

DZIEDZICZENIE I ABSTRAKCJA c.d.

- Dziedziczenie
- Elementy UML
- Klasy abstrakcyjne
- Interfejsy

Wykład częściowo oparty na materiałach A. Jaskot

ABSTRAKCJA (ang. Abstraction)

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

Abstrakcja i enkapsulacja nie są równoznaczne:

- enkapsulacja dotyczy grupowania danych (pól) i zachowań (metod) w klasy
- abstrakcja dotyczy ukrywania szczegółów implementacyjnych

start()
accelerate()
turn()



start()
accelerate()
turn()



KLASY ABSTRAKCYJNE (ang. Abstract classes)

```
abstract class Shape {  
    protected int x, y;  
  
    public void move(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public abstract double calculateArea();  
}
```

```
public class Main {  
    public static void main(String[] args) {  
  
        Shape s = new Shape();  
  
    }  
}
```

KLASA ABSTRAKCYJNA

- oznaczana słowem kluczowym *abstract*
- nie można jej użyć do stworzenia obiektu, ale można po niej dziedziczyć

Class 'Anonymous class derived from Shape' must either be declared abstract or implement abstract method 'calculateArea()' in 'Shape'

[Implement methods](#) Alt+Shift+Enter [More actions...](#) Alt+Enter

KLASY ABSTRAKCYJNE (ang. Abstract classes)

KLASA ABSTRAKCYJNA

- służy przede wszystkim do tworzenia typu bazowego dla powiązanych klas dziedziczących:
 - definiuje wspólne pola
 - definiuje zestaw metod, jakie muszą posiadać klasy dziedziczące

```
abstract class Shape {  
    protected int x, y;  
  
    public void move(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public abstract double calculateArea();  
}
```

KLASY ABSTRAKCYJNE (ang. Abstract classes)

KLASA ABSTRAKCYJNA

- służy przede wszystkim do tworzenia typu bazowego dla powiązanych klas dziedziczących:
 - definiuje wspólne pola
 - definiuje zestaw metod, jakie muszą posiadać klasy dziedziczące

```
abstract class Shape {  
    int x, y;  
  
    public void move(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public abstract double calculateArea();  
}
```



METODA ABSTRAKCYJNA

- oznaczana słowem kluczowym *abstract*
- nie posiada ciała – definiuje tylko sygnaturę (nazwę, typ zwracany i parametry)
- nie może być prywatna

KLASY ABSTRAKCYJNE (ang. Abstract classes)

```
abstract class Shape {  
    protected int x, y;  
  
    public void move(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public abstract double calculateArea();  
}
```

```
class Circle extends Shape {  
    private double radius;  
  
    // no implementation for calculateArea()  
}
```

METODA ABSTRAKCYJNA

- klasy dziedziczące muszą implementować metody abstrakcyjne z klasy bazowej

Class 'Circle' must either be declared abstract or implement abstract method 'calculateArea()' in 'Shape'

[Implement methods](#) Alt+Shift+Enter [More actions...](#) Alt+Enter

KLASY ABSTRAKCYJNE (ang. Abstract classes)

```
abstract class Shape {  
    public abstract double calculateArea();  
}
```

METODA ABSTRAKCYJNA

- klasy dziedziczące muszą implementować metody abstrakcyjne z klasy bazowej

```
class Circle extends Shape {  
    private double radius;  
  
    public double calculateArea() {  
        return Math.PI * radius * radius;  
    }  
}
```

```
class Rectangle extends Shape {  
    private double a, b;  
  
    public double calculateArea() {  
        return a * b;  
    }  
}
```

KLASY ABSTRAKCYJNE (ang. Abstract classes)

```
abstract class Employee {  
    private String name;  
    private int employeeId;  
  
    public Employee(String name, int employeeId) {  
        this.name = name;  
        this.employeeId = employeeId;  
    }  
  
    public String getInfo() {  
        return name + " (" + employeeId + ")";  
    }  
  
    public abstract double calculateSalary();  
}
```

Kiedy stosować klasy abstrakcyjne?

