150 MAP MYŚLI Z JAVY

Do nauki, nawet jeśli zaczynasz od zera

Od podstaw składni i logiki aż po kolekcje, programowanie obiektowe i praktyczne projekty.

JAVA

zmienne

```
int wiek = 25;
String imie = "Java";
boolean aktywny = true;
```

pętle

```
for (int i = 0; i < 5; i++){
    System.out.println(i);
}
```

metody

```
public static void
    witaj() {
      System.out.println(
        "Cześć!");
}
```

tablice

```
int[] liczby = new int[5];
liczby[0] = 10;
```

0 1 2 3 4

ArrayList

```
ArrayList<String> lista =
new ArrayList<>();
lista.add("Java");
lista.add("Programowanie");
```

OOP

```
class Osoba {
   String imie;
   public void pokaz() {
     System.out.println(imie);
   }
}
```

wyjątki

```
try {
   // kod programu
} catch (Exception e) {
   System.out.println(e.getMessage());
}
```

projekty

- aplikacja konsolowa
- system zarządzania zadaniami
- prosty CRUD

COPYRIGHT

© 2026. WSZELKIE PRAWA ZASTRZEŻONE.

- Ten materiał ma charakter edukacyjny i został przygotowany z myślą o nauce języka Java w sposób wizualny i uporządkowany.
- Żadna część tego materiału nie może być kopiowana, rozpowszechniana ani publikowana bez zgody autora/wydawcy.
- Materiał przeznaczony jest do użytku osobistego.

NOTA PRAWNA

- Java jest zastrzeżonym znakiem towarowym firmy Oracle i/lub jej podmiotów stowarzyszonych.
- Ten materiał ma charakter niezależny i edukacyjny. Nie jest oficjalnie powiązany z firmą Oracle.

Wyniki zależą od regularnej nauki, praktyki i samodzielnego pisania kodu.

14 modułów
150 map myśli
od podstaw do
projektów!

SPIS TREŚCI

Twoja mapa
drogi w nauce
Javy!

00 Okładka

01 Copyright

02 Spis treści

03 Wprowadzenie

04-19 **Moduł 01:** Fundamenty i start z Javą

20-35 **Moduł 02:** Logika, warunki i pętle

36-51 **Moduł 03:** Metody, parametry i organizacja kodu

52-67 **Moduł 04:** Stringi, tablice i przetwarzanie danych

68-83 **Moduł 05:** Listy, kolekcje i generyki

84-99 **Moduł 06:** Klasy, obiekty i podstawy OOP

100-115 **Moduł 07:** Dziedziczenie, interfejsy i polimorfizm

116-131 **Moduł 08:** Wyjątki, pliki i operacje na danych

132-147 **Moduł 09:** Nowoczesna Java i dobre praktyki

148-163 **Moduł 10:** Projekty, wzorce myślenia i integracja wiedzy

JAK KORZYSTAĆ z tych map, żeby naprawdę uczyć się JAVY

POWITANIE

Witaj!

Ten materiał powstał po to, aby uprościć naukę Javy i pomóc Ci zobaczyć cały temat krok po kroku.

