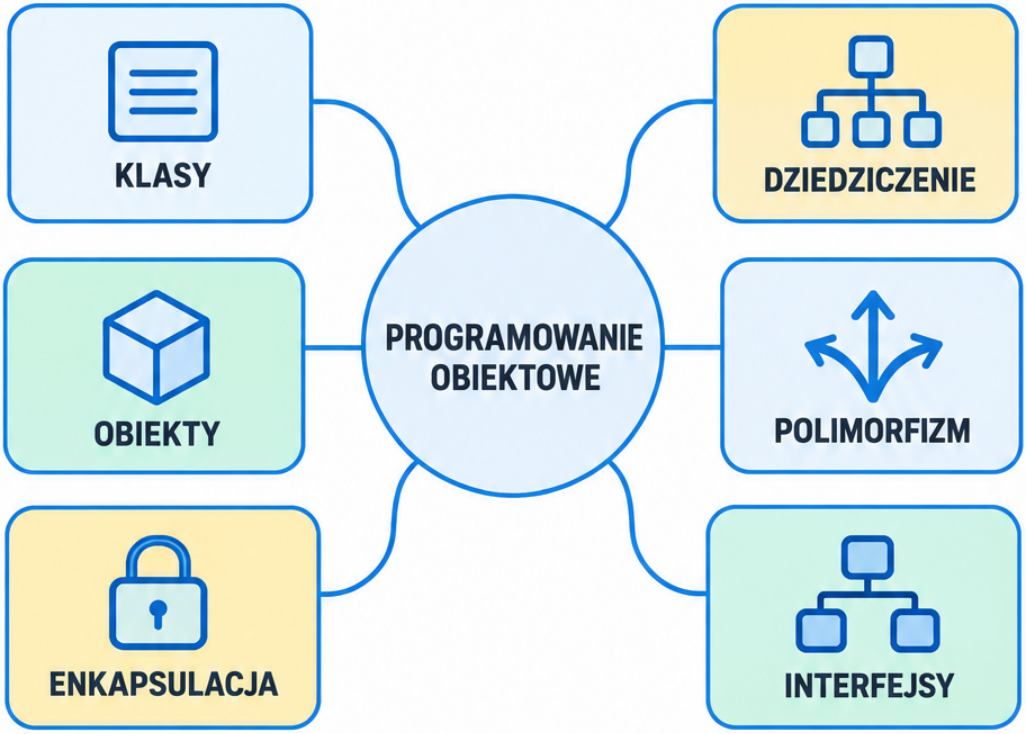
BONUS #5

WIZUALNY PRZEWODNIK PO PROGRAMOWANIU OBIEKTOWYM

Zrozum klasy, obiekty i najważniejsze zasady OOP w Javie — krok po kroku.



Informacje o materiale

Bonus #5 · Wizualny przewodnik po programowaniu obiektowym

© 2026. Wszelkie prawa zastrzeżone.

Materiał został przygotowany w celach edukacyjnych i służy do samodzielnej nauki podstaw programowania obiektowego w języku Java.

Przedstawione przykłady mają charakter dydaktyczny i pomagają zrozumieć najważniejsze mechanizmy OOP.

Jak korzystać z przewodnika?

1. **Przeczytaj wyjaśnienie.**
2. **Przeanalizuj schemat.**
3. **Sprawdź przykład kodu.**
4. **Spróbuj samodzielnie zastosować poznaną zasadę.**



SPIS TREŚCI

01 **FUNDAMENTY PROGRAMOWANIA OBIEKTOWEGO**
Strona 04
Klasy, obiekty, pola, metody i pierwsze instancje.



02 **ENKAPSULACJA I ZARZĄDZANIE STANEM**
Strona 13
Kontrola dostępu, konstruktory i bezpieczne zarządzanie danymi.



03 **DZIEDZICZENIE, POLIMORFIZM I ABSTRAKCJA**
Strona 22
Relacje między klasami, wspólne zachowania i elastyczność kodu.



04 **INTERFEJSY, KOMPOZYCJA I PROJEKTOWANIE KLAS**
Strona 32
Kontrakty, współpraca obiektów i czytelna struktura programu.



05 **PROJEKT PRAKTYCZNY I UTRWALENIE WIEDZY**
Strona 41
Praktyczny projekt i podsumowanie najważniejszych zasad.



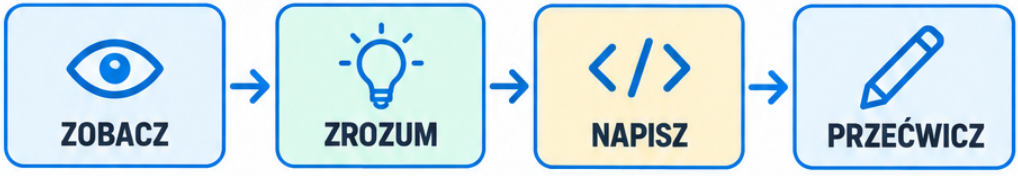
WITAJ W ŚWIECIE PROGRAMOWANIA OBIEKTOWEGO!

Zacznij myśleć o kodzie jak o świecie współpracujących obiektów.

Programowanie obiektowe, czyli OOP, to sposób organizowania kodu wokół obiektów, które przechowują dane i wykonują określone działania.

Nie musisz znać wszystkich zaawansowanych pojęć, aby zacząć. W tym przewodniku będziesz poznawać je stopniowo — od prostych klas aż po współpracujące obiekty.

JAK BĘDZIEMY SIĘ UCZYĆ?



Po ukończeniu przewodnika będziesz potrafić:

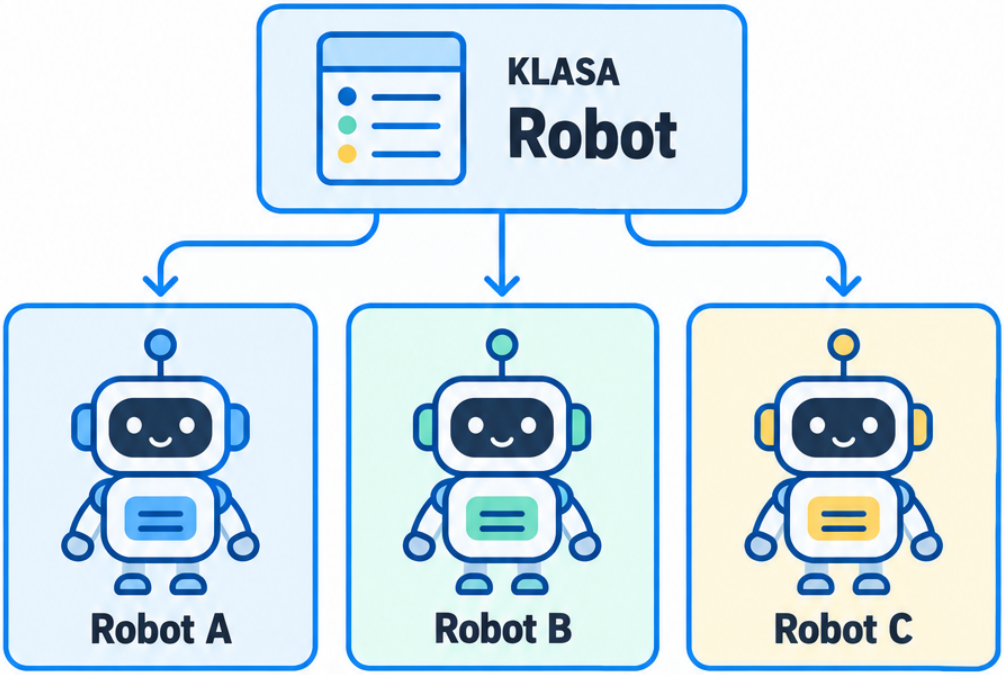
- rozpoznawać klasy i obiekty;
- rozumieć ich pola i metody;
- wyjaśniać podstawowe zasady OOP;
- tworzyć proste programy obiektowe w Javie.



MODUŁ 01

FUNDAMENTY PROGRAMOWANIA OBIEKTOWEGO

Poznaj klasy, obiekty, pola i metody — podstawowe elementy OOP.

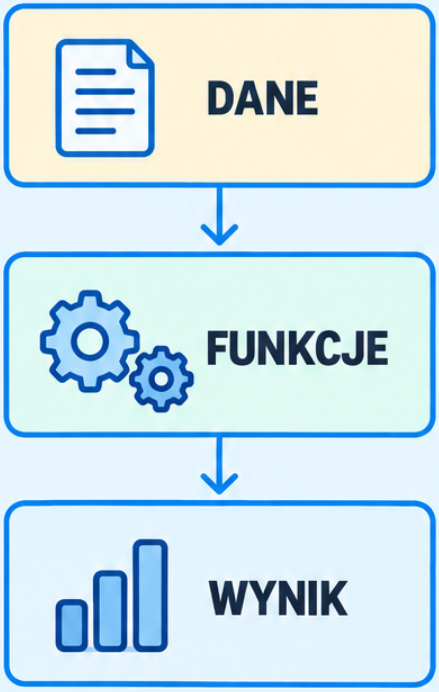


Jedna klasa → wiele obiektów

Jedna klasa może posłużyć do stworzenia wielu niezależnych obiektów.

CZYM JEST PROGRAMOWANIE OBIEKTOWE?

PODEJŚCIE PROCEDURALNE



Dane i funkcje są organizowane oddzielnie.

PODEJŚCIE OBIEKTOWE



Obiekt łączy dane i działania, które na nich operują.

OOP pomaga organizować większe programy w czytelne, współpracujące części.

KLASA A OBIEKT — POZNAJ RÓŻNICĘ

KLASA = PLAN

Klasa opisuje, jakie dane i działania będą miały obiekty.

OBIEKT = KONKRETNY EGZEMPLARZ

Obiekt to instancja klasy, utworzona podczas działania programu.

KLASA ROBOT

`nazwa · energia · przedstawSie()`



OBIEKT 1

Robo · 80 energii



OBIEKT 2

Mika · 50 energii

Przykładowe stany obiektów po przypisaniu wartości.
Te wartości są jedynie przykładem.

```
Robot robot1 = new Robot();
Robot robot2 = new Robot();
```

Jedna klasa może tworzyć wiele obiektów, a każdy obiekt może mieć własny stan.