- kiedy potrzebujemy bazowej funkcjonalności dla kilku blisko powiązanych klas, ale nie potrzebujemy tworzyć obiektów klasy bazowej

KLASY ABSTRAKCYJNE (ang. Abstract classes)

```
abstract class Employee {  
    private String name;  
    private int employeeId;  
  
    public Employee(String name, int employeeId) {  
        this.name = name;  
        this.employeeId = employeeId;  
    }  
}
```

```
public class FullTimeEmployee extends Employee {  
    private double monthlySalary;  
  
    public FullTimeEmployee  
        (String name, int employeeId, double monthlySalary) {  
        super(name, employeeId);  
        this.monthlySalary = monthlySalary;  
    }  
  
    @Override  
    public double calculateSalary() {  
        return monthlySalary;  
    }  
}
```

Kiedy stosować klasy abstrakcyjne?

- kiedy potrzebujemy bazowej funkcjonalności dla kilku blisko powiązanych klas, ale nie potrzebujemy tworzyć obiektów klasy bazowej

```
public class PartTimeEmployee extends Employee {  
    private double hourlyWage;  
    private int hoursWorked;  
  
    public PartTimeEmployee(String name,  
        int employeeId, double hourlyWage, int hoursWorked) {  
        super(name, employeeId);  
        this.hourlyWage = hourlyWage;  
        this.hoursWorked = hoursWorked;  
    }  
  
    @Override  
    public double calculateSalary() {  
        return hourlyWage * hoursWorked;  
    }  
}
```

KLASY ABSTRAKCYJNE (ang. Abstract classes)

```
abstract class Shape {  
    int x, y;  
  
    public void move(int x, int y) { // some code }  
}
```

- klasa abstrakcyjna nie musi mieć metod abstrakcyjnych

KLASY ABSTRAKCYJNE (ang. Abstract classes)

```
abstract class Shape {  
    int x, y;  
  
    public void move(int x, int y) { // some code }  
}
```

```
class Shape {  
    abstract double calculateArea();  
}
```

Class 'Shape' must either be declared abstract or implement abstract method 'calculateArea()' in 'Shape'

Make 'Shape' abstract Alt+Shift+Enter More actions... Alt+Enter

- klasa abstrakcyjna nie musi mieć metod abstrakcyjnych
- ale:
- jeżeli klasa ma chociaż jedną metodę abstrakcyjną, to musi być zadeklarowana jako klasa abstrakcyjna

KLASY ABSTRAKCYJNE (ang. Abstract classes)

```
abstract class Shape {  
    int x, y;  
  
    abstract double calculateArea();  
}
```

```
abstract class Circle extends Shape {  
    double radius;  
  
    // no implementation for calculateArea()  
}
```


- klasa dziedzicząca nie musi implementować metod abstrakcyjnych z klasy bazowej, jeżeli też jest zadeklarowana jako abstrakcyjna

KLASY ABSTRAKCYJNE (ang. Abstract classes)

```
abstract class Shape {  
    protected int x, y;  
  
    abstract double calculateArea();  
}
```

```
abstract class Color {  
    private String color;  
  
    public String getColor(); { // some code }  
}
```

```
class Circle extends Shape, Color {  
}
```