JAK PRACOWAĆ Z MAPAMI

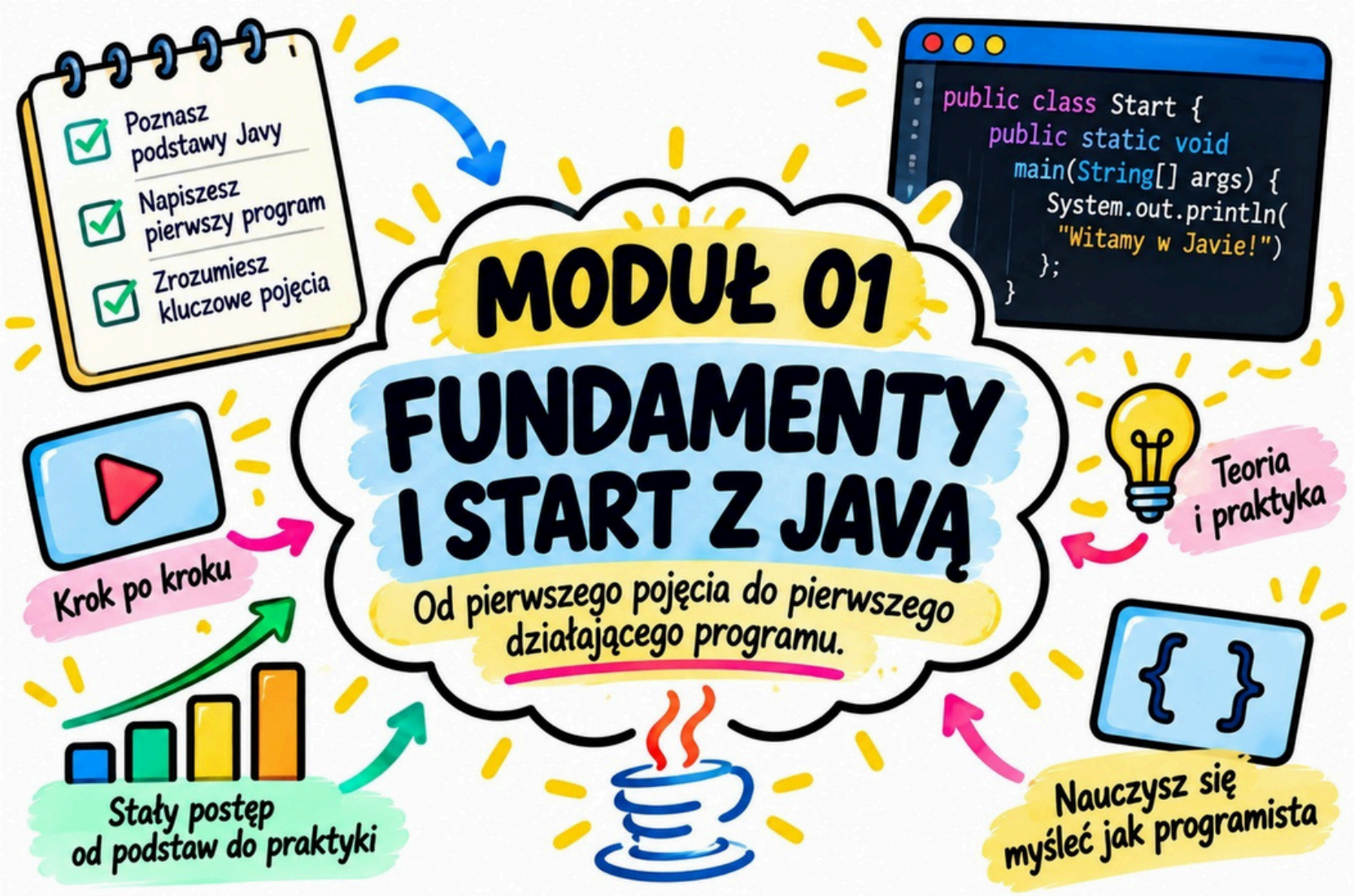
- czytaj po 1–3 mapy dziennie;
- przepisuj krótkie przykłady kodu;
- zatrzymuj się przy trudniejszych pojęciach;
- wracaj do map podczas ćwiczeń.

CZEGO SIĘ TU NAUCZYSZ

- składni i podstaw,
- logiki i pętli,
- metod i struktur danych,
- programowania obiektowego,
- pracy z wyjątkami i plikami,
- budowania prostych projektów.

WAŻNA ZASADA

Same mapy nie zastąpią praktyki. Najlepsze efekty daje czytanie + pisanie kodu + testowanie własnych rozwiązań.


Poznasz podstawy Javy
Napiszesz pierwszy program
Zrozumiesz kluczowe pojęcia
public class Start {
public static void
main(String[] args) {
System.out.println(
"Witamy w Javie!")
};
}
MODUŁ 01
FUNDAMENTY I START Z JAVĄ
Od pierwszego pojęcia do pierwszego działającego programu.
Krok po kroku
Teoria i praktyka
Stały postęp od podstaw do praktyki
Nauczysz się myśleć jak programista

JAVA

Write once, run anywhere.

CO TO JEST?

- język programowania wysokiego poziomu
- obiektowy i uniwersalny
- czytelny i popularny

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

GDZIE SIĘ UŻYWA?

- aplikacje backendowe
- systemy firmowe
- aplikacje desktopowe
- Android / mobile
- narzędzia edukacyjne i projekty

DLACZEGO WARTO?

- duża popularność
- dużo materiałów do nauki
- dobry wybór na start
- rozwija logiczne myślenie

CO DAJE NAUKA JAVY?

- rozumienie programowania
- budowanie projektów
- podstawa do dalszego rozwoju

JDK • JRE • JVM

Poznaj różnicę i zobacz, jak to działa razem.