ANATOMIA KLASY W JAVIE

```
class Robot {

   String nazwa;
   int energia;

   void przedstawSie() {
      System.out.println(nazwa);
   }
}
```

1. NAZWA KLASY

`Robot`

Określa typ tworzonych obiektów.

2. POLA

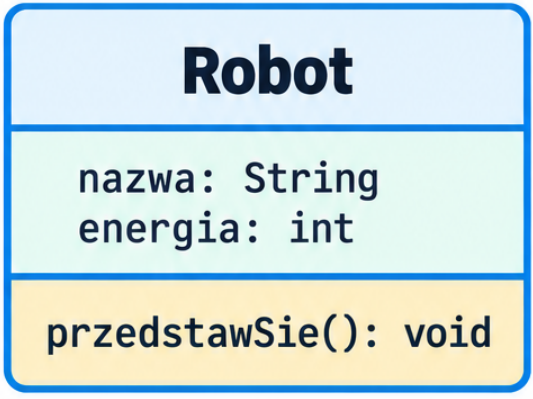
`nazwa, energia`

Przechowują dane obiektu.

3. METODA

`przedstawSie()`

Określa działanie, które obiekt może wykonać.



Klasa definiuje strukturę i zachowania obiektów.

JAK TWORZYĆ OBIEKTY ZA POMOCĄ NEW?

1. KROK 1 — ZDEFINIUJ KLASĘ

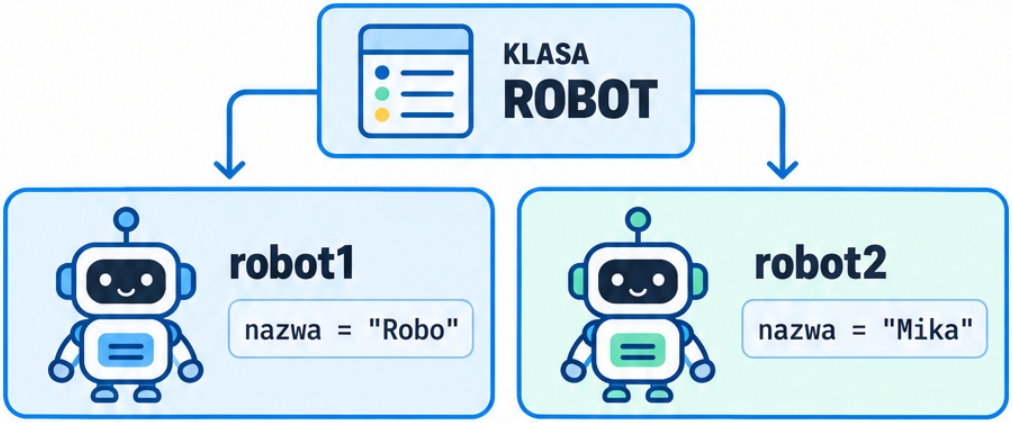
```
class Robot {
    String nazwa;
}
```

2. KROK 2 — UTWÓRZ OBIEKTY

```
Robot robot1 = new Robot();
Robot robot2 = new Robot();
```

3. KROK 3 — NADAJ IM WARTOŚCI

```
robot1.nazwa = "Robo";
robot2.nazwa = "Mika";
```



Słowo kluczowe **new** tworzy nowy obiekt. Zmienna przechowuje referencję do tego obiektu.

Dwa wywołania new Robot() tworzą dwa osobne obiekty.

POLA I METODY — CO OBIEKT WIE I CO POTRAFI?



OBIEKT
ROBOT

POLA — CO OBIEKT WIE?

nazwa — imię robota;
energia — poziom energii.

METODY — CO OBIEKT POTRAFI?

`przedstawSie()` — przedstawia swoją nazwę;
`laduj()` — może zwiększać poziom energii.

```
class Robot {
    String nazwa;
    int energia;

    void przedstawSie() {
        System.out.println(nazwa);
    }

    void laduj() {
        energia = 100;
    }
}
```



ROBOT
energia = 30

→ `robot.laduj()` → `energia = 100`

Pola opisują stan obiektu, a metody określają jego zachowanie i mogą zmieniać jego stan.

JAK WYWOŁYWAĆ METODY OBIEKTÓW?

OBIEKT → METODA → DZIAŁANIE → WYNIK

```java
class Robot {
    String nazwa;
    void przedstawSie() {
        System.out.println(
            "Cześć! Jestem " + nazwa
        );
    }
}
```

```java
Robot robot = new Robot();
robot.nazwa = "Robo";
robot.przedstawSie();
```



WYNIK:

```
Cześć! Jestem Robo
```

1. OBIEKT

`robot`

Obiekt utworzony na podstawie klasy Robot.

2. METODA

`przedstawSie()`

Metoda opisująca określone działanie.

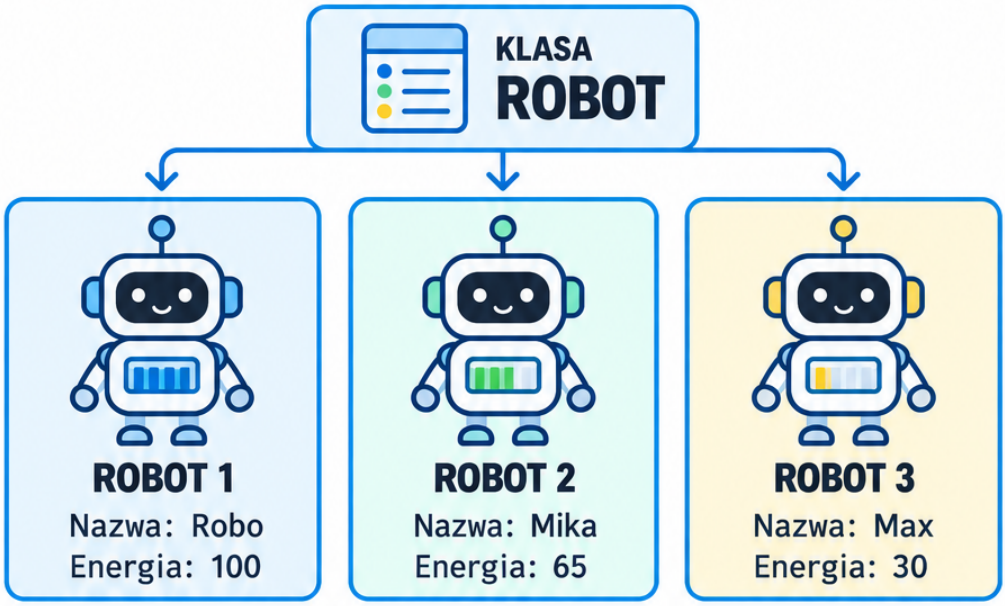
3. WYWOŁANIE

`robot.przedstawSie();`

Polecenie wykonania metody na wskazanym obiekcie.

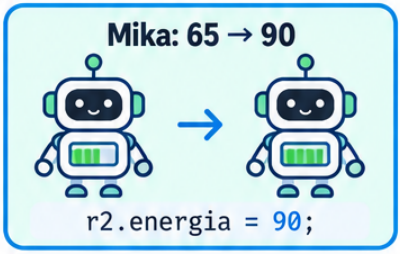
Metody pozwalają obiektom wykonywać określone działania.

JEDNA KLASA, WIELE NIEZALEŻNYCH OBIEKTÓW




```
class Robot {
    String nazwa;
    int energia;
}
Robot r1 = new Robot();
Robot r2 = new Robot();
Robot r3 = new Robot();
r1.nazwa = "Robo"; r1.energia = 100;
r2.nazwa = "Mika"; r2.energia = 65;
r3.nazwa = "Max";  r3.energia = 30;
```

Każdy obiekt ma własne pola instancji. Zmiana energii jednego robota nie zmienia energii pozostałych.



Ta sama klasa może opisywać wiele obiektów o różnych stanach.

MODUŁ 01

KLASY, OBIEKTY, POLA I METODY — WSZYSTKO RAZEM

1. KLASA
Szablon opisujący strukturę i zachowanie obiektów.

2. OBIEKT
Konkretna instancja klasy.



PODSTAWY OOP

3. POLE
Zmienna przechowująca dane obiektu.

4. METODA
Działanie zdefiniowane w klasie.

5. NEW
Słowo kluczowe używane do tworzenia nowych obiektów.

```
class Pies {
    String imie;
    void szczekaj() {
        System.out.println("Hau!");
    }
}

Pies pies = new Pies();
pies.imie = "Azor";
pies.szczekaj();
```

klasa Pies

pole imie

metoda szczekaj()



WYNIK: Hau!

MINI WYZWANIE

1. Jak nazywa się klasa w przykładzie?
2. Który element jest polem?
3. Który element jest metodą?
4. Co robi słowo **new**?

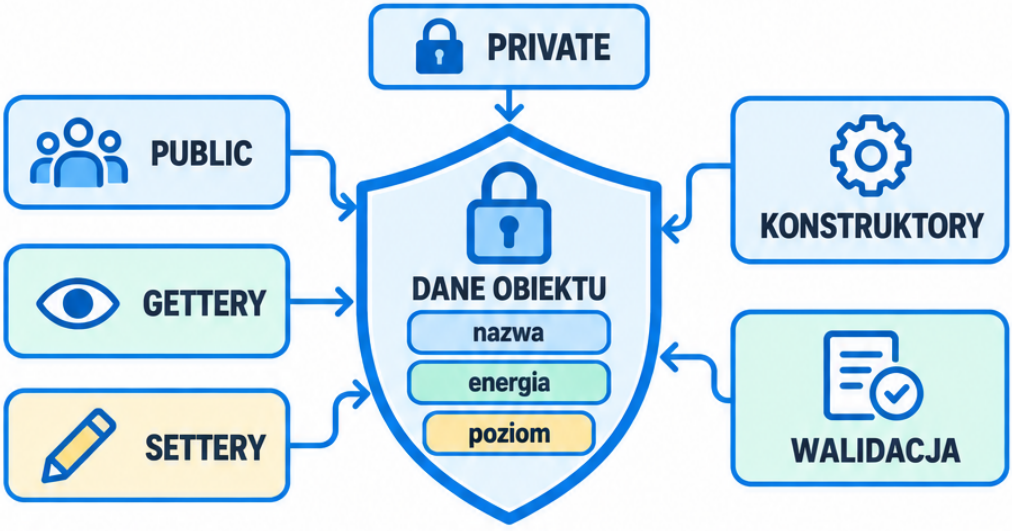
ODPOWIEDZI:

Pies · imie · szczekaj() · tworzy obiekt.