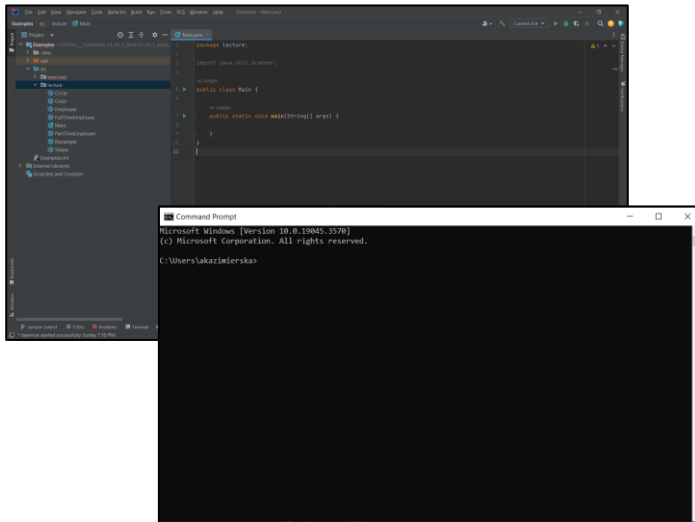


Class cannot extend multiple classes

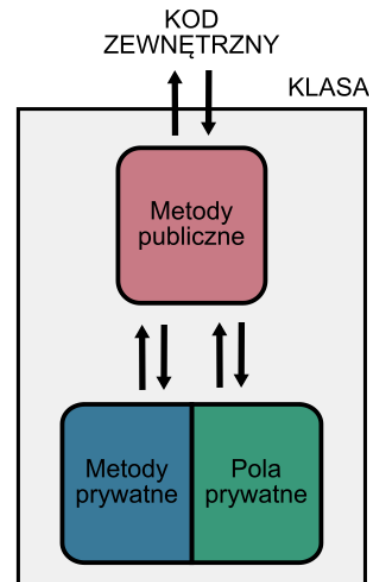
- możliwe jest dziedziczenie tylko po jednej klasie abstrakcyjnej (wciąż brak wielodziedziczenia)

SŁOWO "INTERFEJS" W PROGRAMOWANIU

INTERFEJS UŻYTKOWNIKA
np. graficzny, tekstowy



INTERFEJS KLASY



INTERFEJS
JAKO SPOSÓB NA DEFINIOWANIE
TYPU ABSTRAKCYJNEGO

```
public interface Payable {  
    double getPaymentAmount();  
}
```

INTERFEJSY (ang. Interfaces)


INTERFEJS

- oznaczany słowem kluczowym *interface*
- nie można go użyć do stworzenia obiektu, ale inne klasy mogą go *implementować*
- zawiera tylko deklaracje metod, bez konkretnej implementacji
- stanowi “kontrakt” pomiędzy klasą a innymi fragmentami kodu – “obiecuje” metody, które klasa implementująca interfejs będzie posiadać

```
interface Shape {  
    double area();  
    double perimeter();  
    void displayInfo();  
}
```

INTERFEJSY (ang. Interfaces)

```
interface Shape {  
  
    double area();  
    double perimeter();  
    void displayInfo();  
  
}
```



- metody interfejsu są publiczne i abstrakcyjne (nawet jeżeli zostaną podane bez modyfikatorów)
- klasy implementujące interfejs muszą zapewnić konkretne implementacje jego metod
- interfejs nie może zawierać konstruktora

```
class Rectangle implements Shape {  
    private double width, height;  
  
    public Rectangle(double width, double height) {  
        this.width = width;  
        this.height = height;  
    }  
  
    @Override  
    public double area() {  
        return width * height;  
    }  
  
    @Override  
    public double perimeter() {  
        return 2 * (width + height);  
    }  
  
    @Override  
    public void displayInfo() {  
        System.out.println("Rectangle: " + width + " x " + height);  
    }  
}
```


INTERFEJSY (ang. Interfaces)

```
interface Payable {  
    double calculatePayment();  
}
```

```
interface Moveable {  
    void move();  
}
```

```
interface Shape {  
}
```

```
interface List {  
}
```

Konwencja nazw

- przymiotniki kończące się na “able” lub “ible”, jeżeli interfejs zapewnia “zdolności”


w przeciwnym wypadku:

- rzeczowniki (analogicznie jak nazwy klas)

