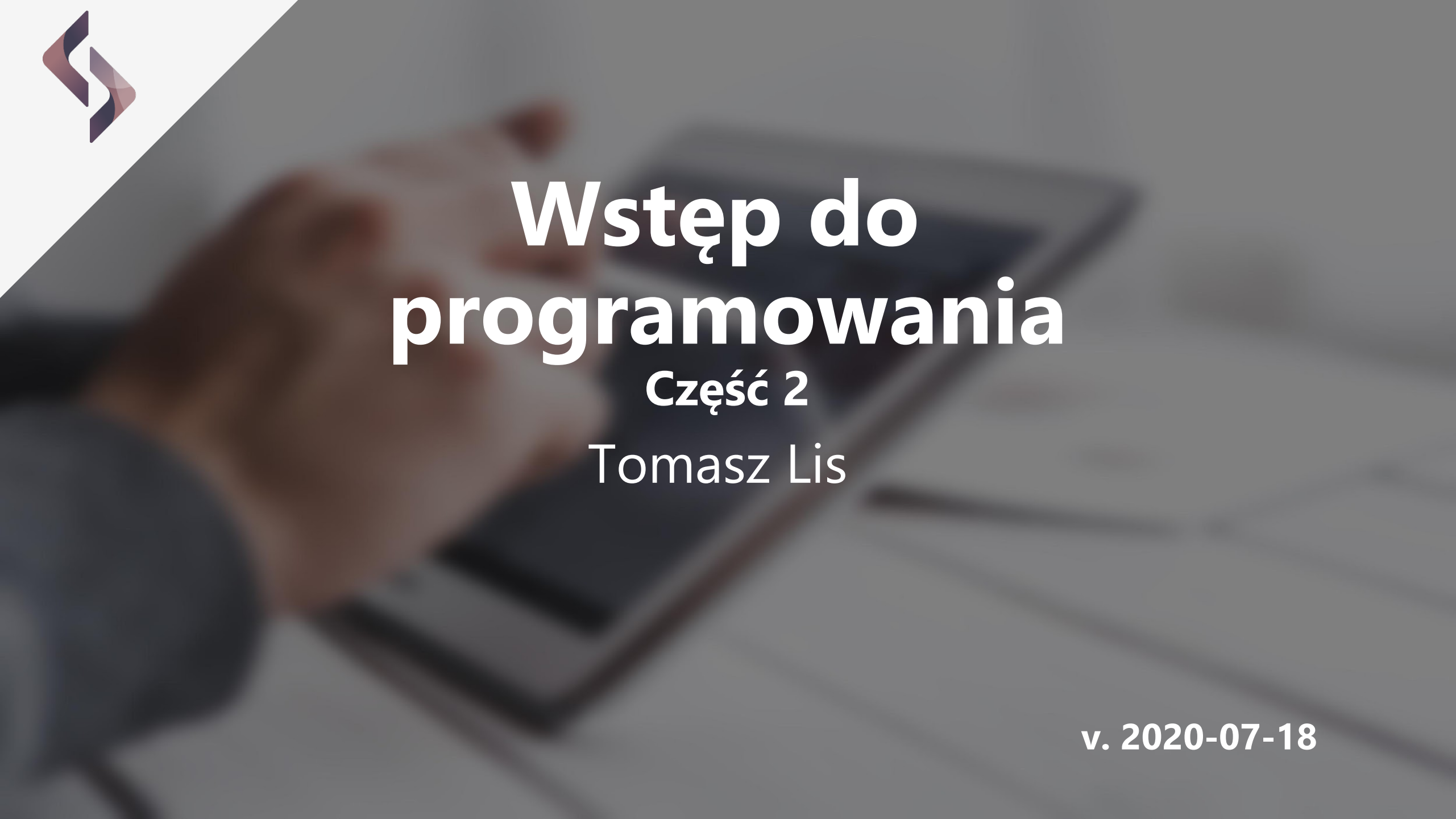


Wstęp do programowania

Część 2

Tomasz Lis

v. 2020-07-18

Plan zajęć

1. Definiowanie klasy
2. Deklaracja pól i inicjalizacja pól
3. Deklaracja metod i ich parametrów
4. Konstruktory, gettery i settery