JDK

zestaw dla programisty

- zawiera narzędzia do tworzenia kodu
- obejmuje kompilator i inne narzędzia

to, czego używasz do tworzenia



JRE

środowisko uruchomieniowe

- pozwala uruchamiać programy
- zawiera biblioteki potrzebne do działania

to, czego potrzebujesz do uruchamiania

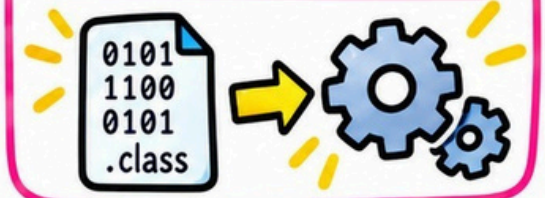


JVM

maszyna wirtualna Javy

- wykonuje kod bajtowy
- odpowiada za uruchomienie programu

silnik, który wykonuje kod



Prosty skrót:

JDK = tworzenie

JRE = uruchamianie

JVM = wykonywanie

JAK PRZYGOTOWAĆ ŚRODOWISKO DO NAUKI JAVY

KROK 1
ZAINSTALUJ JDK

- pobierz odpowiednią wersję
- zainstaluj na komputerze



KROK 2
WYBIERZ EDYTOR LUB IDE

- IntelliJ IDEA
- Eclipse
- VS Code

KROK 3
SPRAWDŹ INSTALACJĘ

- użyj komendy:

```
java -version
```

```
C:\> java -version
java version "21.0.1"
Java(TM) SE Runtime Environment
(build 21.0.1+12-LTS)
Java HotSpot(TM) 64-Bit Server VM
```

KROK 4
UTWÓRZ PIERWSZY PROJEKT

- nowy folder / nowy projekt
- plik Main.java



KROK 5
URUCHOM PROSTY KOD

- sprawdź, czy program działa poprawnie

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Cześć, Java!");
    }
}
```

NA START WYSTARCZY:

- JDK
- IDE
- chęć ćwiczenia
- kilka minut dziennie

BUDOWA PIERWSZEGO PROGRAMU W JAVIE

public class Main

- definicja klasy
- nazwa pliku powinna pasować do nazwy klasy

Main.java

```
public class Main {

    public static void main(String[] args) {

        System.out.println("Cześć, Java!");
    }
}
```

main(...)

- punkt startowy programu
- od tego miejsca Java zaczyna wykonywanie

System.out.println(...)

- wypisuje tekst na ekranie

Cześć, Java!

String[] args

- tablica argumentów przekazanych do programu

args[0]	args[1]	args[2]

CO ZAPAMIĘTAĆ?

- program ma klasę
- punkt startowy to `main`
- println służy do wyświetlania

ZMIENNE i TYPY DANYCH

1. CZYM JEST ZMIENNA?

- miejsce na dane
- ma nazwę i wartość
- można ją zmieniać



Zmienna to jak pudełko, w którym przechowujemy wartość.

2. JAK JĄ ZAPISAĆ?

- typ + nazwa + wartość
- przykład:

```
int wiek = 20;
```



Najpierw podajemy typ danych, potem nazwę zmiennej, a na końcu wartość.

3. TYPY PODSTAWOWE

`int`	liczby całkowite
`double`	liczby z przecinkiem
`char`	pojedynczy znak
`boolean`	prawda/fałsz

To najczęściej używane typy danych w Javie.

4. TYP TEKSTOWY

- `String` – tekst i napisy

```
String imie = "Java";
```



`String` służy do przechowywania tekstu (np. imion, nazw, komunikatów).

5. DOBRE ZASADY

- używaj czytelnych nazw
- nie mieszaj typów bez potrzeby



Dobre nazwy ułatwiają zrozumienie kodu Tobie i innym.

10. OPERATORY ARYTMETYCZNE

Licz i przetwarzaj wartości krok po kroku.

1. DO CZEGO SŁUŻĄ?

- wykonują działania na liczbach
- pomagają liczyć, porównywać i przetwarzać dane

Operatory arytmetyczne pozwalają wykonywać podstawowe działania matematyczne w programie.

2. PODSTAWOWE OPERATORY

+	dodawanie
-	odejmowanie
*	mnożenie
/	dzielenie
%	reszta z dzielenia

3. PRZYKŁADY