INTERFEJSY (ang. Interfaces)

```
public class FullTimeEmployee {  
    private String name;  
    private int employeeID;  
    private double monthlySalary;  
  
    public FullTimeEmployee  
    (String name, int employeeId, double monthlySalary) {  
        this.name = name;  
        this.employeeID = employeeId;  
        this.monthlySalary = monthlySalary;  
    }  
  
    public String getInfo() {  
        return "Employee: " + employeeID + ", " + name;  
    }  
  
    public double calculateSalary() {  
        return monthlySalary;  
    }  
}
```

```
public class PayrollManager {  
  
    private ArrayList<FullTimeEmployee> employees  
    = new ArrayList<>();  
  
    public void addEmployee(FullTimeEmployee employee) {  
        employees.add(employee);  
    }  
  
    public void generatePayroll() {  
        for (FullTimeEmployee employee : employees) {  
            System.out.println(employee.getInfo()  
                + " - " + employee.calculateSalary());  
        }  
    }  
}
```



```
Employee: 123, xxx - 5000.0  
Employee: 456, yyy - 3000.0  
Employee: 111, zzz - 5500.0
```

INTERFEJSY (ang. Interfaces)

```
public class PayrollManager {  
  
    private ArrayList<FullTimeEmployee> employees  
    = new ArrayList<>();  
  
    public void addEmployee(FullTimeEmployee employee) {  
        employees.add(employee);  
    }  
  
    public void generatePayroll() {  
        for (FullTimeEmployee employee : employees) {  
            System.out.println(employee.getInfo()  
                + " - " + employee.calculateSalary());  
        }  
    }  
}
```

Problem:
Potrzebujemy dodać do programu klasę
PartTimeEmployee, zachowując jedną klasę
PayrollManager

INTERFEJSY (ang. Interfaces)

```
public class PayrollManager {  
  
    private ArrayList<FullTimeEmployee> employees  
    = new ArrayList<>();  
  
    public void addEmployee(FullTimeEmployee employee) {  
        employees.add(employee);  
    }  
  
    public void generatePayroll() {  
        for (FullTimeEmployee employee : employees) {  
            System.out.println(employee.getInfo()  
                + " - " + employee.calculateSalary());  
        }  
    }  
}
```

Problem:
Potrzebujemy dodać do programu klasę
PartTimeEmployee, zachowując jedną klasę
PayrollManager

```
interface Employee {  
    String getInfo();  
    double calculateSalary();  
}
```

INTERFEJSY (ang. Interfaces)

```
public class FullTimeEmployee implements Employee {
    private String name;
    private int employeeID;
    private double monthlySalary;

    public FullTimeEmployee
    (String name, int employeeId, double monthlySalary) {
        this.name = name;
        this.employeeID = employeeId;
        this.monthlySalary = monthlySalary;
    }

    @Override
    public String getInfo() {
        return "Employee: " + employeeID + ", " + name;
    }

    @Override
    public double calculateSalary() {
        return monthlySalary;
    }
}
```

```
public class PartTimeEmployee implements Employee{
    private String name;
    private double hourlyWage;
    private int hoursWorked;

    public PartTimeEmployee
    (String name, double hourlyWage, int hoursWorked) {
        this.name = name;
        this.hourlyWage = hourlyWage;
        this.hoursWorked = hoursWorked;
    }

    @Override
    public String getInfo() {
        return "Employee: " + name + " (part-timer)";
    }

    @Override
    public double calculateSalary() {
        return hourlyWage * hoursWorked;
    }
}
```

INTERFEJSY (ang. Interfaces)


```
public class PayrollManager {  
  
    private ArrayList<Employee> employees  
    = new ArrayList<>();  
  
    public void addEmployee(Employee employee) {  
        employees.add(employee);  
    }  
  
    public void generatePayroll() {  
        for (Employee employee : employees) {  
            System.out.println(employee.getInfo()  
                + " - " + employee.calculateSalary());  
        }  
    }  
}
```

Typ interfejsu zamiast typu konkretnej klasy

INTERFEJSY (ang. Interfaces)

```
public class PayrollManager {  
  
    private ArrayList<Employee> employees  
    = new ArrayList<>();  
  
    public void addEmployee(Employee employee) {  
        employees.add(employee);  
    }  
  
    public void generatePayroll() {  
        for (Employee employee : employees) {  
            System.out.println(employee.getInfo()  
                + " - " + employee.calculateSalary());  
        }  
    }  
}
```

```
public class Main {  
    public static void main(String[] args) {  
        PayrollManager payroll = new PayrollManager();  
  
        payroll.addEmployee  
            (new FullTimeEmployee("xxx", 123, 5000));  
        payroll.addEmployee  
            (new FullTimeEmployee("yyy", 456, 3000));  
        payroll.addEmployee  
            (new PartTimeEmployee("aaa", 10, 30));  
        payroll.addEmployee  
            (new FullTimeEmployee("zzz", 111, 5500));  
        payroll.addEmployee  
            (new PartTimeEmployee("bbb", 25, 33));  
  
        payroll.generatePayroll();  
    }  
}
```