Definiowanie klasy

Podstawowa składnia Klasy

Podstawową konstrukcję klasy już znamy. Wygląda ona następująco:

```
public class SimpleClass {}
```

W składni wyróżniamy:

1) Modyfikator widoczności : `public,` lub modyfikator domyślny – jeśli nie napiszemy private ani public, klasa jest widoczna na poziomie pakietu
2) Słowo kluczowe: `class`
3) Nazwy klasy: `SimpleClass`
4) Nawiasów klamrowych (`{}`)
5) Ciała klasy, w przykładzie wyżej ciało klasy jest puste – jest to obszar pomiędzy nawiasami klamrowymi

Klasa dziedzicząca

Podstawową konstrukcję klasy już znamy. Wygląda ona następująco:

```
public class ExtendedClass extends SimpleClass{}
```

Klasa dziedzicząca musi wskazywać klasę po której dziedziczy.
Deklarację dziedziczenia wykonujemy słowem kluczowym:

```
extends
```

Po słowie kluczowym znajduje się nazwa klasy, po której dziedziczymy.
W powyższym przykładzie jest to:

```
SimpleClass
```

Czy pamiętamy co klasa `ExtendedClass` odziedziczy po klasie `SimpleClass`?

Klasy zagnieżdżone

W niektórych przypadkach potrzebne jest stworzenie klasy wewnętrznej. Wówczas użyjemy składnię:

```
public class OuterClass {
    protected class InnerClass{}
}
```

Klasa zagnieżdżona jest definiowana bardzo podobnie do zwykłej klasy. Z pewnymi różnicami:

1) Modyfikator widoczności może być spośród: `public`, `private`, `protected` lub modyfikator domyślny
2) Klasa nazywamy wewnętrzną gdy znajduje się w ciele innej klasy

Pakiet

Klasy umieszczamy w pakietach. Wiemy już, że jeśli nie zadeklarujemy modyfikatora widoczności to klasa jest widoczna z poziomu pakietu.
Ale jak zadeklarować pakiet, w którym znajduje się klasa?
Składnia wygląda następująco:

```
package pl.tlis.classes;
public class TestClass { }
```

W przykładzie powyżej widzimy klasę `TestClass`, która znajduje się w pakiecie `pl.tlis.classes`. Wyróżniamy tu dwa elementy składniowe:

1) Słowo kluczowe:
2) Nazwa pakietu:

Oczywiście jak każdą linię, tak i linię deklaracji pakietu należy zakończyć średnikiem (`;`)

Import

Wiemy już, że klasy są umieszczane w pakietach. Domyślny modyfikator powoduje widoczność z poziomu pakietu. Aby uzyskać dostęp do klasy z innego pakietu musimy ją zaimportować. Aby zaprezentować jak wygląda import zmodyfikujemy klasę z poprzedniego slajdu tak aby dziedziczyła po klasie, która jest w innym pakiecie:

```
package pl.tlis.classes;
import pl.tlis.classes.outer.OuterClass;
public class TestClass extends OuterClass { }
```

W przykładzie powyżej widzimy, że klasa `TestClass` dziedziczy po klasie OuterClass. Pojawia się jednak nowy element składniowy zwany importem, składa się on z:

1) Słowa kluczowego: `import`
2) Nazwy pakietu wraz z nazwą klasy: `pl.tlis.classes.outer.OuterClass`

Oczywiście jak każdą linię, tak i linię deklarację importu należy zakończyć średnikiem (`;`)

