

# ZASADY PROGRAMOWANIA ZORIENTOWANEGO OBIEKTOWO

- Klasy i obiekty (powtórka)
- 4 podstawowe zasady programowania zorientowanego obiektowo
- Dobre praktyki

# PROGRAMOWANIE ZORIENTOWANE OBIEKTOWO (OOP, z ang. Object-Oriented Programming)

- Obiekty posiadają **typ**, **stan** (pola) i **zachowania** (metody)
- Obiekty łączą się w **klasy**
- Działanie programu jest opisywane przez **interakcje pomiędzy obiektami**



## KLASA

Szablon pozwalający definiować własne,  
bardziej złożone typy danych




## OBIEKT

Konkretny "egzemplarz"  
(instancja) klasy

# TWORZENIE KLAS I OBIEKTÓW

```
public class MyClass {  
  
    String field1;  
    String field2;  
    int field3;  
    MyClass2 field4;  
  
    public MyClass(String field1, String field2,  
                    int field3, MyClass2 field4) {  
        this.field1 = field1;  
        this.field2 = field2;  
        this.field3 = field3;  
        this.field4 = field4;  
    }  
  
    public void myMethod1(int val) {  
        this.field3 += val;  
    }  
  
    public String displayInfo() {  
        String name = this.field1 + ", " + this.field2;  
        return "Name: " + name + "(" + this.field3 + ")";  
    }  
}
```

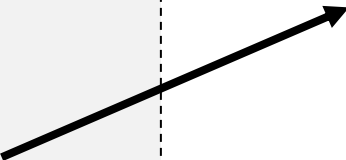


## POLA

- mają typ i nazwę
- mogą być typu wbudowanego lub własnego
- domyślnie inicjalizowane na:
  - wartości numeryczne: 0
  - boolean: false
  - char: '\u0000'
  - inne (np. String): null

# TWORZENIE KLAS I OBIEKTÓW

```
public class MyClass {  
  
    String field1;  
    String field2;  
    int field3;  
    MyClass2 field4;  
  
    public MyClass(String field1, String field2,  
                    int field3, MyClass2 field4) {  
        this.field1 = field1;  
        this.field2 = field2;  
        this.field3 = field3;  
        this.field4 = field4;  
    }  
  
    public void myMethod1(int val) {  
        this.field3 += val;  
    }  
  
    public String displayInfo() {  
        String name = this.field1 + ", " + this.field2;  
        return "Name: " + name + "(" + this.field3 + ")";  
    }  
}
```



## KONSTRUKTOR

- wywoływany słowem kluczowym *new*
- tworzy obiekt i inicjalizuje jego pola
- przypomina konstrukcją metodę, ale nie ma typu zwracanego ani instrukcji *return*
- może, ale nie musi przyjmować parametrów
- może być przeciążony

# TWORZENIE KLAS I OBIEKTÓW

```
public class MyClass {  
  
    String field1;  
    String field2;  
    int field3;  
    MyClass2 field4;  
  
    public MyClass(String field1, String field2,  
                   int field3, MyClass2 field4) {  
        this.field1 = field1;  
        this.field2 = field2;  
        this.field3 = field3;  
        this.field4 = field4;  
    }  
  
    public void myMethod1(int val) {  
        this.field3 += val;  
    }  
  
    public String displayInfo() {  
        String name = this.field1 + ", " + this.field2;  
        return "Name: " + name + "(" + this.field3 + ")";  
    }  
}
```


- Jeżeli nie został zdefiniowany jakikolwiek własny konstruktor, występuje konstruktor domyślny:

```
public MyClass() {  
}
```

- Konstruktor domyślny inicjalizuje pola wartościami domyślnymi

# TWORZENIE KLAS I OBIEKTÓW

```
public class MyClass {  
  
    String field1;  
    String field2;  
    int field3;  
    MyClass2 field4;  
  
    public MyClass(String field1, String field2,  
                    int field3, MyClass2 field4) {  
        this.field1 = field1;  
        this.field2 = field2;  
        this.field3 = field3;  
        this.field4 = field4;  
    }  
  
    public void myMethod1(int val) {  
        this.field3 += val;  
    }  
  
    public String displayInfo() {  
        String name = this.field1 + ", " + this.field2;  
        return "Name: " + name + "(" + this.field3 + ")";  
    }  
}
```