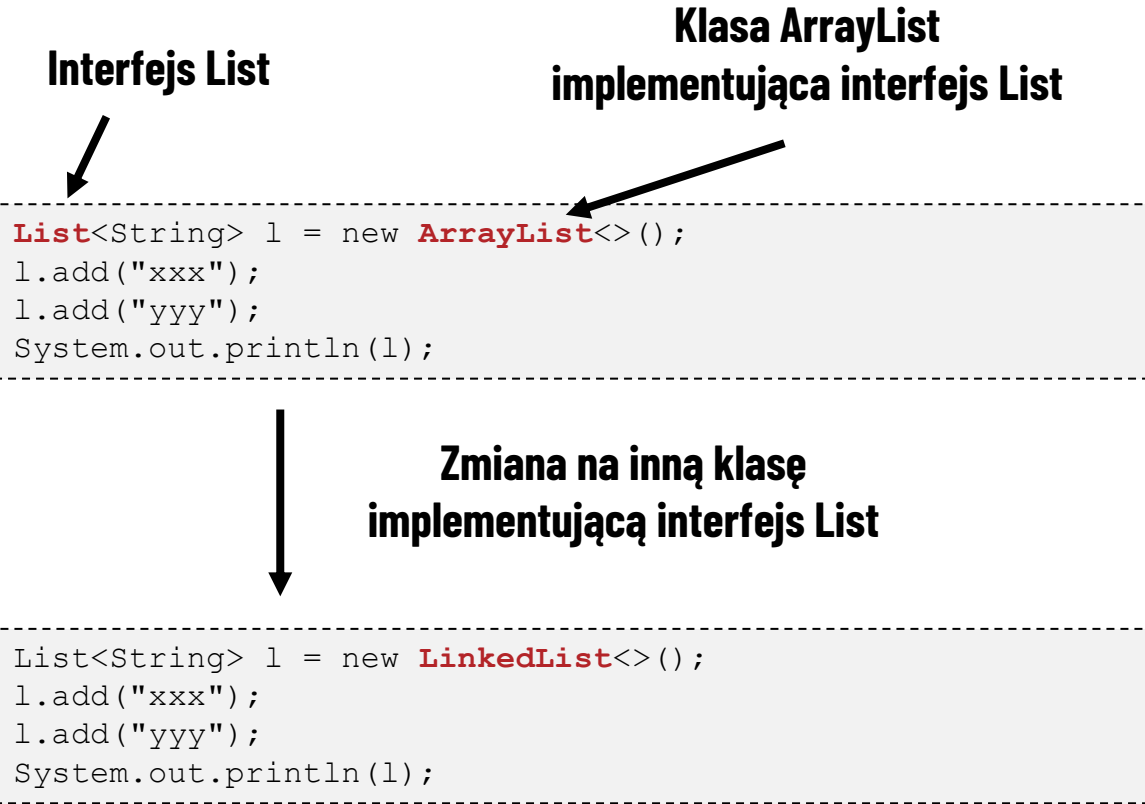


```
Employee: 123, xxx - 5000.0  
Employee: 456, yyy - 3000.0  
Employee: aaa (part-timer) - 300.0  
Employee: 111, zzz - 5500.0  
Employee: bbb (part-timer) - 825.0
```

INTERFEJSY (ang. Interfaces)

Interfejs List

Klasa ArrayList
implementująca interfejs List



```
List<String> l = new ArrayList<>();  
l.add("xxx");  
l.add("yyy");  
System.out.println(l);
```

Zmiana na inną klasę
implementującą interfejs List

```
List<String> l = new LinkedList<>();  
l.add("xxx");  
l.add("yyy");  
System.out.println(l);
```

Kiedy stosować interfejsy?