Deklaracja pól

Pola klasy

Znamy już podstawową składnię klasy, w ramach podstaw programowania poznaliśmy też pojęcie zmiennej oraz typy prymitywne i wyliczeniowe.
Pola w klasie deklarowane są bardzo podobnie do zmiennych z pewnymi różnicami.
Zacznijmy od kilku przykładowych pól klasy:

```
public class ClassWithFields {
    public static final String CONSTANT = "Stała";
    public int id;
    String name;
    private ClassWithFields selfRefference;
    protected static char favoriteLetter;
}
```

Można zauważyć że pole klasy składa się ze słów kluczowych związanych z:
1) modyfikatorem dostępu (`public, private, protected` czy domyślny)
2) deklaracją `static` – przynależność do klasy a nie konkretnego obiektu
3) słowa `final` – oznaczającego, że raz zainicjalizowane pole nie może ulec zmianie. Takie pola nazywamy stałymi czy też po angielsku constant.
4) typu lub klasy obiektu referencjonowanego (`String, int, char`)
5) nazwy pola (*`favoriteLetter`*, `selfRefference, name, id,` *`CONSTANT`*)

Pola klasy a pola instancji

Pola w klasie mogą przynależeć do całej klasy lub do instancji klasy (obiektu).

W nomenklaturze języka mówimy o:

1) Polach statycznych (static field) – to właśnie te, które należą do całej klasy i są wraz z nią inicjalizowane
2) Pola instancyjne (instance field) – to pola, które powstają wraz z nowo utworzonym obiektem

Inicjalizacja pól klasy

Wartości pól mogą zostać zainicjalizowane na kilka sposobów:

1) Wraz z deklaracją
2) W bloku instancyjnym lub bloku statycznym
3) W konstruktorze – Konstruktorom poświęcam oddzielną sekcję
4) W dowolnej metodzie – zobaczymy rodzaje metod w tym wyspecjalizowane później

Na tym etapie zajmiemy się pierwszymi dwoma sposobami.

Deklaracja wraz z inicjalizacją

Poniżej pokazuję kilka przykładów deklaracji pól wraz z ich inicjalizacją:

```
public class ClassWithFields {
    public String name = "Wartość";
    public int id = 1;
    public char favoriteLetter = 'a';
    private float cost = 10.50f;
    private double biggerCost = 100005151500.10d;
    protected long externalId = 241512;
}
```

Jak widać inicjalizacja pola jest analogiczna do inicjalizacji zmiennej.
Z tą różnicą, że w przypadku zmiennych inicjalizowane są one na czas wywołania metody. W przypadku pól pola zainicjalizowane wraz z deklaracją podstawione będą:

1) Przy utworzeniu obiektu – w przypadku pól instancyjnych
2) Raz na czas życia klasy – w przypadku pól statycznych

Stałe

Słowo final można użyć oczywiście zarówno dla pól instancji jak i pól statycznych. Zachowanie będzie oczywiście inne:

1) W przypadku pola instancyjnych – stała zostanie podstawiona dokładnie raz na każde utworzenie obiektu i nie będzie mogła być później nadpisana
2) W przypadku pola statycznego – inicjalizacja będzie miała miejsce raz na czas życia programu dokładnie wtedy kiedy dana klasa zostaje utworzona w środowisku uruchomieniowym.

Przykłady:

```
private static final String CONSTANT = "Stała dla klasy";
private final String constant = "Stała dla obiektu";
```

Inicjalizacja w blokach kodu

Zmienne statyczne jak i instancyjne można deklarować w blokach kodu.
Stałe mogą być zainicjalizowane w bloku kodu o ile nie zostały wcześniej przypisane wraz z deklaracją.
Przykłady:

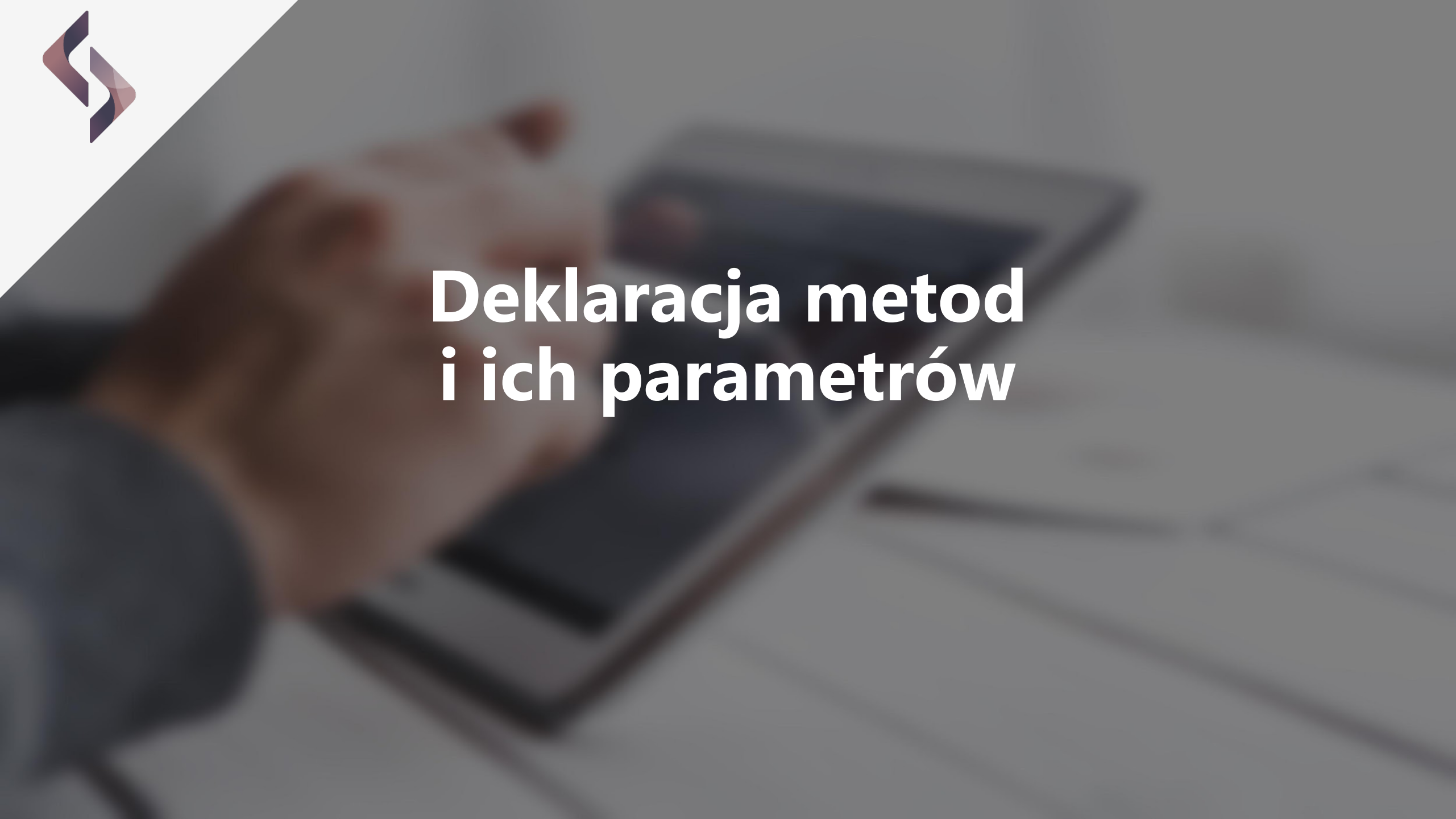
```
private static final String CONSTANT;
private static final String OTHER_CONSTANT;
private static final int INT_CONSTANT;
static {
    INT_CONSTANT = 1;
    CONSTANT = "Stała dla klasy";
    OTHER_CONSTANT = CONSTANT;
}
private final String constant;
private final int intConstant;
{
    constant = "Stała dla obiektu";
    intConstant = 1;
}
```

Operacje w blokach kodu

Pola które nie są stałe również mogą być przypisane w blokach kodu.