MODUŁ 02

ENKAPSULACJA I ZARZĄDZANIE STANEM

Dowiedz się, jak chronić dane obiektów i kontrolować ich zmiany.



Enkapsulacja pomaga chronić stan obiektu przed niekontrolowanymi zmianami.

Dzięki odpowiednio zaprojektowanym metodom możemy decydować, które dane można odczytywać i w jaki sposób wolno je modyfikować.

1. **Zrozumiesz zasadę enkapsulacji.**
2. **Poznasz modyfikatory dostępu.**
3. **Nauczysz się używać konstruktorów.**
4. **Odkryjesz zastosowanie getterów i setterów.**
5. **Poznasz podstawy walidacji danych.**

ENKAPSULACJA — CHROŃ DANE SWOJEGO OBIEKTU

BEZ ENKAPSULACJI

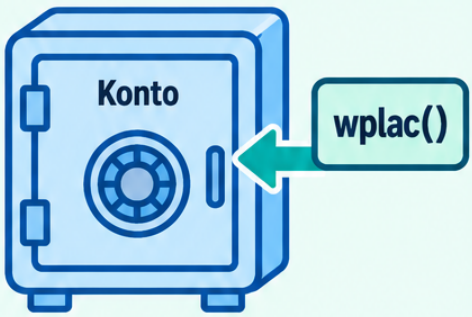
```
class Konto {
    public double saldo;
}

Konto konto = new Konto();
konto.saldo = -500;
```

PROBLEM:
Można bezpośrednio ustawić nieprawidłowe saldo.



Z ENKAPSULACJĄ

```
class Konto {
    private double saldo;
    public void wplac(
        double kwota) {
        if (kwota > 0) {
            saldo += kwota;
        }
    }
}
```



PRIVATE

Ogranicza bezpośredni dostęp do pola z innych klas.

PUBLIC

Pozwala udostępniać wybrane metody.

KONTROLA

Metoda sprawdza warunek przed zmianą danych.

Enkapsulacja nie oznacza ukrywania wszystkiego. Oznacza świadome kontrolowanie dostępu do danych i zachowań obiektu.

PUBLIC, PRIVATE, PROTECTED — KTO MA DOSTĘP?

Modyfikator	Dostęp
`public`	Z każdego miejsca, w którym klasa jest dostępna
`private`	Tylko wewnątrz tej samej klasy
`protected`	W tym samym pakiecie oraz w podklasach zgodnie z zasadami Javy
`brak modyfikatora`	W obrębie tego samego pakietu

```
class Robot {
    public String nazwa;
    private int energia;
    protected int poziom;
    String model;
}
```



PUBLIC
Dostęp szeroki.



PRIVATE
Dostęp tylko w klasie.



PROTECTED
Dostęp pakietowy i dla podklas.



PACKAGE-PRIVATE
Dostęp w tym samym pakiecie.

Brak modyfikatora to dostęp pakietowy, często nazywany package-private.

GETTERY I SETTERY — KONTROLOWANY DOSTĘP DO DANYCH

GETTER — ODCZYT DANYCH

`getEnergia()`

Zwraca aktualną wartość pola.

OBIEKT ROBOT



`private int energia;`

SETTER — ZMIANA DANYCH

`setEnergia(int e)`

Umożliwia kontrolowaną zmianę pola.

```
class Robot {
    private int energia;

    public int getEnergia() {
        return energia;
    }

    public void setEnergia(int e) {
        if (e >= 0 && e <= 100) {
            energia = e;
        }
    }
}
```

Przykład użycia:

```
Robot robot = new Robot();
robot.setEnergia(80);
System.out.println(robot.getEnergia());
```

WYNIK: 80

1. `setEnergia(80)` → sprawdzenie wartości → zapis.
2. `getEnergia()` → odczyt → wynik 80.

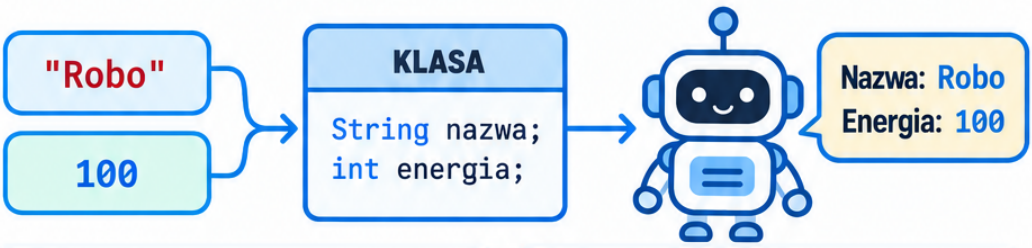
Nie każde prywatne pole musi mieć setter.
Udostępniaj tylko operacje potrzebne do poprawnego działania programu.

KONSTRUKTOR — JAK NADAĆ OBIEKTOWI STAN POCZĄTKOWY?

NEW → **KONSTRUKTOR** → **GOTOWY OBIEKT**

```
class Robot {
     String nazwa;
     int energia;

     Robot(String n, int e) {
         nazwa = n;
         energia = e;
     }
}
```

```
Robot robot = new Robot("Robo", 100);
```



1. NAZWA
Konstruktor ma taką samą nazwę jak klasa.

2. BRAK TYPU ZWROTNEGO
Konstruktor nie deklaruje typu zwracanego, nawet void.

3. PARAMETRY
Pozwalają przekazać dane początkowe.

4. WYWOŁANIE
Konstruktor jest uruchamiany podczas tworzenia obiektu.

Konstruktor pomaga przygotować nowy obiekt do działania.

THIS — JAK ODNIEŚĆ SIĘ DO BIEŻĄCEGO OBIEKTU?

BEZ THIS — RÓŻNE NAZWY

```
class Robot {
    String nazwa;

    Robot(String nowaNazwa) {
        nazwa = nowaNazwa;
    }
}
```

Z THIS — TAKIE SAME NAZWY

```
class Robot {
    String nazwa;

    Robot(String nazwa) {
        this.nazwa = nazwa;
    }
}
```

```
this.nazwa = nazwa;
```

this.nazwa
Pole należące do bieżącego obiektu.

nazwa
Parametr przekazany do konstruktora.



KIEDY UŻYWAMY THIS?

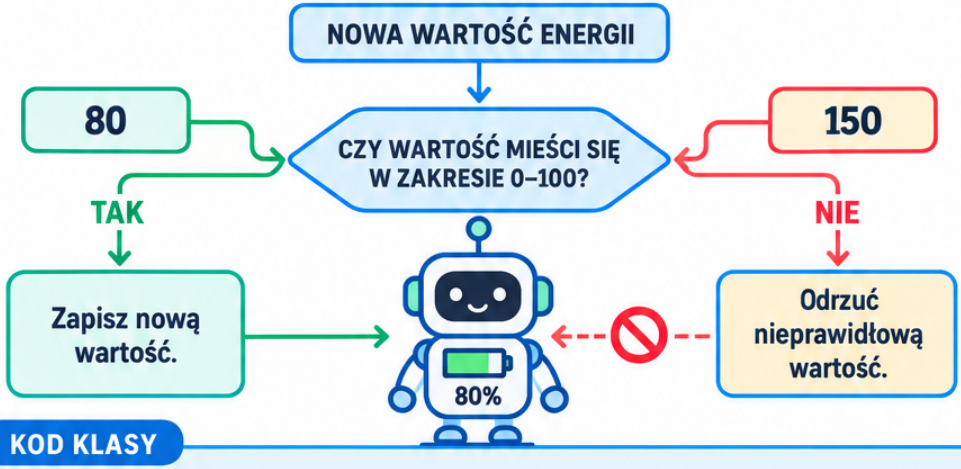
1. Aby wskazać pole bieżącego obiektu.
2. Aby rozróżnić pole i parametr o tej samej nazwie.
3. Aby wywołać inny konstruktor tej samej klasy za pomocą `this(...)`.

this jest referencją do bieżącego obiektu i pomaga pisać czytelniejszy kod.

WALIDACJA — NIE POZWÓL NA NIEPRAWIDŁOWY STAN

PROBLEM

Robot może mieć energię od 0 do 100. Co stanie się, jeśli ktoś spróbuje ustawić 150?



KOD KLASY

```
class Robot {
    private int energia = 50;

    public void setEnergia(int e) {
        if (e >= 0 && e <= 100) {
            energia = e;
        }
    }

    public int getEnergia() {
        return energia;
    }
}
```

UŻYCIE KLASY

```
Robot robot = new Robot();
robot.setEnergia(80);
robot.setEnergia(150);
System.out.println(robot.getEnergia());
```

WYNIK

```
WYNIK: 80
```

WYJAŚNIENIE

Wartość 150 została odrzucona, dlatego energia pozostała równa 80.

Walidacja pomaga utrzymywać poprawny stan obiektów.
Nieprawidłowe dane można odrzucić lub zgłosić błąd — zależnie od projektu programu.

WIELE KONSTRUKTORÓW W JEDNEJ KLASIE

PRZECIĄŻANIE KONSTRUKTORÓW

1. KONSTRUKTOR BEZ PARAMETRÓW

Tworzy obiekt z domyślnymi wartościami ustalonymi przez programistę.

2. KONSTRUKTOR Z JEDNYM PARAMETREM

Pozwala określić jedną wartość podczas tworzenia obiektu.

3. KONSTRUKTOR Z DWOMA PARAMETRAMI

Pozwala przekazać więcej danych początkowych.

```
class Robot {
      String nazwa;
      int energia;
      Robot() {
          this("Bez nazwy", 100);
      }
      Robot(String nazwa) {
          this(nazwa, 100);
      }
      Robot(String nazwa, int energia) {
          this.nazwa = nazwa;
          this.energia = energia;
      }
}
```

```
Robot r1 = new Robot();
Robot r2 = new Robot("Robo");
Robot r3 = new Robot("Mika", 70);
```



R1: Bez nazwy
Energia 100



R2: Robo
Energia 100



R3: Mika
Energia 70

Przeciążanie polega na tworzeniu konstruktorów o różnych listach parametrów.
Wywołanie this(...) pozwala jednemu konstruktorowi skorzystać z innego konstruktora tej samej klasy.