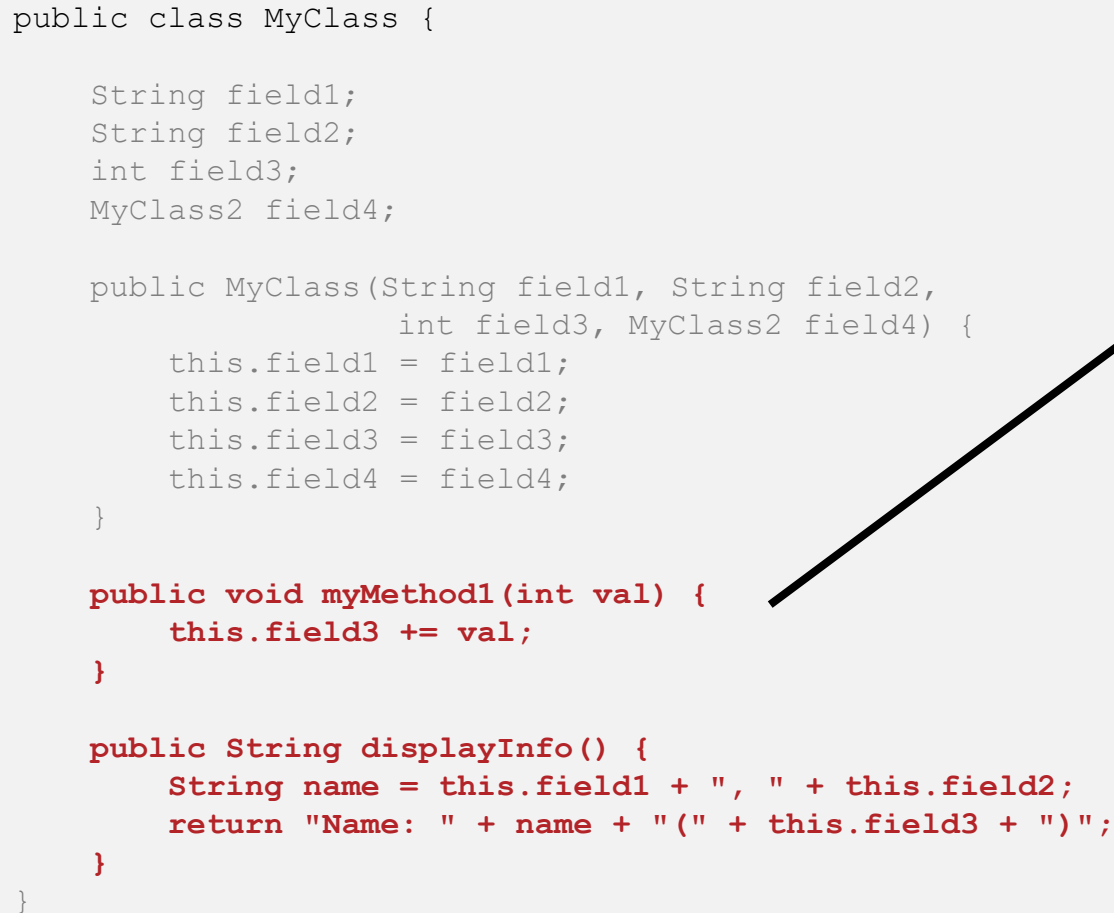


## SŁOWO KLUCZOWE THIS

- jest referencją obiektu na samego siebie
- pozwala rozróżnić pola klasy od parametrów metod i konstruktorów o pokrywających się nazwach