- kiedy chcemy zapewnić “elastyczność” kodu – możliwość łatwej wymiany jednej klasy na inne

INTERFEJSY (ang. Interfaces)

```
public interface Payable {  
    double getPaymentAmount();  
}
```

```
public interface Printable {  
    void print();  
}
```

```
class Invoice implements Payable, Printable {  
    // Invoice class-specific variables  
  
    @Override  
    public double getPaymentAmount() { // some code }  
  
    @Override  
    public void print() { // some code }  
}
```

Kiedy stosować interfejsy?

- kiedy potrzebujemy “dziedziczenia” po więcej niż jednym typie bazowym
- kiedy klasy implementujące interfejs mają być ze sobą luźno powiązane (bez ścisłej hierarchii dziedziczenia)

INTERFEJSY (ang. Interfaces)

```
public interface Payable {  
    double getPaymentAmount();  
}
```

Kiedy stosować interfejsy?

- kiedy potrzebujemy “dziedziczenia” po więcej niż jednym typie bazowym
- kiedy klasy implementujące interfejs mają być ze sobą luźno powiązane (bez ścisłej hierarchii dziedziczenia)

```
class Invoice implements Payable, Printable {  
    // Invoice class-specific variables  
  
    @Override  
    public double getPaymentAmount() { // some code }  
  
    @Override  
    public void print() { // some code }  
}
```

```
class Intern implements Payable {  
    // Intern class-specific variables  
  
    @Override  
    public double getPaymentAmount() { // some code }  
}
```

INTERFEJSY (ang. Interfaces)

```
interface Shape {  
    double calculateArea();  
    void draw();  
}
```

```
interface ThreeDimensionalShape extends Shape {  
    double calculateVolume();  
}
```

```
class Sphere implements ThreeDimensionalShape {  
    private double radius;  
  
    @Override  
    public double calculateArea() { // some code }  
  
    @Override  
    public double calculateVolume() { // some code }  
  
    @Override  
    public void draw() { // some code }  
}
```

- interfejs może rozszerzać inne interfejsy (dziedziczyć po interfejsach) za pomocą słowa kluczowego *extends*
- klasy implementujące “rozszerzony” interfejs muszą też implementować metody interfejsu “bazowego”

INTERFEJSY (ang. Interfaces)

```
interface Shape {  
    double PI = 3.14159265359;  
  
    double calculateArea();  
    double calculatePerimeter();  
}
```

```
class Circle implements Shape {  
    private double radius;  
  
    public Circle(double radius) {  
        this.radius = radius;  
    }  
  
    @Override  
    public double calculateArea() {  
        return PI * radius * radius; }  
  
    @Override  
    public double calculatePerimeter() {  
        return 2 * PI * radius; }  
}
```

- interfejs może posiadać pola, które domyślnie są stałymi – są publiczne, statyczne i finalne
- stałe interfejsu są dostępne dla wszystkich implementujących go klas

INTERFEJSY (ang. Interfaces)

```
interface MediaControls {  
    void play();  
    void pause();  
    void stop();  
}
```

```
class MediaPlayer1 implements MediaControls {  
  
    // implementation for play(), pause() and stop()  
  
}
```

```
class MediaPlayer2 implements MediaControls {  
  
    // implementation for play(), pause() and stop()  
  
}
```

Od Javy 8:


- interfejs może posiadać metody domyślne (słowo kluczowe *default*) z konkretną implementacją
- interfejs może posiadać metody statyczne, których nie da się przesłonić w klasach implementujących

Od Javy 9:

- interfejs może posiadać metody prywatne

INTERFEJSY (ang. Interfaces)

```
interface MediaControls {  
    void play();  
    void pause();  
    void stop();  
  
    default void setVolume(int volume) {  
        // default implementation  
    }  
}
```



```
class MediaPlayer1 implements MediaControls {  
  
    // implementation for play(), pause() and stop()  
  
}
```

```
class MediaPlayer2 implements MediaControls {  
  
    // implementation for play(), pause() and stop()  
  
}
```