BEZPIECZNY OBIEKT — WSZYSTKIE ELEMENTY RAZEM

1. PRIVATE
Chroni pola przed bezpośrednią modyfikacją spoza klasy.

2. KONSTRUKTOR
Nadaje obiektowi początkowy stan.



BEZPIECZNE PROJEKTOWANIE KLASY

3. GETTER
Umożliwia kontrolowany odczyt danych.

4. SETTER
Umożliwia zmianę danych zgodnie z ustalonymi zasadami.

5. WALIDACJA
Sprawdza, czy podana wartość jest poprawna.

```java
class Robot {
    private String nazwa;
    private int energia;

    Robot(String nazwa) {
        this.nazwa = nazwa;
        this.energia = 100;
    }

    public int getEnergia() {
        return energia;
    }

    public void setEnergia(int energia) {
        if (energia >= 0 && energia <= 100) {
            this.energia = energia;
        }
    }
}
```

```java
Robot robot = new Robot("Robo");
robot.setEnergia(75);
System.out.println(robot.getEnergia());
```

WYNIK: 75

MINI WYZWANIE

1. Dlaczego pole `energia` jest prywatne?
2. Która metoda pozwala odczytać energię?
3. Co stanie się po wywołaniu `setEnergia(150)`?

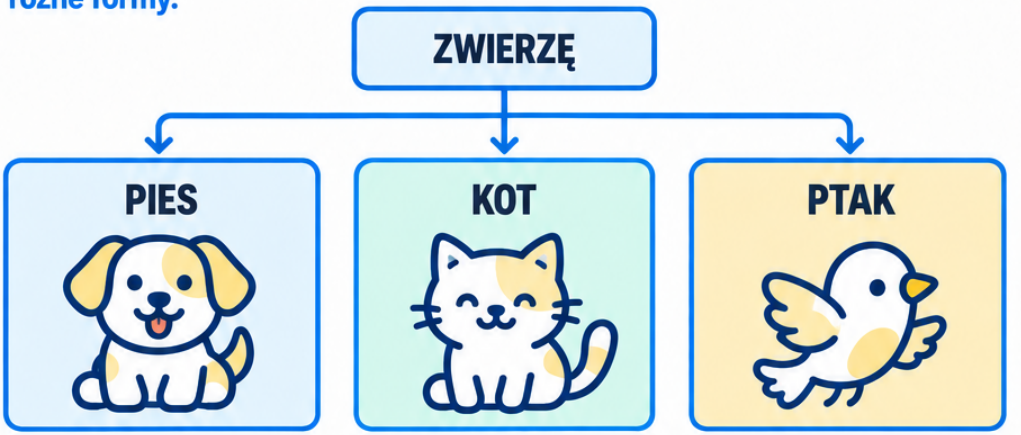
ODPOWIEDZI:

1. Aby kontrolować dostęp do pola.
2. `getEnergia()`.
3. Energia nie zostanie zmieniona.

MODUŁ 03

DZIEDZICZENIE, POLIMORFIZM I ABSTRAKCJA

Odkryj, jak klasy mogą współpracować, dziedziczyć zachowania i przyjmować różne formy.



DZIEDZICZENIE

Jedna klasa może rozszerzać inną klasę.

POLIMORFIZM

Jedno wywołanie metody może prowadzić do różnych zachowań.

ABSTRAKCJA

Pozwala skupić się na istotnych cechach i działaniach.

Dzięki tym mechanizmom można budować programy, które są łatwiejsze do rozszerzania, organizowania i utrzymywania.

1. **Zrozumiesz dziedziczenie klas.**
2. **Nauczysz się używać extends i super.**
3. **Poznasz nadpisywanie metod.**
4. **Odkryjesz działanie polimorfizmu.**
5. **Nauczysz się korzystać z klas abstrakcyjnych.**

DZIEDZICZENIE — JEDNA KLASA ROZSZERZA DRUGĄ

ZWIERZĘ
Klasa bazowa
oddychaj()

PIES



oddychaj()
szczekaj()

KOT



PTAK



KLASA BAZOWA
Zawiera wspólne pola i metody.

KLASA POCHODNA
Dziedziczy dostępne zachowania klasy bazowej i może dodawać własne.

```
class Zwierze {
  void oddychaj() {
    System.out.println("Oddycham");
  }
}

class Pies extends Zwierze {
  void szczekaj() {
    System.out.println("Hau!");
  }
}
```

UŻYCIE:

```
Pies pies = new Pies();
pies.oddychaj();
pies.szczekaj();
```

WYNIK:

```
Oddycham
Hau!
```

Dziedziczenie opisuje relację „jest rodzajem”.
Pies jest rodzajem zwierzęcia.

EXTENDS I SUPER — JAK ŁĄCZYĆ KLASY?

EXTENDS — DZIEDZICZENIE

Służy do określenia, że jedna klasa rozszerza drugą.

SCHEMAT DEKLARACJI KLASY

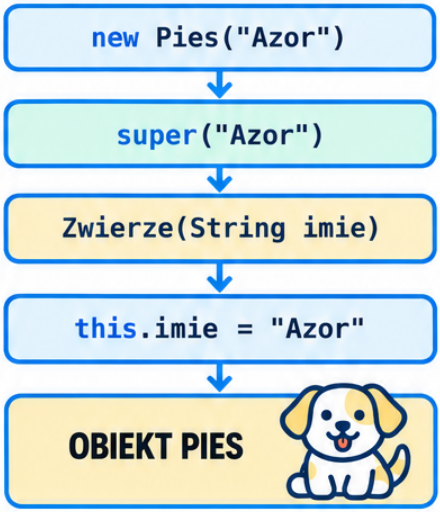
```
class Pies extends Zwierze { }
```

SUPER — KLASA BAZOWA

Pozwala odwołać się do wybranych elementów klasy bazowej, w tym jej konstruktora.



```
class Zwierze {
    String imie;
    Zwierze(String imie) {
        this.imie = imie;
    }
    void przedstawSie() {
        System.out.println(imie);
    }
}

class Pies extends Zwierze {
    Pies(String imie) {
        super(imie);
    }
}
```



PRZYKŁAD UŻYCIA:

```
Pies pies = new Pies("Azor");
pies.przedstawSie();
```

WYNIK:
Azor

NAJWAŻNIEJSZE OBSERWACJE:

1. `extends` tworzy relację dziedziczenia między klasami.
2. `super(...)` wywołuje konstruktor klasy bazowej.
3. Konstruktor klasy bazowej przygotowuje jej część obiektu.

Konstruktory nie są dziedziczone. Konstruktor klasy pochodnej może jednak wywołać konstruktor klasy bazowej.

@OVERRIDE – GDY KLASA ZMIENIA ODZIEDZICZONE ZACHOWANIE

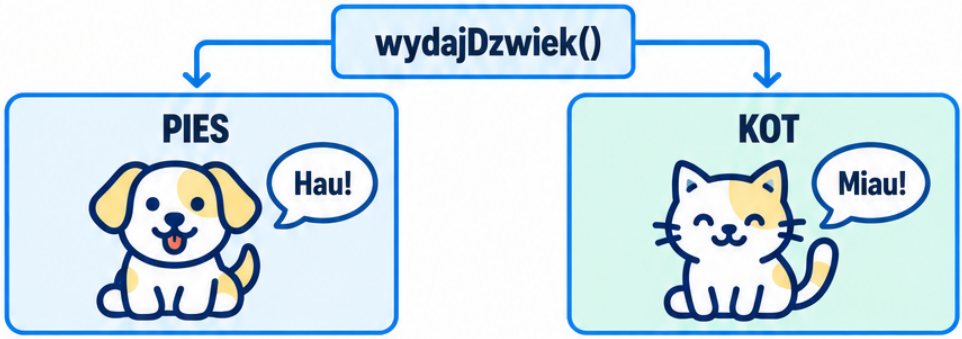
KLASA BAZOWA – ZWIERZE

```
class Zwierze {
    void wydajDzwiek() {
        System.out.println("Dźwięk");
    }
}
```

KLASY POCHODNE

```
class Pies extends Zwierze {
    @Override
    void wydajDzwiek() {
        System.out.println("Hau!");
    }
}
```

```
class Kot extends Zwierze {
    @Override
    void wydajDzwiek() {
        System.out.println("Miau!");
    }
}
```



1. TA SAMA NAZWA METODY

Metoda w klasie pochodnej ma zgodną sygnaturę.

2. NOWA IMPLEMENTACJA

Klasa pochodna określa własne działanie.

3. @OVERRIDE

Adnotacja pomaga kompilatorowi sprawdzić poprawność nadpisywania.

Nadpisywanie metod różni się od przeciążania.

- **Nadpisywanie**: zmiana odziedziczonej implementacji.
- **Przeciążanie**: ta sama nazwa, ale inna lista parametrów.

POLIMORFIZM — JEDNA METODA, RÓŻNE ZACHOWANIA

SCHEMAT POLIMORFIZMU

POLIMORFIZM

↓

JEDNO WYWOŁANIE
`wydajDzwiek()`

↓

RÓŻNE WYNIKI

PIES: Hau!	KOT: Miau!	KROWA: Muuu!
		

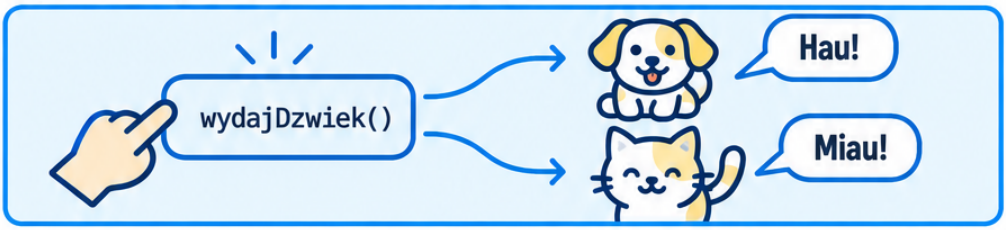
Klasy Zwierze, Pies i Kot z poprzedniej strony.

```
Zwierze a = new Pies();
Zwierze b = new Kot();

a.wydajDzwiek();
b.wydajDzwiek();
```

WYNIK KODU:

```
Hau!
Miau!
```

Chociaż zmienne **a** i **b** mają typ Zwierze, wywoływane są odpowiednie implementacje metod obiektów Pies i Kot.