# TWORZENIE KLAS I OBIEKTÓW

```
public class MyClass {  
  
    String field1;  
    String field2;  
    int field3;  
    MyClass2 field4;  
  
    public MyClass(String field1, String field2,  
                    int field3, MyClass2 field4) {  
        this.field1 = field1;  
        this.field2 = field2;  
        this.field3 = field3;  
        this.field4 = field4;  
    }  
  
    public void myMethod1(int val) {  
        this.field3 += val;  
    }  
  
    public String displayInfo() {  
        String name = this.field1 + ", " + this.field2;  
        return "Name: " + name + "(" + this.field3 + ")";  
    }  
}
```



## METODY

- mogą, ale nie muszą przyjmować parametrów
- mogą, ale nie muszą zwracać wyników
- mogą operować na wartościach przechowywanych w polach klasy
- mogą być wywoływane wewnątrz tej samej klasy

# TWORZENIE KLAS I OBIEKTÓW

```
public class Person {  
  
    String name;  
    int age;  
  
    public Person() {}  
  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
}
```

```
Person p1 = new Person();
```

- name = null
- age = 0

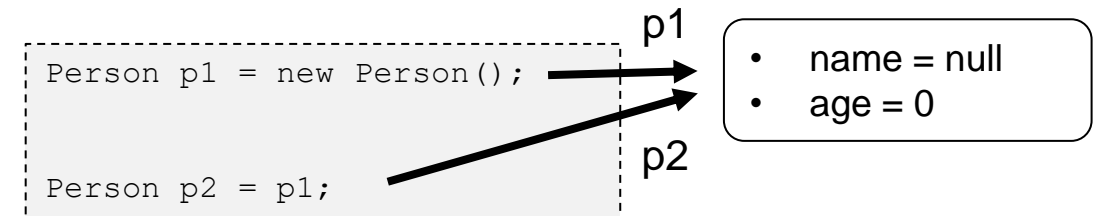
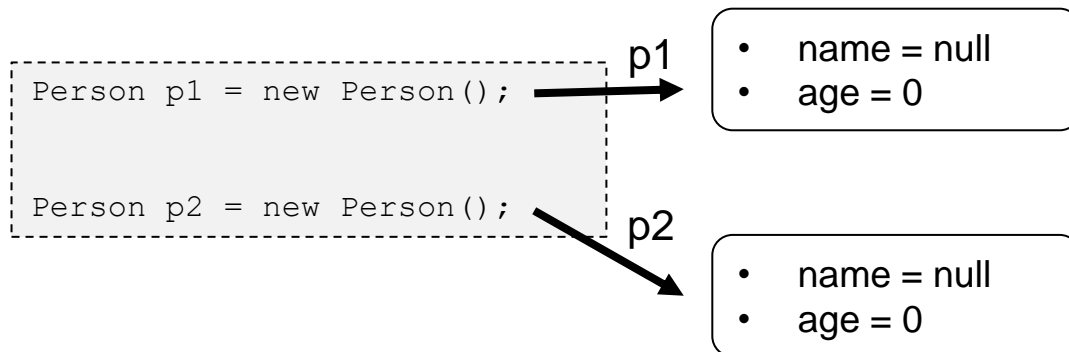
```
Person p2 = new Person("XXX", 20);
```

- name = "XXX"
- age = 20



# REFERENCJE I NULL POINTER EXCEPTION

- referencja wskazuje na obiekt w pamięci programu
- konstruktor (słowo kluczowe *new*) tworzy nowy obiekt danej klasy



# REFERENCJE I NULL POINTER EXCEPTION

```
Person p1 = new Person();  
p1.name = "XXX";  
  
System.out.println("P1 name: " + p1.name);  
  
Person p2 = new Person();  
p2.name = "YYY";  
  
System.out.println("P1 name: " + p1.name);  
System.out.println("P2 name: " + p2.name);
```



```
P1 name: XXX  
P1 name: XXX  
P2 name: YYY
```

```
Person p1 = new Person();  
p1.name = "XXX";  
  
System.out.println("P1 name: " + p1.name);  
  
Person p2 = p1;  
p2.name = "YYY";  
  
System.out.println("P1 name: " + p1.name);  
System.out.println("P2 name: " + p2.name);
```



```
P1 name: XXX  
P1 name: YYY  
P2 name: YYY
```