Od Javy 8:

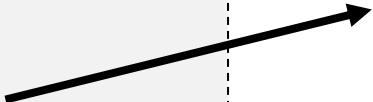
- interfejs może posiadać metody domyślne (słowo kluczowe *default*) z konkretną implementacją
- interfejs może posiadać metody statyczne, których nie da się przesłonić w klasach implementujących

Od Javy 9:

- interfejs może posiadać metody prywatne

INTERFEJSY (ang. Interfaces)

```
interface LoggerUtils {  
    void logError(String message);  
    void logWarning(String message);  
    void logInfo(String message);  
  
    static void logMessage(String message) {  
        System.out.println("Log: " + message);  
    }  
}
```



Od Javy 8:

- interfejs może posiadać metody domyślne (słowo kluczowe *default*) z konkretną implementacją
- interfejs może posiadać metody statyczne, których nie da się przesłonić w klasach implementujących

Od Javy 9:

- interfejs może posiadać metody prywatne

UML, z ang. Unified Modeling Language

- graficzny język ogólnego przeznaczenia służący do modelowania systemów
- przystosowany głównie do programowania zorientowanego obiektowo
- opisuje statyczne (diagramy struktury) oraz dynamiczne (diagramy zachowań) elementy systemu

DIAGRAMY STRUKTURY	
<ul style="list-style-type: none">• Diagram klas• Diagram obiektów• Diagram pakietów• Diagram komponentów	<ul style="list-style-type: none">• Diagram struktur połączonych• Diagram wdrożeniowy
DIAGRAMY ZACHOWAŃ	
<ul style="list-style-type: none">• Diagram aktywności• Diagram maszyny stanowej• Diagram przypadków użycia	<ul style="list-style-type: none">• Diagram komunikacji• Diagram sekwencji• Diagram przebiegów czasowych• Diagram stanów

DIAGRAM CZYNNOŚCI (ang. Activity diagram)