Przykłady:

```
private static String nonConstantString;
private static int nonConstantInt;
static {
    nonConstantString = "AAA";
    nonConstantInt = 0;
}
private int instanceInt;
{
    nonConstantInt += 2;
    instanceInt = 0;
}
```



Deklaracja metod i ich parametrów

Podstawowa składnia metody

Składnię metody częściowo poznaliśmy tworząc metodę main. Omówmy podstawową składnię oraz jej elementy na innym przykładzie:

```
public int plus(int a, int b){
    return a + b;
}
```

Metoda składa się z:

1) Modyfikatora dostępu: `public, private, protected` czy domyślny
2) Opcjonalnych modyfikatorów: `final, static, abstract`
3) Typu zwracanego – tutaj `int`, lecz może być to typ prymitywny, wyliczeniowy, referencyjny lub `void` gdy metoda nie zwraca wyniku
4) Nazwy metody: `plus`, w przypadku wielu wyrazów pamiętamy o cammelCase
5) Parametrów metody: `int a, int b` umieszczonymi za nazwą metody w nawiasach `()`
6) Ciała metody pomiędzy `{` i `}`
7) Słowa kluczowego `return` z wynikiem zwracanym (tu jest to wynik wyrażenia `a + b`)

Parametry metody

Jak widać na poprzednim slajdzie i w znanej nam metodzie main, metody mogą przyjmować parametry. Mogą ale nie muszą, dla przykładu można przytoczyć metody, które wywoływaliśmy na obiekcie klasy Scanner:

```
s.nextFloat();
s.next();
```

Do ich wywołania nie były potrzebne parametry. W przypadku jednak main:

```
public static void main(String... args){
}
```

Przyjmowaliśmy argumenty w postaci tablicy. Tak i w przypadku z poprzedniego slajdu:

```
public int plus(int a, int b)...
```

Metoda przyjmowała dwa parametry typu int. Parametrów może być zatem od 0 do wielu. Mogą też deklarować stałość (final):

```
public abstract void methodA(final String a, final int b);
```

Ciało metody

Ciało metody definiuje jej zachowanie, w szczególności jest to zwrócenie wyniku.
W ciele metody możemy jednak:

1) Przypisywać wartości zmiennym

```
public void methodA(){
    int a = 1;
}
```

2) Wykonywać operacje na zmiennych, parametrach lub polach przy pomocy operatorów

```
public void methodA(){
    int a = 1;
    a++;
}
```

Ciało metody ciąg dalszy

3) Wywoływać metody na obiekcie danej klasy ale również obiektach innych klas o ile mamy do nich referencję w polu, zmiennej lub parametrze.

```
public final static void scan(Scanner s){
    s.nextFloat();
}
```

4) Można utworzyć nowy obiekt i wywołać na nim metody

```
public final static int scan(){
    Scanner s = new Scanner(System.in);
    s.nextFloat();
}
```

5) Zwrócić wynik

```
public final static int plus(int a, int b){
    return a + b;
}
```

Wywołanie metody

Wywołanie metody wykonywaliśmy już wielokrotnie. Utrwalimy sobie na czym ono polega. Załóżmy następującą klasę:

```
public class SumCalculator {
    private static final SumCalculator instance = new SumCalculator();
    private int plus(int a, int b) {
        return a + b;
    }
    private static int staticPlus(int a, int b) {
        return instance.plus(a, b); /*wywołanie metody na referencji statycznej do obiektu klasy */
    }
    public static void main(String... args) {
        int a = Integer.valueOf(args[0]); // wywołanie metody statycznej na klasie Integer
        int b = Integer.valueOf(args[0]); // wywołanie metody statycznej na klasie Integer
        int c = staticPlus(a, b); // wywołanie metody statycznej wewnątrz klasy
        System.out.println("a + b = " + c); /* wywołanie metody na referencji statycznej
                                               obiektu PrintStream w klasie System */
    }
}
```