Polimorfizm pozwala tworzyć kod, który współpracuje z różnymi obiektami bez konieczności sprawdzania każdego konkretnego typu.



To rzeczywisty typ obiektu decyduje, która nadpisana metoda instancyjna zostanie wykonana.

UPCASTING – OBIEKT KLASY POCHODNEJ JAKO TYP BAZOWY

Upcasting polega na przypisaniu obiektu klasy pochodnej do referencji typu jego klasy bazowej. Dzięki temu można korzystać z polimorfizmu i traktować różne obiekty przez wspólny typ.

REFERENCJA TYPU ZWIERZE

```
Zwierze zwierze = new Pies();
```

OBIEKT TYPU PIES



TYP REFERENCJI

`Zwierze`

Określa, jakie metody są dostępne bez rzutowania.

TYP OBIEKTU

`Pies`

Określa rzeczywisty rodzaj utworzonego obiektu.

WARSTWA ZEWNĘTRZNA

Referencja: `Zwierze`

WARSTWA WEWNĘTRZNA

Obiekt: `Pies`



`Pies.wydajDzwiek()`



WYWOŁANIE METODY Z OBIEKTU

```
zwierze.wydajDzwiek();
```

WYNIK: `Hau!`

PRÓBA WYWOŁANIA NIEDOSTĘPNEJ METODY

```
zwierze.szczekaj();
```

BŁĄD KOMPILACJI

DODATKOWA METODA KLASY PIES

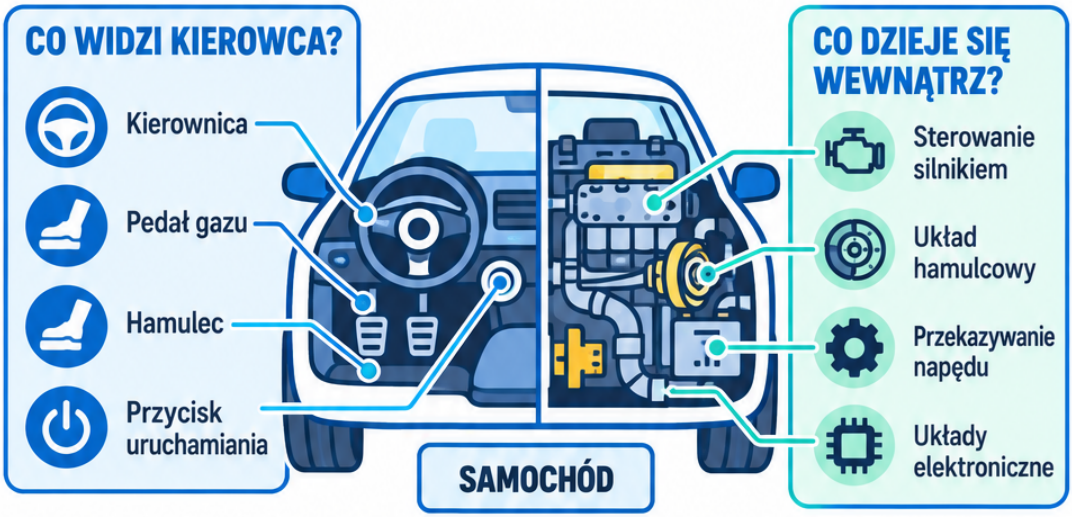
```
void szczekaj() {
  System.out.println("Hau!");
}
```

Jeśli metoda `szczekaj()` nie istnieje w klasie `Zwierze`, referencja typu `Zwierze` nie pozwala wywołać jej bezpośrednio.

PODSUMOWANIE

Upcasting pozwala traktować różne obiekty przez wspólny typ bazowy, zachowując ich polimorficzne zachowanie.

ABSTRAKCJA –
SKUP SIĘ NA TYM, CO NAJWAŻNIEJSZE



Kierowca korzysta z prostych elementów sterowania bez konieczności poznawania wszystkich szczegółów technicznych samochodu.

przykładowy interfejs obiektu

```
samochod.uruchom();
samochod.przyspiesz();
samochod.zatrzymaj();
```

1. PROSTOTA

Udostępniaj najważniejsze operacje.

2. UKRYCIE SZCZEGÓŁÓW

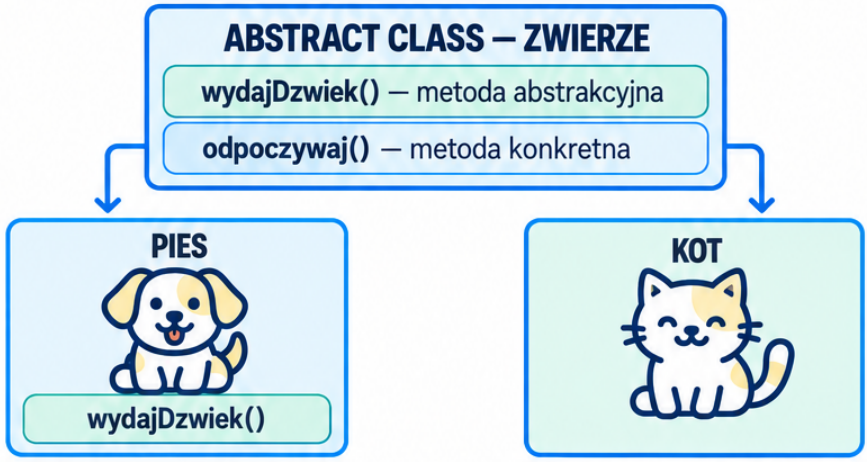
Nie wymagaj od użytkownika znajomości implementacji.

3. CZYTELNOŚĆ

Stosuj zrozumiałe nazwy i jasne odpowiedzialności.

Abstrakcja pomaga zarządzać złożonością programu, pokazując użytkownikowi obiektu tylko potrzebne informacje i operacje.

ABSTRACT — TWÓRZ WSPÓLNE ZASADY DLA WIELU KLAS



Klasa abstrakcyjna może definiować wspólne pola, gotowe metody oraz metody abstrakcyjne, które muszą zostać zaimplementowane przez pierwszą konkretną podklasę, jeśli nie zostały wcześniej zaimplementowane.

```
abstract class Zwierze {
  abstract void wydajDzwiek();
  void odpoczywaj() {
    System.out.println(
      "Odpoczywam");
  }
}
```

```
class Pies extends Zwierze {
  @Override
  void wydajDzwiek() {
    System.out.println("Hau!");
  }
}
```

```
Zwierze pies = new Pies();
pies.wydajDzwiek();
pies.odpoczywaj();
```

WYNIK:

```
Hau!
Odpoczywam
```

Klasa abstrakcyjna	Klasa konkretna
Nie można utworzyć jej instancji bezpośrednio	Można tworzyć jej obiekty
Może zawierać metody abstrakcyjne	Implementuje wymagane metody
Definiuje wspólną podstawę	Dostarcza konkretne zachowanie

Klasy abstrakcyjne łączą wspólne zachowania z zasadami, które muszą spełnić konkretne podklasy.

JEDNA LISTA, WIELE TYPÓW OBIEKTÓW

WSPÓLNY TYP — ZWIERZE

PIES



`wydajDzwiek()` → Hau!

KOT



`wydajDzwiek()` → Miau!

KROWA



`wydajDzwiek()` → Muuu!

Zakładamy, że klasy Pies, Kot i Krowa rozszerzają Zwierze i nadpisują metodę `wydajDzwiek()`.

Kod z tablicą obiektów:

```
Zwierze[] zwierzeta = {
   new Pies(),
   new Kot(),
   new Krowa()
};
for (Zwierze z : zwierzeta) {
   z.wydajDzwiek();
}
```

WYNIK:

```
Hau!
Miau!
Muuu!
```

TABLICA ZWIERZĄT → **PĘTLA FOR** → **WYWOŁANIE METODY**







1. WSPÓLNY TYP

Wszystkie obiekty można traktować jako Zwierze.

2. JEDNO WYWOŁANIE

Każdy obiekt otrzymuje to samo wywołanie metody.

3. RÓŻNE ZACHOWANIA

Wykonywana jest implementacja odpowiednia dla rzeczywistego typu obiektu.



Polimorfizm pozwala obsługiwać różne obiekty za pomocą wspólnego kodu.

DZIEDZICZENIE, POLIMORFIZM I ABSTRAKCJA – MAPA WIEDZY

1. DZIEDZICZENIE

Klasa pochodna rozszerza klasę bazową.

Słowo kluczowe: **extends**

2. SUPER

Umożliwia odwołanie się do konstruktora lub dostępnych elementów klasy bazowej.

3. NADPISYWANIE

Klasa pochodna dostarcza własną implementację odziedziczonej metody.

Adnotacja: **@Override**

ZAAWANSOWANE PODSTAWY OOP



4. POLIMORFIZM

Wspólny typ może reprezentować obiekty różnych klas pochodnych.

5. ABSTRAKCJA

Pozwala skupić się na potrzebnych operacjach, ukrywając zbędne szczegóły.

6. KLASA ABSTRAKCYJNA

Definiuje wspólną podstawę dla klas pochodnych.

Słowo kluczowe: **abstract**

MINI WYZWANIE

```
abstract class Pojazd {
    abstract void jedz();
}
class Rower extends Pojazd {
    @Override
    void jedz() {
        System.out.println("Jadę!");
    }
}
```

PYTANIE 1

Która klasa jest abstrakcyjna?

PYTANIE 2

Która klasa dziedziczy po innej?

PYTANIE 3

Która metoda została zaimplementowana w klasie pochodnej?

ODPOWIEDZI

1. `Pojazd`
2. `Rower`
3. `jedz()`

MODUŁ 04

INTERFEJSY, KOMPOZYCJA I PROJEKTOWANIE KLAS

Naucz się tworzyć elastyczne programy złożone ze współpracujących obiektów.



Dobry program obiektowy to nie tylko zbiór klas.
To system, w którym obiekty mają jasno określone zadania i potrafią ze sobą współpracować.

1. **Zrozumiesz działanie interfejsów.**
2. **Nauczysz się używać implements.**
3. **Poznasz różnice między interfejsem a klasą abstrakcyjną.**
4. **Odkryjesz kompozycję obiektów.**
5. **Nauczysz się projektować klasy o jasnych odpowiedzialnościach.**

INTERFEJS — WSPÓLNY KONTRAKT DLA KLAS

INTERFEJS – LATAJACY

`lec()`

PTAK



Potrafi latać.