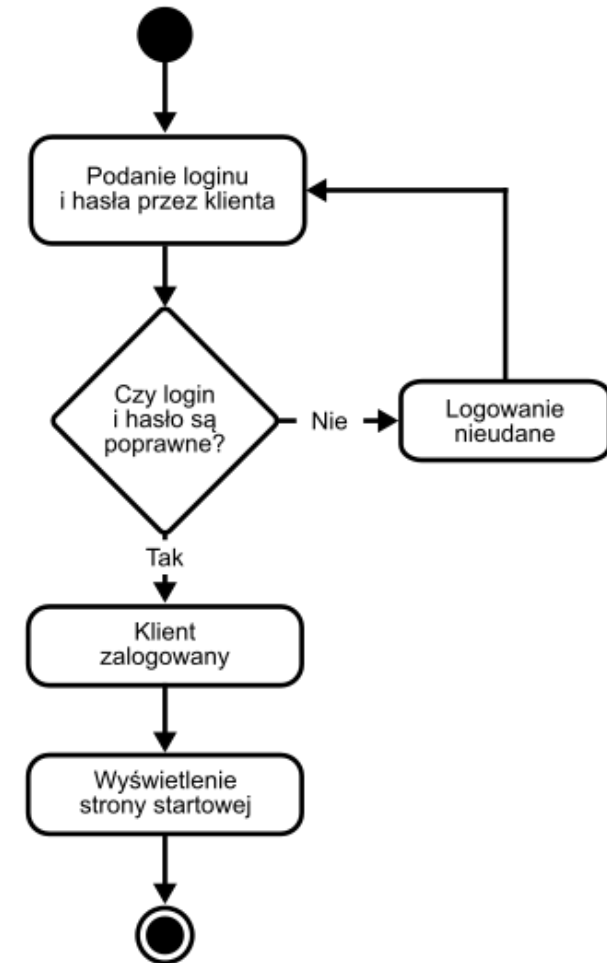
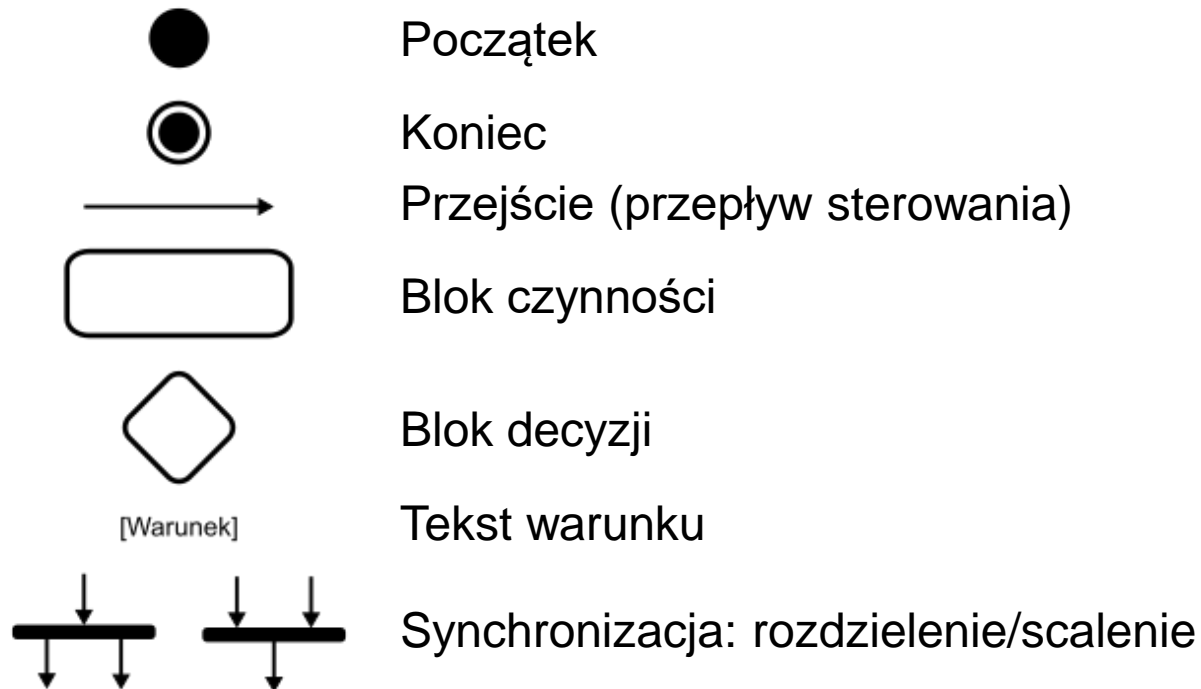
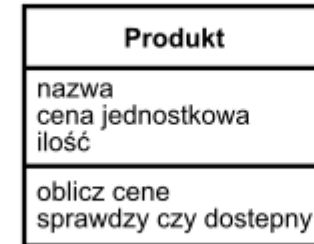
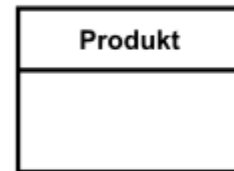
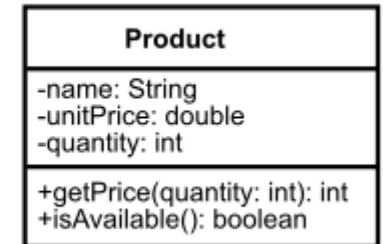


DIAGRAM KLAS (ang. Class diagram)

- reprezentuje klasy oraz zależności pomiędzy klasami
- na poziomie pojedynczej klasy opisuje:
 - nazwę klasy,
 - pola i metody klasy,
 - poziom dostępu do atrybutów
- na poziomie relacji między klasami opisuje rodzaj zależności (np. dziedziczenie, implementacja, agregacja)



Koncepcja



Implementacja

Modyfikatory dostępu:

+ public
protected
- private

DIAGRAM KLAS (ang. Class diagram)

