrażeń lambda:

1. Dodawanie, odejmowanie, mnożenie, dzielenie.
2. Suma elementów (typu int) listy.
3. Liczba wyrazów w wyrażeniu wejściowym (lista zawierająca elementy typu String).
4. *Lista przed i po sortowaniu (skorzystaj z klasy Arrays oraz wyrażenia lambda: `String::compareToIgnoreCase` jako komparatora)



Strumienie

- Strumienie można stosować do każdej struktury dziedziczącej po klasie `Collection`.
- Pozwalają na **wydajne wykonywanie** różnych **operacji na zbiorach danych**, m.in.: filtrowanie, sortowanie, przetwarzanie, wyświetlanie, grupowanie, konwersję na inny zbiór danych.
- Na **otwartym strumieniu** możemy wykonać **sekwencyjnie wiele operacji**, np. korzystając z **wyrażeń lambda**:



- Ostatnią instrukcją, jaką należy wykonać w celu zamknięcia strumienia jest np. **forEach(...)** lub **collect(...)** w celu pogrupowania przetworzonych danych w odpowiednią (nową) strukturę lub jej wyświetlenie.



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

public class Stream1 {

    public static void main(String[] args) {
        Dieta warzywna = new Dieta("warzywna", 1000);
        Dieta wywazona = new Dieta("wywazona", 2200);
        Dieta naMase = new Dieta("Na mase", 3500);
        List<Dieta> diety = new ArrayList<>();
        diety.add(warzywna);
        diety.add(wywazona);
        diety.add(naMase);

        List<Dieta> zdroweDiety = diety.stream()
            .filter(dieta -> dieta.podajLiczbeKalorii() < 2500)
            .collect(Collectors.toList());
        for (Dieta dieta: zdroweDiety) {
            System.out.println(dieta.podajNazwe());
        }
    }
}
```

```
class Dieta {
    private int liczbaKalorii;
    private String nazwa;

    Dieta(String nazwa, int kalorie) {
        this.nazwa = nazwa;
        this.liczbaKalorii = kalorie;
    }

    public int podajLiczbeKalorii() {
        return this.liczbaKalorii;
    }

    public String podajNazwe() {
        return this.nazwa;
    }
}
```



Zadania podsumowujące

Autor: Marek Bobcow

Prawa do korzystania z materiałów posiada Software Development Academy



Przed rozpoczęciem: podziel projekt na możliwie małe kawałki, funkcjonalności dodawaj iteracyjnie, stwórz bardzo podstawowe klasy, sprawdź poprawność poszczególnych elementów.

1. Stwórz fabrykę, która zawiera Kierownika zmiany (Manager), pracowników (Worker) oraz *Dyrektora.
 - a) pętle, warunki – dodawanie pracowników, wyświetlanie aktualnie pracujących (co robią), pobieranie rozkazów od użytkownika.
 - b) oop – wszyscy pracownicy (włączając dyrektora), dziedziczą po wspólnej klasie (Employee). Każdy z pracowników może posiadać narzędzie pracy (np. Hammer, Laptop, PointingFinger).
 - c) kolekcje – odpowiednie grupowanie danych
 - d) *wątki – użyj pętli while wyłącznie do pobierania ewentualnych rozkazów od użytkownika, skorzystaj z wątków do wyświetlania stanu/wprowadzania modyfikacji dla każdego obiektu.
 - e) **dodatkowe funkcjonalności – np. wyświetlanie informacji o tym kto ile zarobił od początku pracy (naliczanie co kilka sekund), „kolejno odlicz” – posortowana lista pracowników (po nazwisku), dane pracowników mogą być pobrane z pliku podczas startu programu, ...
 - f) **obsługa klasy Dyrektor – wyświetlanie co jakiś czas informacji „wszyscy pracują – świetnie!”, „gdzie jest <nazwisko>?!” (np. w przypadku, gdy dany pracownik zakończy pracę), ...
2. Wybierz własny projekt, który będzie wykorzystywał zdobytą wiedzę (pętle, warunki, oop, kolekcje, *wątki..). Np. Hogwart, Warsztat Samochodowy, Komisariat, Apteka...



Dziękuję za uwagę!