DRON



Potrafi latać.

```
interface Latajacy {
    void lec();
}
```

```
class Ptak implements Latajacy {
   public void lec() {
     System.out.println("Ptak leci");
   }
}
```

```
class Dron implements Latajacy {
   public void lec() {
     System.out.println("Dron leci");
   }
}
```

INTERFEJS

Określa, jakie operacje powinien udostępniać obiekt.

IMPLEMENTS

Informuje, że klasa realizuje wskazany interfejs.

IMPLEMENTACJA

Klasa określa sposób wykonania wymaganych metod.

Metody abstrakcyjne deklarowane w interfejsie są domyślnie publiczne. Ich implementacje w klasach muszą zachować publiczny dostęp.

Interfejs określa wspólny kontrakt, a klasy dostarczają konkretne zachowania.

JEDNA KLASA MOŻE IMPLEMENTOWAĆ WIELE INTERFEJSÓW

INTERFEJS LATAJACY



Metoda: lec()

KLASA AMFIBIA



lec()
plywaj()

INTERFEJS PLYWAJACY



Metoda: plywaj()

```java
interface Latajacy {
    void lec();
}
```

```java
interface Plywajacy {
    void plywaj();
}
```

```java
class Amfibia implements Latajacy, Plywajacy {
    public void lec() {
        System.out.println("Lecę");
    }

    public void plywaj() {
        System.out.println("Płynę");
    }
}
```

UŻYCIE:

```java
Amfibia pojazd = new Amfibia();
pojazd.lec();
pojazd.plywaj();
```

WYNIK:

```
Lecę
Płynę
```

1. WIELE KONTRAKTÓW

Klasa może implementować więcej niż jeden interfejs.

2. RÓŻNE MOŻLIWOŚCI

Każdy interfejs opisuje określony zestaw zachowań.

3. JEDEN OBIEKT

Obiekt może udostępniać wszystkie zaimplementowane operacje.



Java pozwala implementować wiele interfejsów, chociaż klasa może bezpośrednio rozszerzać tylko jedną klasę.

INTERFEJS CZY KLASA ABSTRAKCYJNA? POZNAJ RÓŻNICĘ

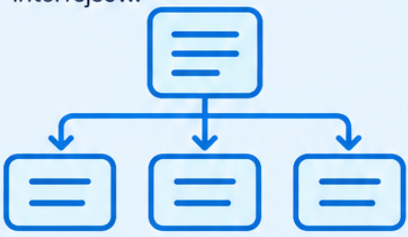
INTERFEJS

Cel:
Określenie wspólnego kontraktu.

Słowo kluczowe: `interface`
Implementacja: `implements`

Charakterystyka:

- Może deklarować metody abstrakcyjne.
- Może zawierać metody domyślne i statyczne.
- Nie przechowuje zwykłych pól instancji.
- Klasa może implementować wiele interfejsów.



```
interface Latajacy {
  void lec();
}
```

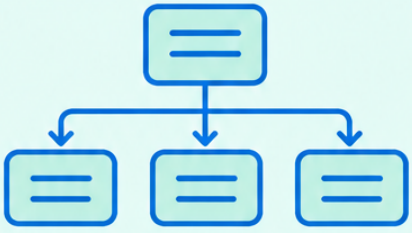
KLASA ABSTRAKCYJNA

Cel:
Udostępnienie wspólnej podstawy dla klas pochodnych.

Słowo kluczowe: `abstract class`
Dziedziczenie: `extends`

Charakterystyka:

- Może posiadać pola instancji.
- Może zawierać konstruktory.
- Może definiować metody abstrakcyjne.
- Może zawierać gotowe implementacje metod.



```
abstract class Zwierze {
   String imie;
   abstract void wydajDzwiek();
}
```

KIEDY WYBRAĆ INTERFEJS?

Gdy różne klasy mają oferować wspólną możliwość, na przykład latanie.

KIEDY WYBRAĆ KLASĘ ABSTRAKCYJNĄ?

Gdy klasy pochodne potrzebują wspólnego stanu lub wspólnej implementacji.

Interfejs opisuje przede wszystkim kontrakt, a klasa abstrakcyjna może dodatkowo udostępniać wspólny stan i implementację.

KOMPOZYCJA — BUDUJ OBIEKTY Z INNYCH OBIEKTÓW

SAMOCHÓD



SILNIK



Odpowiada za napęd.

KOŁA



Umożliwiają poruszanie się.

HAMULCE



Pozwalają zatrzymać pojazd.

Przykład uproszczony

```
class Silnik {
  void uruchom() {
    System.out.println("Silnik działa");
  }
}

class Samochod {
  private final Silnik silnik = new Silnik();
  void uruchom() {
    silnik.uruchom();
  }
}
```

Użycie:

```
Samochod auto = new Samochod();
auto.uruchom();
```

Wynik:

```
WYNIK: Silnik działa
```

Wyjaśnienie:

Klasa `Samochod` zawiera obiekt `Silnik` i korzysta z jego metody.

SAMOCHÓD → **MA** → **SILNIK**

PIES → **JEST RODZAJEM** → **ZWIERZĘ**

1. RELACJA „MA”

Jeden obiekt posiada inny obiekt jako część swojej struktury.

2. WSPÓŁPRACA

Obiekty przekazują sobie zadania za pomocą metod.

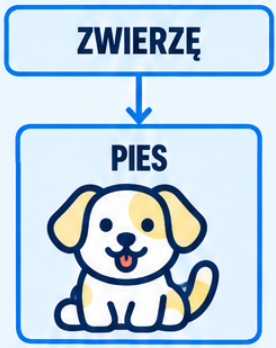
3. MODUŁOWOŚĆ

Każda klasa może odpowiadać za inną część działania programu.

Kompozycja pozwala tworzyć większe systemy przez łączenie mniejszych obiektów.

DZIEDZICZENIE CZY KOMPOZYCJA? WYBIERZ DOBRĄ RELACJĘ

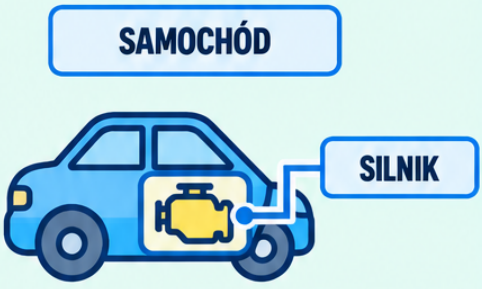
DZIEDZICZENIE – „JEST RODZAJEM”



Pies jest rodzajem zwierzęcia.

```
class Zwierze { }
class Pies extends Zwierze { }
```

KOMPOZYCJA – „MA”



Samochód ma silnik.

```
class Silnik { }
class Samochod {
  private final Silnik silnik
      = new Silnik();
}
```

Pytanie	Relacja
Czy obiekt jest rodzajem innego obiektu?	Dziedziczenie
Czy obiekt posiada inny obiekt jako część?	Kompozycja
Czy zachowanie ma być realizowane przez współpracujący komponent?	Rozważ kompozycję

DZIEDZICZENIE

Pozwala korzystać ze wspólnej struktury i zachowań klasy bazowej.

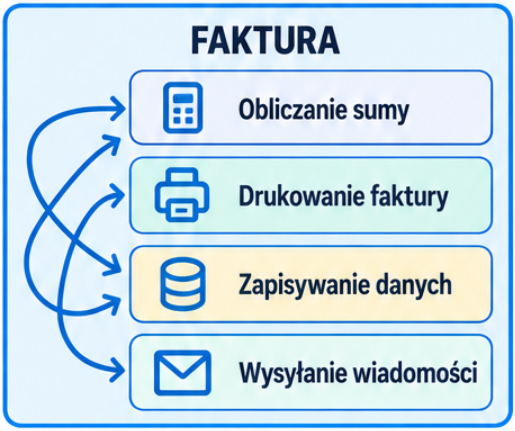
KOMPOZYCJA

Pozwala delegować działania do innych obiektów.

Nie używaj dziedziczenia tylko po to, by ponownie wykorzystać kod. Najpierw sprawdź, czy rzeczywiście istnieje relacja „jest rodzajem”.

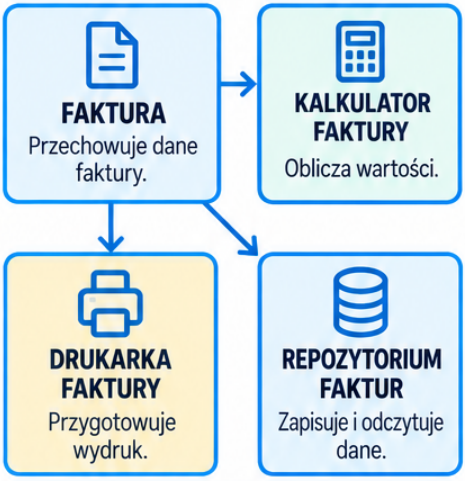
JEDNA KLASA — JEDNA GŁÓWNA ODPOWIEDZIALNOŚĆ

PRZED — JEDNA KLASA ROBI WSZYSTKO



PROBLEM
Zbyt wiele różnych odpowiedzialności w jednej klasie.

PO — PODZIAŁ NA KLASY



Wysyłanie wiadomości wydziel do osobnej klasy.

```
class Faktura {
    double kwota;
}

class KalkulatorFaktury {
    double obliczPodatek(
        Faktura faktura) {
        return faktura.kwota * 0.23;
    }
}
```

Każda klasa powinna mieć jasno określoną odpowiedzialność.
Zasada pojedynczej odpowiedzialności oznacza, że klasa powinna mieć jeden główny powód do zmiany.

1. CZYTELNOŚĆ
Łatwiej zrozumieć kod.

2. TESTOWANIE
Łatwiej sprawdzać działanie poszczególnych elementów.

3. ROZWÓJ
Zmiany w jednym obszarze rzadziej wpływają na inne części programu.

Dobrze zaprojektowane klasy mają jasne zadania i ograniczają niepotrzebne zależności.