Wywołanie metody

Wywołanie metody wykonywaliśmy już wielokrotnie. Utrwalimy sobie na czym ono polega. Załóżmy następującą klasę:

```
public class SumCalculator {
    private static final SumCalculator instance = new SumCalculator();
    private int plus(int a, int b) {
        return a + b;
    }
    private static int staticPlus(int a, int b) {
        return instance.plus(a, b); /*wywołanie metody na referencji statycznej do obiektu klasy */
    }
    public static void main(String... args) {
        int a = Integer.valueOf(args[0]); // wywołanie metody statycznej na klasie Integer
        int b = Integer.valueOf(args[0]); // wywołanie metody statycznej na klasie Integer
        int c = staticPlus(a, b); // wywołanie metody statycznej wewnątrz klasy
        System.out.println("a + b = " + c); /* wywołanie metody na referencji statycznej
                                               obiektu PrintStream w klasie System */
    }
}
```

Konstruktory, gettery i settery

Czyli ustanawianie stanu obiektu

Konstruktor

Konstruktor to specjalny rodzaj metody, która tworzy obiekt. Klasa może definiować jeden lub więcej konstruktorów. Jeśli nie zaimplementujemy konstruktora kompilator w ramach kompilacji doda domyślny konstruktor podobny do przedstawionego poniżej:

```
public SumCalculator() {
    super();
}
```

Jeśli nie chcemy aby konstruktor inicjalizował zmienne, lub wykonywał specyficzne dla tworzenia obiektu danej klasy operacje, jest to całkowicie wystarczające.
Nawet taki prosty konstruktor wart jest opisania. W skład konstruktora wchodzą:

1) Modyfikator widoczności: `public, private, protected` czy domyślny
2) Nazwa konstruktora równa nazwie klasy tu `SumCalculator`. Stosujemy CammelCase od dużej litery
3) Nawiasy `()` pomiędzy którymi, podobnie jak inne metody, konstruktor może deklarować parametry. Domyślny konstruktor jest bezparametryczny.
4) Nawiasy klamrowe ograniczające ciało metody konstruktora `{}`
5) Wywołanie konstruktora klasy nadrzędnej `super();`

Inicjalizacja pól w konstruktorze

Jak było wspomniane wcześniej konstruktor może inicjalizować stan pól. Poniżej omówimy strukturę bardziej złożonego konstruktora:

```
public class Coordinates {
    private final int x; // pole klasy
    private final int y; // drugie pole klasy

    public Coordinates(){ // kontruktor bezparametryczny
        this(0, 0); // wywołanie konstruktora parametrycznego z wartościami domyślnymi 0, 0
    }
    // deklaracja konstruktora o parametrach x i y
    public Coordinates(int x, int y) { // pomiędzy nawiasami parametry
        this.x = x; // Przypisanie parametrów do pól klasy
        this.y = y; // jak widać zarówno parametry jak i pola klasy nazywają się tak samo. W takim przypadku
        // należy wskazać, które to pola. Robimy to przez wskazanie this.x, this.y - oznacza to odwołanie do
        // pola przynależnego do klasy
        this.method(); // wywołanie metody na this
        method(); // w tym przypadku również metoda również zostanie wywołana na tym obiekcie
    }
    public void method(){ }
}
```

Pojawił się tu nowy element składni, słowo kluczowe `this`. Oznacza ono odwołanie do bieżącego obiektu. Można skorzystać z tego odwołania w celu podstawienia pól klasy albo wywołania metody na bieżącym obiekcie. Istnieje również szczególne zastosowanie słowa kluczowego . Jest to wywołanie innego konstruktora w tej samej klasie.