# REFERENCJE I NULL POINTER EXCEPTION

- NullPointerException pojawia się podczas próby odwołania do pola lub metody obiektu przez zainicjowaną zmienną

```
Person p1 = null;  
System.out.println("P1 name: " + p1.name);
```

```
Exception in thread "main" java.lang.NullPointerException Create breakpoint : Cannot read field "name" because "p1" is null  
at lecture.app.Ex1.main(Ex1.java:7)
```

```
Person[] people = new Person[10];  
people[0].name = "XXX";
```

```
Exception in thread "main" java.lang.NullPointerException Create breakpoint : Cannot assign field "name" because "people[0]" is null  
at lecture.app.Ex1.main(Ex1.java:7)
```

# DZIEDZICZENIE, ang. Inheritance

- relacja typu “jest” (ang. IS-A)
- wyrażana przez słowo kluczowe *extends*
- wykorzystywana w sytuacji, gdy nowa klasa jest szczególnym rodzajem istniejącej klasy

# DZIEDZICZENIE, ang. Inheritance

## Klasa bazowa (nadrzędna)

```
class Animal {  
    private String name;  
    private int age;  
  
    public Animal(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public int getAge() {  
        return age;  
    }  
  
    public void eat() {  
        System.out.println(name + " is eating.");  
    }  
}
```

## Klasy dziedziczące (pochodne)

```
class Dog extends Animal {  
    public Dog(String name, int age) {  
        super(name, age);  
    }  
  
    public void bark() {  
        System.out.println(getName() + " is barking.");  
    }  
}
```

```
class Cat extends Animal {  
    public Cat(String name, int age) {  
        super(name, age);  
    }  
  
    public void meow() {  
        System.out.println(getName() + " is meowing.");  
    }  
}
```

# AGREGACJA, ang. Aggregation

- relacja typu “posiada” (ang. HAS-A)
- wykorzystywana w sytuacji, gdy dany obiekt składa się z wielu obiektów składowych
- obiekty składowe mogą istnieć niezależnie oraz należeć do wielu całości

# AGREGACJA, ang. Aggregation

```
class Department {  
    private String name;  
  
    public Department(String name) {  
        this.name = name;  
    }  
  
    // Department-specific methods...  
}
```

```
class University {  
    private String name;  
    private List<Department> departments; // Aggregation  
  
    public University(String name) {  
        this.name = name;  
        this.departments = new ArrayList<>();  
    }  
  
    public void addDepartment(Department department) {  
        departments.add(department);  
    }  
  
    // University-specific methods...  
}
```

# AGREGACJA, ang. Aggregation

```
class Address {  
    private String street;  
    private String city;  
    private String postalCode;  
  
    public Address(String street,  
        String city, String postalCode) {  
        this.street = street;  
        this.city = city;  
        this.postalCode = postalCode;  
    }  
  
    // getters and setters  
}
```

```
class Person {  
    private String name;  
    private Address address;  
  
    public Person(String name, Address address) {  
        this.name = name;  
        this.address = address;  
    }  
  
    // getters and setters  
}
```

## Tworzenie obiektu klasy Person:

```
Address empAddress = new Address("123 Main St",  
    "Springfield", "12345");  
Person employee1 = new Person("John Doe", empAddress);
```



# KOMPOZYCJA, ang. Composition

- relacja typu “jest częścią” (ang. PART-OF)
- szczególny przypadek agregacji
- obiekty składowe nie mogą istnieć bez obiektu głównego ani należeć do wielu całości
- usunięcie obiektu głównego powoduje usunięcie obiektów składowych

# KOMPOZYCJA, ang. Composition

```
class Department {  
    private String name;  
  
    public Department(String name) {  
        this.name = name;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    // Department-specific methods...  
}
```

```
class University {  
    private String name;  
    private Department scienceDepartment; // Composition  
    private Department mathsDepartment; // Composition  
  
    public University(String name) {  
        this.name = name;  
        this.scienceDepartment = new Department("Science");  
        this.mathsDepartment = new Department("Maths");  
    }  
  
    // University-specific methods...  
  
    public void displayDepartments() {  
        System.out.println("University: " + name);  
        System.out.println("Departments:");  
        System.out.println("1. " + scienceDepartment.getName());  
        System.out.println("2. " + mathsDepartment.getName());  
    }  
}
```