ELASTYCZNY KOD – ŁĄCZ OBIEKTY PRZEZ INTERFEJSY

SYTUACJA

Aplikacja musi wysyłać powiadomienia. Czasami przez e-mail, a czasami przez SMS.




```
interface Powiadomienie {
   void wyslij(String tekst);
}
class Serwis {
   private final Powiadomienie
      kanal;
   Serwis(
      Powiadomienie kanal) {
      this.kanal = kanal;
   }
   void powiadom(String tekst) {
      kanal.wyslij(tekst);
   }
}
```

Serwis nie zależy bezpośrednio od klasy Email ani Sms.

```
class Email implements
   Powiadomienie {
   public void wyslij(
      String tekst) {
      System.out.println(
         "Email: " + tekst);
   }
}
class Sms implements
   Powiadomienie {
   public void wyslij(
      String tekst) {
      System.out.println(
         "SMS: " + tekst);
   }
}
```

```
Serwis serwis = new Serwis(new Email());
serwis.powiadom("Witaj!");
```

WYNIK:
Email: Witaj!

1. INTERFEJS
Definiuje wspólną operację.

2. KOMPOZYCJA
Serwis przechowuje referencję do współpracującgo obiektu.

3. ELASTYCZNOŚĆ
Można przekazać inną implementację bez zmiany klasy Serwis.

Przykład pokazuje wypisywanie komunikatów w konsoli. Nie wysyła rzeczywistych wiadomości.

Programowanie oparte na interfejsach ułatwia wymianę komponentów, testowanie i rozwijanie aplikacji.

INTERFEJSY, KOMPOZYCJA I DOBRY PROJEKT – MAPA WIEDZY

DOBRE PROJEKTOWANIE OOP

INTERFEJS
Określa wspólny kontrakt dla różnych klas.

Słowo kluczowe: `interface`

IMPLEMENTACJA
Klasa realizuje operacje określone przez interfejs.

Słowo kluczowe: `implements`

KLASA ABSTRAKCYJNA
Może udostępniać wspólny stan, metody i zasady dla podklas.

KOMPOZYCJA
Pozwala tworzyć większe obiekty z mniejszych elementów.

Relacja: „ma".

JEDNA ODPOWIEDZIALNOŚĆ
Każda klasa powinna mieć jasno określony cel.

ELASTYCZNOŚĆ
Program może współpracować z różnymi implementacjami wspólnego interfejsu.

DZIEDZICZENIE



PIES → ZWIERZĘ

Relacja „jest rodzajem".

KOMPOZYCJA



SAMOCHÓD → SILNIK

Relacja „ma".

INTERFEJS



DRON → LATAJACY

Relacja „realizuje kontrakt".

MINI WYZWANIE	ODPOWIEDZI
1. Czym różni się extends od implements?	1. **extends** służy do dziedziczenia, a implements do implementowania interfejsów.
2. Kiedy warto wybrać kompozycję?	2. Gdy obiekt współpracuje z innymi obiektami lub zawiera je jako części.
3. Dlaczego warto rozdzielać odpowidzialności klas?	3. Aby kod był czytelniejszy, łatwiejszy do testowania i rozwijania.

Dobre OOP polega na tworzeniu klas, które mają jasne zadania i potrafią ze sobą współpracować.

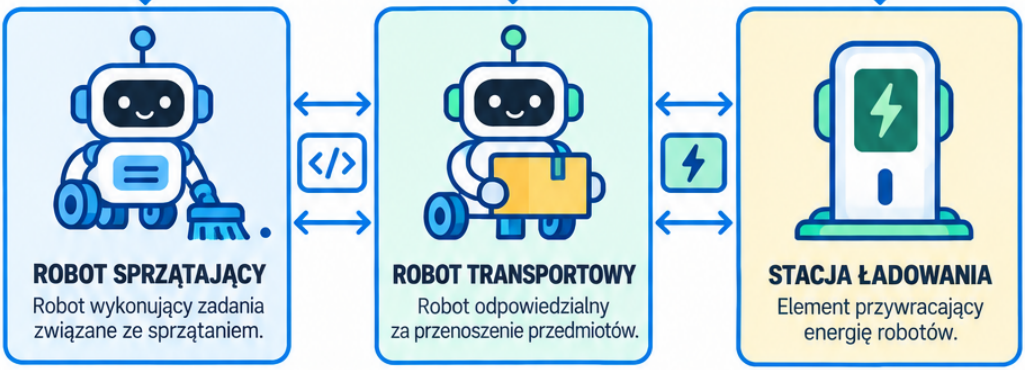
MODUŁ 05

PROJEKT PRAKTYCZNY I UTRWALENIE WIEDZY

Zbuduj własny symulator robotów i wykorzystaj najważniejsze zasady OOP w Javie.

MINI SYMULATOR ROBOTÓW

Stworzysz aplikację Java, w której różne roboty wykonują zadania i zarządzają swoją energią.



ROBOT SPRZĄTAJĄCY
Robot wykonujący zadania związane ze sprzątaniem.

ROBOT TRANSPORTOWY
Robot odpowiedzialny za przenoszenie przedmiotów.

STACJA ŁADOWANIA
Element przywracający energię robotów.

WYKORZYSTASZ W PROJEKCIE

- Klasy i obiekty.
- Dziedziczenie.
- Pola i metody.
- Polimorfizm.
- Konstruktory.
- Klasy abstrakcyjne.
- Enkapsulacja.
- Interfejsy i współpraca obiektów.

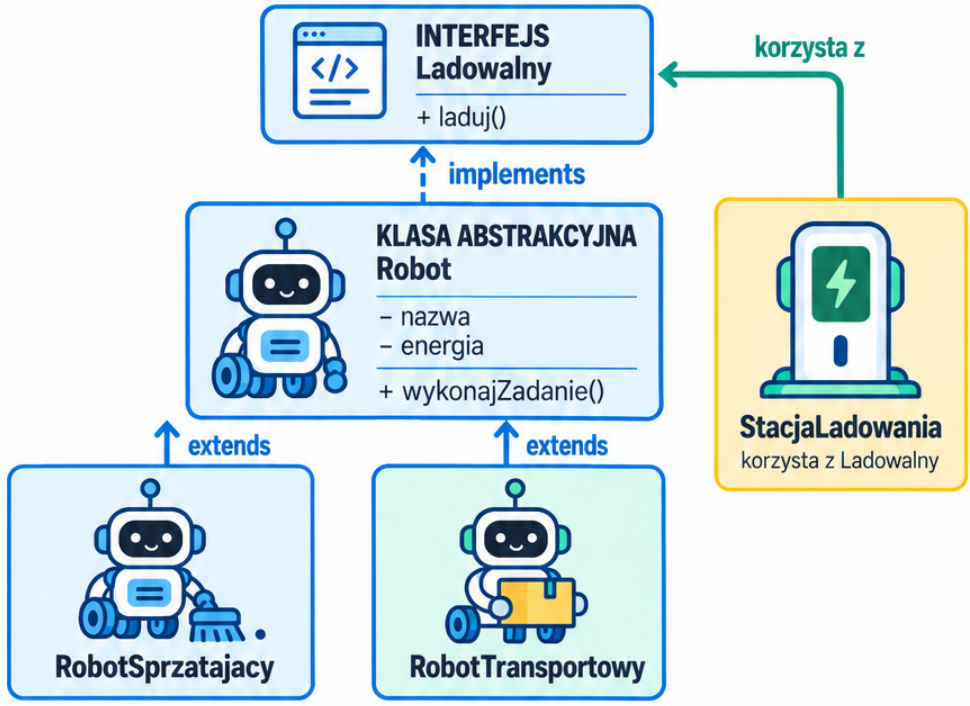
PO UKOŃCZENIU PROJEKTU:

Będziesz potrafić zaprojektować prostą hierarchię klas, tworzyć wyspecjalizowane obiekty, korzystać ze wspólnego interfejsu i uruchamiać program wykorzystujący polimorfizm.

ZAPROJEKTUJ SYSTEM, ZANIM NAPISZESZ KOD

NASZ CEL

Chcemy stworzyć program, w którym różne roboty wykonują zadania, zużywają energię i mogą zostać naładowane.



1.	ROBOT	Wspólna abstrakcyjna klasa bazowa.
2.	ROBOTSPRZATAJACY	Specjalizacja wykonująca sprzątanie.
3.	ROBOTTRANSPORTOWY	Specjalizacja wykonująca transport.
4.	LADOWALNY	Interfejs określający możliwość ładowania.
5.	STACJALADOWANIA	Klasa współpracująca z obiektami implementującymi Ladowalny.

Najpierw określ odpowiedzialności klas i relacje między nimi. Dopiero później przejdź do pisania kodu.

KROK 1 — STWÓRZ WSPÓLNĄ KLASĘ ROBOT

NASZE WYMAGANIA

Każdy robot powinien mieć:

1. własną nazwę;
2. określony poziom energii;
3. możliwość odczytu danych;
4. metodę wykonywania zadania;
5. możliwość ładowania.

ETAP 1 — INTERFEJS

Ladowalny.java

```
interface Ladowalny {
  void laduj();
}
```

ETAP 2 — KLASA ABSTRAKCYJNA

Robot.java

```
abstract class Robot
    implements Ladowalny {
  private final String nazwa;
  private int energia = 100;
  protected Robot(String nazwa) {
    if (nazwa == null
        || nazwa.isBlank()) {
      throw new
          IllegalArgumentException();
    }
    this.nazwa = nazwa;
  }
  public String getNazwa() {
    return nazwa;
  }
  public int getEnergia() {
    return energia;
  }
}
```

FINAL
Zapobiega ponownemu przypisaniu wartości do pola nazwa.

PRIVATE
Chroni stan obiektu.

PROTECTED ROBOT(...)
Udostępnia konstruktor klasom pochodnym.

GETTERY
Pozwalają odczytywać dane bez bezpośredniego dostępu do pól.

- Przykład wymaga Javy 11 lub nowszej.
- Metody zachowania dodamy do tej klasy na następnej stronie.

Klasa bazowa przechowuje elementy wspólne dla wszystkich robotów.

MODUŁ 05

KROK 2 — DODAJ METODY I KONTROLĘ ENERGII

ZASADY ENERGII

- Każdy robot rozpoczyna pracę ze 100 punktami energii.
- Wykonanie zadania zużywa określoną ilość energii.
- Robot nie może mieć ujemnej energii.
- Ładowanie przywraca energię do 100.

Dodaj poniższe metody **WEWNĄTRZ** klasy **Robot** ze strony 43, przed jej ostatnią klamrą.