Settery

Ostatni typ podstawienia pól instancyjnych to podstawienie pola w metodzie. Istnieją metody wyspecjalizowane w tym zadaniu. Nazywamy je setterami. Ciało takiej metody składa się jedynie z podstawienia wartości parametru metody do pola. Składania takiej metody wygląda następująco:

```
public class DynamicCoordinates {
    private int x; // pole klasy
    private int y; // drugie pole klasy

    public void setX(int x) { // metoda ustawiająca pole x
        this.x = x;
    }

    public void setY(int y) { // metoda ustawiająca pole y
        this.y = y;
    }
}
```

Należy zwrócić uwagę na kilka istotnych elementów:

1) Setter posiada dokładnie jeden parametr, w dodatku o tym samym typie co pole klasy
2) Nazwa metody zawiera nazwę z pola poprzedzoną stałym prefiksem „set”. Zachowujemy też cammelCase.
3) Metody set nie zwracają żadnego wyniku, zatem deklarują `void`
4) W ciele metody znajduje się operacja przypisania pola, np.: `this.y = y;`

Getter

Dobrym zwyczajem się jest używanie hermetyzacji. Stosuje się praktykę tworzenia pól jako prywatne i udostępnianie ich podstawienie przez wcześniej opisane settery, o dostęp do nich przy użyciu getterów. Oczywiście klasa nie musi pozwalać zarówno na dostęp do wszystkich zmiennych jak i na ich podstawienie. Nie każdy getter oznacz towarzyszący setter i w drugą stronę. Dla zobrazowania rozważmy klasę z jednym konstruktorem i getterami:

```
public class FinalCoordinates {
    private final int x; // pole finalne instancji
    private final int y; // drugie pole finalne instancji klasy

    public FinalCoordinates(int x, int y) { // konstruktor parametryczny
        this.x = x; // podstawienie x
        this.y = y; // podstawienie y
    }

    public int getX() { // metoda dostępu do x
        return x;
    }

    public int getY() { // metoda dostępu do y
        return y;
    }
}
```

Klasa ta pokazuje, że pola finalne są podstawiane tylko raz przy utworzeniu obiektu. Możliwy jest jedynie dostęp do ich wartości poprzez metody get. Metody te charakteryzuje:

1) Nazwa składająca się z prefiksu „get” i nazwy zmiennej. Pamiętamy o cammelCase.
2) Ciało zwracające pole przy pomocy `return`, oraz typ zwracany odpowiadający typowi pola
3) Pusta lista parametrów metody

Podsumowanie

Składowe klasy

```
public class FinalCoordinates {
    /* POLA */
    // statyczne
    private static final int DEFAULT;
    private static final FinalCoordinates DEFAULT_COORDINATES = new FinalCoordinates();
    // instancyjne
    private final int x;
    private final int y;
    // bloki inicjalizacji statycznej lub instancyjnej
    static {
        DEFAULT = 0;
    }
    /* Konstruktory */
    public FinalCoordinates() {
        this(DEFAULT, DEFAULT);
    }
    public FinalCoordinates(int x, int y) {
        this.x = x;
        this.y = y;
    }
    /* Metody */
    // instancyjne
    public int getX() {
        return x;
    }
    public int getY() {
        return y;
    }
    // statyczne
    public static FinalCoordinates defaultCoorinates(){
        return new FinalCoordinates();
    }
}
```

Wywołanie metod

```
package pl.tlis.classes.fields;

public class Methods extends ClassWithFields{
    public void methodA(){
        Coordinates coordinates = new Coordinates();
        coordinates.method(); // wywołanie metody na referencji obiektu klasy
    }
    public void methodB() {
        this.methodA(); // wywołani metody na bieżącym obiekcie
        methodB(); // druga opcja wywołania metody na obiekcie
    }
    public void methodC() {
        String a = String.valueOf(1); // wowołanie metody statycznej
        methodD(a); // wywołanie metody statycznej wewnątrz klasy nie wymaga podania nazwy klasy
    }
    public static void methodD(String a){
        System.out.println(a);
    }
}
```


Dziękuję!