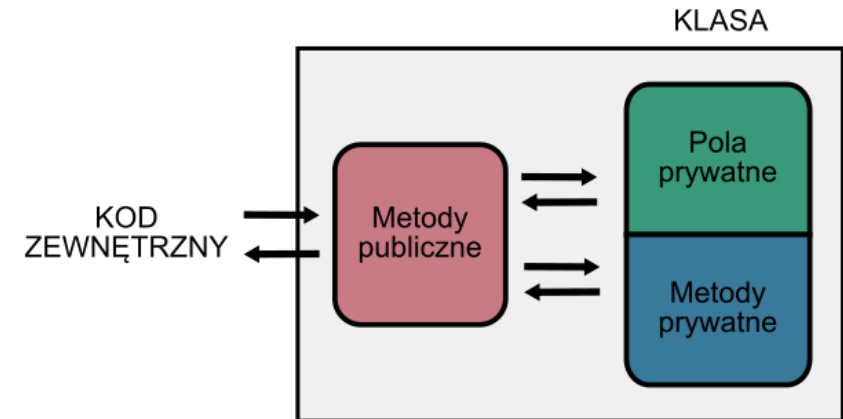
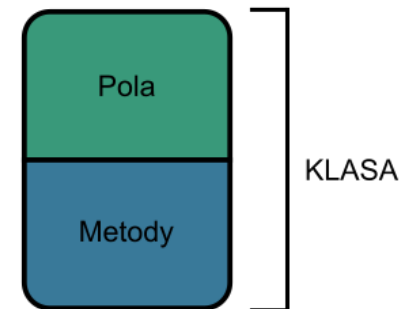
# ENKAPSULACJA (ang. Encapsulation)

Także: hermetyzacja

*Grupowanie danych i metod operujących na tych danych  
ale także:*

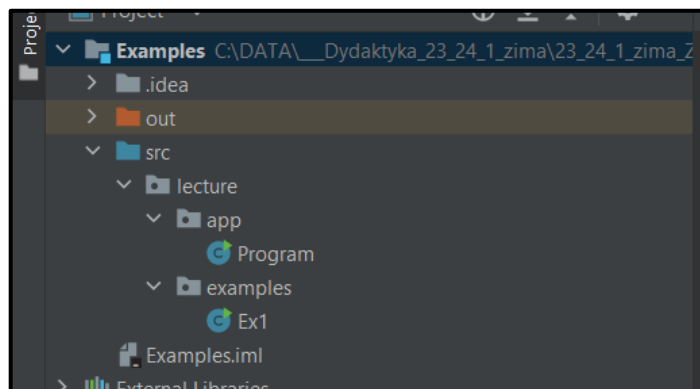
*Ograniczanie dostępu do wewnętrznych elementów klasy*

- klasy, pola, metody
- modyfikatory dostępu (ang. access modifiers)
- paczki (ang. packages)
- moduły (ang. modules)



# PACZKI (ang. Packages)

- służą do grupowania powiązanych elementów kodu (np. klas)
- ułatwiają kontrolę dostępu do elementów kodu
- zapobiegają konfliktom nazw



## Przypisywanie do paczki

```
package nazwa.naszego.pakietu;
```

```
package lecture.examples;
```

## Odwołanie do elementu paczki

```
import nazwa.naszego.pakietu.NazwaKlasy;
```

```
import lecture.examples;
```

```
import lecture.examples.Ex1;
```

# MODYFIKATORY DOSTĘPU (ang. Access modifiers)

- określają, kto (co) może korzystać z danego elementu kodu
- służą do ograniczania dostępu do elementów kodu