```
int suma = 2 + 3;
int roznica = 10 - 4;
int iloczyn = 3 * 5;
int iloraz = 8 / 2;
int reszta = 7 % 2;
```

4. NA CO UWAŻAĆ?

- dzielenie liczb całkowitych obcina część po przecinku
- typ danych wpływa na wynik

Aby otrzymać wynik z miejscami po przecinku, jedna z liczb musi być typu `float` lub `double`.

WARTO PAMIĘTAĆ!

`7 / 2` daje 3,
ale `7.0 / 2` daje 3.5

5. GDZIE SIĘ PRZYDAJĄ?

- liczenie punktów
- obliczanie cen
- przetwarzanie danych wejściowych

Operatory arytmetyczne używamy w wielu sytuacjach, np. w grach, sklepach czy analizie danych.

11. PORÓWNANIA I LOGIKA

Sprawdzaj warunki i podejmuj decyzje w programie.

1. OPERATORY PORÓWNANIA

==	równe
!=	różne
>	większe
<	mniejsze
>=	większe lub równe
<=	mniejsze lub równe



2. WYNIK PORÓWNANIA

- zawsze daje `true` albo `false`

Porównanie odpowiada na pytanie: czy warunek jest spełniony?

true

false

3. OPERATORY LOGICZNE

&&	i
||	lub
!	negacja



4. PRZYKŁADY

```
wiek >= 18
```

```
punkty > 50 && obecny == true
```

```
!(haslo.equals(""))
```

KIEDY UŻYWAMY?

- w `if`
- w pętlach
- przy sprawdzaniu warunków programu



12. INKREMENTACJA, DEKREMENTACJA I SKRÓCONY ZAPIS

Aktualizuj wartości szybciej i czytelniej.

1. INKREMENTACJA

Operator inkrementacji zwiększa wartość zmiennej o 1.

```
x++     // zwiększa o 1
++x     // też zwiększa o 1
```



2. DEKREMENTACJA

Operator dekrementacji zmniejsza wartość zmiennej o 1.

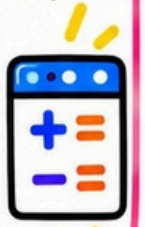
```
x--     // zmniejsza o 1
--x     // też zmniejsza o 1
```



3. SKRÓCONY ZAPIS DZIAŁAŃ

Możemy używać skróconego zapisu, aby szybciej zmieniać wartość zmiennej.

```
x += 5;  // to samo co x = x + 5
x -= 2;  // to samo co x = x - 2
x *= 3;  // to samo co x = x * 3
x /= 2;  // to samo co x = x / 2
```



4. DLACZEGO TO WAŻNE?

Inkrementacja, dekrementacja i skrócony zapis to proste narzędzia, które bardzo ułatwiają pisanie kodu.

- Kod jest krótszy.
- Łatwiej aktualizować zmienne.
- Bardzo często używane w pętlach.



5. PROSTE PRZYKŁADY

Zobaczmy, jak to działa w praktyce.

```
int licznik = 0;
licznik++;
licznik += 10;
licznik--;
```





Najpierw zrozum zwykły zapis, potem używaj skrótów świadomie.

13. KONWERSJA TYPÓW

Zmieniaj typ danych świadomie i bezpiecznie.

1. PO CO KONWERTOWAĆ?

- Gdy chcemy użyć danych w innym formacie.
- Gdy łączymy różne typy w obliczeniach.

Konwersja pozwala nam używać danych w odpowiednim typie do danej sytuacji.

2. KONWERSJA NIEJAWNA

Mniejszy typ do większego.
Np. `int` do `double`.

`int` (4 bajty) → `double` (8 bajtów)

```
int a = 5;
double b = a;  // konwersja niejawna
```

Jest wykonywana automatycznie przez kompilator.

3. KONWERSJA JAWNA

Trzeba wpisać typ ręcznie.
Np. `(int) 3.99`.

`double` (8 bajtów) → `int` (4 bajty)

```
double cena = 9.99;
int pelna = (int) cena;  // konwersja jawna
```

Musimy jawnie wskazać, na jaki typ konwertujemy.

4. PRZYKŁADY

Kilka przykładów konwersji typów w praktyce.

```
int a = 5;
double b = a;       // konwersja niejawna

double cena = 9.99;
int pelna = (int) cena;  // konwersja jawna
```

W pierwszym przykładzie `int` jest automatycznie zamieniany na `double`. W drugim `(int)` usuwa część po przecinku.

5. UWAGA NA UTRATĘ DANYCH

- Przy rzutowaniu można stracić część po przecinku.
- Nie każda konwersja jest bezpieczna (np. duża liczba do mniejszego typu).