1. ETAP 1 — ZUŻYWANIE ENERGII

```
protected boolean zuzyjEnergie(
    int koszt) {
    if (koszt <= 0
        || koszt > energia) {
        return false;
    }
    energia -= koszt;
    return true;
}
```

2. ETAP 2 — ŁADOWANIE

```
@Override
public void laduj() {
    energia = 100;
}
```

3. ETAP 3 — ZADANIE ABSTRAKCYJNE

```
public abstract void
    wykonajZadanie();
```



Metoda `zuzyjEnergie()` zwraca wartość `true`, gdy udało się pobrać energię, lub `false`, gdy operacja nie była możliwa.

KROK 3 – STWÓRZ DWA RODZAJE ROBOTÓW



ROBOT SPRZĄTAJĄCY

Zadanie: sprzątanie.
Koszt energii: 20 punktów.

RobotSprzatajacy.java

```
class RobotSprzatajacy
    extends Robot {

  RobotSprzatajacy(
      String nazwa) {
      super(nazwa);
  }

  @Override
  public void
      wykonajZadanie() {
      if (zuzyjEnergie(20)) {
          System.out.println(
            getNazwa()
            + " sprzata.");
      } else {
          System.out.println(
            "Za mało energii.");
      }
  }
}
```



ROBOT TRANSPORTOWY

Zadanie: transport.
Koszt energii: 30 punktów.

RobotTransportowy.java

```
class RobotTransportowy
    extends Robot {

  RobotTransportowy(
      String nazwa) {
      super(nazwa);
  }

  @Override
  public void
      wykonajZadanie() {
      if (zuzyjEnergie(30)) {
          System.out.println(
            getNazwa()
            + " transportuje.");
      } else {
          System.out.println(
            "Za mało energii.");
      }
  }
}
```



SPRZĄTANIE





TRANSPORT



EXTENDS
Tworzy relację dziedziczenia.

SUPER
Wywołuje konstruktor klasy bazowej.

OVERRIDE
Pozwala dostarczyć własną implementację metody.

Obie klasy korzystają ze wspólnych mechanizmów, ale każda realizuje własne zadanie.

MODUŁ 05

KROK 4 – POŁĄCZ OBIEKTY PRZEZ INTERFEJS

PROBLEM

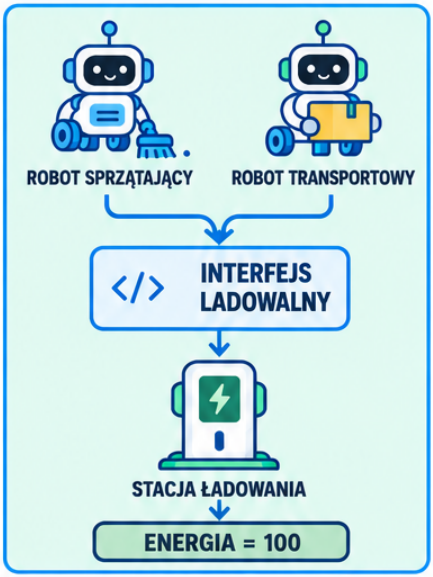
Potrzebujemy stacji, która potrafi ładować różne roboty bez znajomości ich konkretnych klas.

ROZWIĄZANIE

Stacja będzie korzystać z interfejsu Ladowalny.

StacjaLadowania.java

```
class StacjaLadowania {
  private final Ladowalny
      podlaczony;

  StacjaLadowania(
      Ladowalny podlaczony) {
    this.podlaczony = podlaczony;
  }

  void uruchom() {
      podlaczony.laduj();
  }
}
```



PRZYKŁAD UŻYCIA

```
Robot robot = new RobotSprzatajacy("Robo");
robot.wykonajZadanie();
StacjaLadowania stacja =
    new StacjaLadowania(robot);
stacja.uruchom();
```

INTERFEJS

Stacja wymaga obiektu, który potrafi się ładować.

REFERENCJA DO OBIEKTU

Stacja przechowuje referencję do podłączonego obiektu.

ELASTYCZNOŚĆ

Ta sama klasa stacji może obsługiwać różne implementacje Ladowalny.

UWAGA TECHNICZNA

Stacja współpracuje z przekazanym obiektem, ale nie zarządza jego całym cyklem życia. Jest to relacja przez referencję, a nie ścisła kompozycja UML.

Programowanie przez interfejsy pozwala tworzyć komponenty, które można łatwiej wymieniać i rozwijać.

MODUŁ 05

KROK 5 – URUCHOM RÓŻNE ROBOTY JEDNYM POLECENIEM

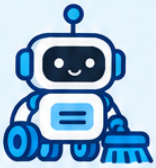
TABLICA ROBOTÓW

ROBOTSPRZATAJACY
Robo

ROBOTTRANSPORTOWY
Mika

PĘTLA FOR

wykonajZadanie()



ROBO
Sprzątanie
Energia: 80



MIKA
Transport
Energia: 70

PRZYKŁADOWY KOD JAVA

```
1  Robot[] roboty = {
2      new RobotSprzatajacy("Robo"),
3      new RobotTransportowy("Mika")
4  };
5  for (Robot robot : roboty) {
6      robot.wykonajZadanie();
7      System.out.println(
8          robot.getNazwa() + ": "
           + robot.getEnergia());
   }
```

WYNIK

```
Robo sprząta.
Robo: 80
Mika transportuje.
Mika: 70
```

1. WSPÓLNY TYP

Wszystkie elementy tablicy mają typ referencji **Robot**.

2. RÓŻNE OBIEKTY

Tablica przechowuje referencje do różnych podklas.

3. POLIMORFIZM

Każdy obiekt wykonuje własną implementację metody **wykonajZadanie()**.

Dzięki polimorfizmowi można obsługiwać wiele typów robotów za pomocą jednej pętli.

MODUŁ 05

KROK 6 – POŁĄCZ WSZYSTKO W DZIAŁAJĄCY PROGRAM

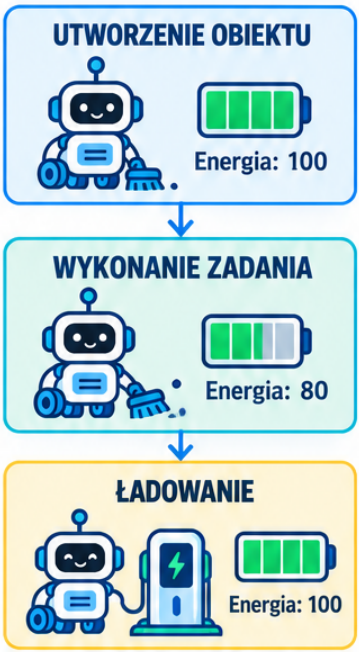
SPRAWDŹ PLIKI PROJEKTU

- Ladowalny.java
- RobotTransportowy.java
- Robot.java
- StacjaLadowania.java
- RobotSprzatajacy.java
- Main.java

ETAP 1 — UTWÓRZ KLASĘ MAIN

Main.java

```
public class Main {
   public static void main(
          String[] args) {
      Robot robo =
         new RobotSprzatajacy("Robo");
      robo.wykonajZadanie();
      System.out.println(
         "Energia: " + robo.getEnergia());
      StacjaLadowania stacja =
         new StacjaLadowania(robo);
      stacja.uruchom();
      System.out.println(
         "Po ładowaniu: "
         + robo.getEnergia());
   }
}
```



ETAP 2 — URUCHOM PROGRAM

Zapisz definicje ze stron 43–46 w odpowiednich plikach.
W pliku Robot.java połącz kod ze stron 43 i 44.
Umieść wszystkie pliki w jednym katalogu.
Wymagany JDK 11 lub nowszy.

```
javac *.java
java Main
```

ETAP 3 — SPRAWDŹ WYNIK

```
Robo sprząta.
Energia: 80
Po ładowaniu: 100
```

Gratulacje! Połączyłeś klasy, obiekty, dziedziczenie, interfejsy, enkapsulację i polimorfizm w jednym programie.

GRATULACJE! ROZUMIESZ PODSTAWY OOP W JAVIE

Teraz wykorzystaj zdobytą wiedzę i rozwijaj własne projekty.

PROGRAMOWANIE OBIEKTOWE W JAVIE

1. KLASY
Definiują strukturę i zachowania obiektów.

2. OBIEKTY
Są konkretnymi instancjami klas.

3. ENKAPSULACJA
Chroni stan i kontroluje dostęp do danych.

4. DZIEDZICZENIE
Pozwala tworzyć klasy pochodne.

5. POLIMORFIZM
Umożliwia różne zachowania przy wspólnym typie.

6. ABSTRAKCJA
Pomaga ukrywać zbędne szczegóły.

7. INTERFEJSY
Definiują kontrakty współpracy.

8. KOMPOZYCJA
Pozwala budować systemy z połączonych obiektów.

WYZWANIE FINAŁOWE — ROZBUDUJ SYMULATOR

1. 

POZIOM 1 — NOWY ROBOT
Dodaj klasę RobotOgrodniczy, która zużywa 15 punktów energii podczas wykonywania zadania.

2. 

POZIOM 2 — WIĘCEJ ZADAŃ
Utwórz tablicę trzech robotów i wykonaj zadania dla każdego z nich.

3. 

POZIOM 3 — TEST ENERGII
Sprawdź, co dzieje się po wielokrotnym wykonywaniu zadań. Upewnij się, że energia nigdy nie spada poniżej zera.

CHECKLISTA WIEDZY

- ☐ Rozumiem różnicę między klasą a obiektem.
- ☐ Potrafię tworzyć konstruktory.
- ☐ Wiem, jak działa enkapsulacja.
- ☐ Rozumiem dziedziczenie i extends.
- ☐ Potrafię wyjaśnić polimorfizm.
- ☐ Znam zastosowanie klas abstrakcyjnych.
- ☐ Potrafię implementować interfejsy.
- ☐ Rozumiem współpracę obiektów.
- ☐ Potrafię zbudować prosty program OOP.

TO DOPIERO POCZĄTEK!

Programowanie **obiektowe staje się łatwiejsze**, gdy regularnie tworzysz własne klasy, eksperymentujesz z kodem i analizujesz relacje między obiektami.

Nie musisz zapamiętać wszystkiego od razu. Najważniejsze jest zrozumienie zasad i wykorzystywanie ich w praktyce.

Powodzenia w dalszej nauce Javy!