**Zwykle należy używać najbardziej restrykcyjnego modyfikatora, jaki jest w danej sytuacji możliwy**

| Modyfikator    |                | DEFAULT | PRIVATE | PROTECTED | PUBLIC |
|----------------|----------------|---------|---------|-----------|--------|
| TA SAMA PACZKA | TA SAMA KLASA  | tak     | tak     | tak       | tak    |
|                | KLASA POCHODNA | tak     | nie     | tak       | tak    |
|                | INNA KLASA     | tak     | nie     | tak       | tak    |
| INNA PACZKA    | KLASA POCHODNA | nie     | nie     | tak       | tak    |
|                | INNA KLASA     | nie     | nie     | nie       | tak    |

# MODYFIKATORY DOSTĘPU (ang. Access modifiers)

```
public class Person {  
    private String name;  
    private int age;  
  
    public Person(String name, int age) {  
        this.name = name;  
        this.age = age;  
    }  
}
```

**Podstawowy przypadek: prywatne pola,  
publiczne metody zwracające (getter)  
i ustawiające (setter)**

ciąg dalszy:

```
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public int getAge() {  
        return age;  
    }  
  
    public void setAge(int age) {  
        if (age >= 0) {  
            this.age = age;  
        } else {  
            System.out.println("Age cannot be negative.");  
        }  
    }  
}
```

## **ABSTRAKCJA (ang. Abstraction)**

*Ukrywanie złożonych szczegółów implementacyjnych  
za uproszczonymi interfejsami, zawierającymi tylko  
niezbędne elementy*

- enkapsulacja
- klasy abstrakcyjne (ang. abstract classes)
- interfejsy (ang. interfaces)

## **DZIEDZICZENIE (ang. Inheritance)**

*Przekazywanie cech (pól, metod) klas nadrzędnych  
do klas podrzędnych*

# POLIMORFIZM (ang. Polymorphism)

*Przyjmowanie przez klasy lub ich elementy (pola, metody) różnych form*

- dziedziczenie i interfejsy
- przeciążanie metod (ang. method overloading)
- przesłanianie metod (ang. method overriding)



# POLIMORFIZM (ang. Polymorphism)

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

ciąg dalszy:

```
public static void main(String[] args) {  
    Calculator calculator = new Calculator();  
  
    int sum1 = calculator.add(5, 10);  
    int sum2 = calculator.add(5, 10, 15);  
    double sum3 = calculator.add(2.5, 3.5);  
    String concatenatedString  
        = calculator.add("Hello, ", "World!");  
}  
}
```

**tw. polimorfizm statyczny**  
**(czasu kompilacji, ang. compile-time polymorphism)**

# POLIMORFIZM (ang. Polymorphism)

```
class Animal {
    void makeSound() {
        System.out.println("Some generic sound");
    }
}

class Dog extends Animal {
    @Override
    void makeSound() {
        System.out.println("Hau! Hau!");
    }
}

class Cat extends Animal {
    @Override
    void makeSound() {
        System.out.println("Miau!");
    }
}
```

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

        Animal[] animals = new Animal[3];
        animals[0] = new Dog();
        animals[1] = new Cat();
        animals[2] = new Animal(); // Generic animal

        for (Animal animal : animals) {
            animal.makeSound();
        }
    }
}
```

**tzw. polimorfizm dynamiczny  
(czasu wykonania, ang. runtime polymorphism)**

# ZASADA DRY, z ang. Don't Repeat Yourself

*Każdy wycinek wiedzy musi mieć dokładnie jedną, jednoznaczną i oficjalną reprezentację w ramach systemu*  
(Hunt A., Thomas, D., *Pragmatyczny programista*. Helion, 2011)

- duplikowanie dotyczy nie tylko fragmentów kodu, ale też logiki, komentarzy czy danych
- rozwiązaniem jest np. wydzielanie funkcji, abstrakcja, normalizacja relacyjnych baz danych

Odwrotność: WET, z ang. We Enjoy Typing/Waste Everyone's Time

# ZASADA DRY, z ang. Don't Repeat Yourself

```
public class AreaCalculator {  
    public static void main(String[] args) {  
        int a1 = 3;  
        int b1 = 4;  
        int area1 = a1 * b1;  
        System.out.println("Pole wynosi: " + area1);  
  
        int a2 = 2;  
        int b2 = 10;  
        int area2 = a2 * b2;  
        System.out.println("Pole wynosi: " + area2);  
    }  
}
```

VS

```
public class AreaCalculator {  
    public static void main(String[] args) {  
        int a1 = 3;  
        int b1 = 4;  
        System.out.println("Pole wynosi: "  
            + calculateArea(a1, b1));  
  
        int a2 = 2;  
        int b2 = 10;  
        System.out.println("Pole wynosi: "  
            + calculateArea(a2, b2));  
    }  
  
    public static int calculateArea(int a, int b) {  
        return a * b;  
    }  
}
```

# **ZASADA KISS, z ang. Keep It Simple, Stupid**

# **ZASADA YAGNI, z ang. You Aren't Gonna Need It**

*Zawsze implementuj rzeczy, kiedy naprawdę  
ich potrzebujesz, a nie wtedy, kiedy przewidujesz,  
że będziesz ich potrzebował.*

*(Jeffries R., You're NOT gonna need it!<sup>1</sup>)*

*Proste jest lepsze niż złożone.*

*Złożone jest lepsze niż skomplikowane.*

*(PEP20 – The Zen of Python<sup>1</sup>)*

<sup>1</sup> <https://ronjeffries.com/xprog/articles/practices/pracnotneed>

<sup>2</sup> <https://peps.python.org/pep-0020/>

# ZASADA KISS, z ang. Keep It Simple, Stupid

# ZASADA YAGNI, z ang. You Aren't Gonna Need It

Zadanie:

Zaimplementuj kalkulator dodający  
do siebie dwie liczby całkowite

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

VS

```
import java.util.ArrayList;  
  
public class OvercomplicatedCalculator {  
    private ArrayList<Integer> numbers  
        = new ArrayList<>();  
  
    public void addNumber(int number) {  
        numbers.add(number);  
    }  
  
    public int sum() {  
        int result = 0;  
        for (int number : numbers) {  
            result += number;  
        }  
        return result;  
    }  
}
```

# ZASADY SOLID

Martin R.C., *Design Principles and Design Patterns*<sup>1</sup>

|          |   |
|----------|---|
| <b>S</b> | <b>Single responsibility principle</b><br>Zasada jednej odpowiedzialności |
| <b>O</b> | <b>Open/closed principle</b><br>Zasada otwarte/zamknięte                  |
| <b>L</b> | <b>Liskov substitution principle</b><br>Zasada podstawienia Liskov        |
| <b>I</b> | <b>Interface segregation principle</b><br>Zasada segregacji interfejsów   |
| <b>D</b> | <b>Dependency inversion principle</b><br>Zasada odwrócenia zależności     |

<sup>1</sup> [https://web.archive.org/web/20150906155800/http://www.objectmentor.com/resources/articles/Principles\\_and\\_Patterns.pdf](https://web.archive.org/web/20150906155800/http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf)

# ZASADA JEDNEJ ODPOWIEDZIALNOŚCI

*Klasa powinna mieć tylko jeden powód do zmiany (jedną odpowiedzialność)*

```
public class User {  
  
    public String name;  
    public String email;  
  
    public User(String name, String email) {  
        this.name = name;  
        if (this.validateEmail(email)) {  
            this.email = email;  
        } else { throw new IllegalArgumentException("..."); }  
    }  
  
    public boolean validateEmail(String email) {  
        // email validation  
    }  
  
    // user class-specific implementation  
}
```



# ZASADA JEDNEJ ODPOWIEDZIALNOŚCI

*Klasa powinna mieć tylko jeden powód do zmiany (jedną odpowiedzialność)*

```
public class User {  
    public String name;  
    public Email email;  
  
    public User(String name, Email email) {  
        this.name = name;  
        this.email = email;  
    }  
  
    // user class-specific implementation  
}
```

```
public class Email {  
    public String email;  
  
    public Email(String email) {  
        if (this.validateEmail(email)) {  
            this.email = email;  
        } else {  
            throw new IllegalArgumentException("...");  
        }  
    }  
  
    public boolean validateEmail(String email) {  
        // email validation  
    }  
}
```