```
double liczba = 3.99;
int wynik = (int) liczba;  // wynik = 3
```

3.99 → 3

Po konwersji z `double` na `int` część po przecinku zostaje odcięta (nie zaokrąglana).

PODSUMOWANIE

- Niejawna = zwykle bezpieczna.
- Jawna = kontrolowana przez programistę.

14. STAŁE W JAVIE

Chroń wartości, które nie powinny się zmieniać.

1. CO TO JEST STAŁA?

Wartość, której nie chcemy zmieniać.

Stała to wartość, która pozostaje taka sama podczas działania programu.

2. JAK JĄ TWORZYMY?

Używamy słowa kluczowego `final`

Słowo `final` informuje kompilator, że wartość tej zmiennej nie powinna się zmieniać.

3. PRZYKŁAD

Przykłady stałych w Javie:

```
final double PI = 3.14159;
final int DNI_W_TYGODNIU = 7;
```

PI = 3.14159

DNI_W_TYGODNIU = 7

4. DLACZEGO WARTO?

- Kod jest czytelniejszy.
- Mniej błędów.
- Jasne zasady działania programu.

5. DOBRE PRAKTYKI

- Nazwy stałych wielkimi literami.
- Używaj, gdy wartość ma pozostać niezmienna.

MAX_ROZMIAR

ZMIENNA vs STAŁA

Zmienna: może się zmieniać.

Stała: nie powinna się zmieniać.

15. KOMENTARZE I CZYTELNOŚĆ KODU

Pisz kod, który jest zrozumiały także po czasie.

1. PO CO SĄ KOMENTARZE?

- Wyjaśniają intencję kodu.
- Ułatwiają naukę i zrozumienie.
- Pomagają wrócić do kodu po czasie.



2. RODZAJE KOMENTARZY

Komentarz jednoliniowy

```
// treść komentarza
```

Komentarz wieloliniowy

```
/*
   treść komentarza
   w kilku liniach
*/
```



3. PRZYKŁADY

```
// obliczamy sumę
int suma = a + b;

/* ten fragment
   wyświetla wynik */
System.out.println(suma);
```



4. DOBRA CZYTELNOŚĆ KODU

- Sensowne nazwy zmiennych.
- Odstępy i wcięcia (poprawne formatowanie).
- Prosty styl zapisu.

```
// dobre nazwy
int liczbaUzytkownikow;
int sumaWynikow;

if (suma > 0) {
    System.out.println(
      "Wynik: " + suma);
}
```



5. UWAGA

- Komentarze nie zastąpią czytelnego kodu.
- Nie komentuj rzeczy oczywistych bez potrzeby.



Najlepszy kod to taki, który jest zrozumiały nawet bez długich wyjaśnień.

16. SCANNER

Pobieraj dane od użytkownika i twórz interaktywne programy.

1. DO CZEGO SŁUŻY?

- pobiera dane od użytkownika
- pozwala wpisywać tekst, liczby i inne wartości



```
Wpisz dane:
```

2. CO TRZEBA ZROBIĆ?

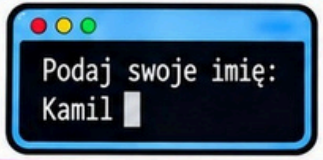
- zaimportować klasę
- utworzyć obiekt Scanner




```
import java.util.Scanner;
Scanner sc = new Scanner(System.in);
```

3. PODSTAWOWY PRZYKŁAD

```
import java.util.Scanner;
Scanner sc = new Scanner(System.in);
String imie = sc.nextLine();
```



4. CZĘSTE METODY

nextLine()	tekst
nextInt()	liczba całkowita
nextDouble()	liczba zmiennoprzecinkowa

5. GDZIE SIĘ PRZYDAJE?

- quizy
- kalkulatory
- proste formularze
- programy ćwiczeniowe



PAMIĘTAJ:

- Po pobieraniu danych warto jasno komunikować użytkownikowi, co ma wpisać.



17. WYŚWIETLANIE DANYCH

Łącz tekst z wartościami i komunikuj się z użytkownikiem.

1. PODSTAWOWA METODA

```
System.out.println()
```

służy do wyświetlania informacji.



2. TEKST I ZMIENNE RAZEM

Możemy łączyć napisy z wartościami.

```
String imie = "Anna";
int wiek = 20;
System.out.println("Cześć, jestem " + imie);
```

Cześć, jestem Anna!

3. PRZYKŁADY

Proste przykłady łączenia tekstu z wartościami.

```
String imie = "Anna";
int wiek = 20;
System.out.println("Mam na imię " + imie);
System.out.println("Mój wiek to " + wiek);
```

```
Mam na imię Anna
Mój wiek to 20
```

4. OPERATOR + PRZY TEKSTACH



- łączy napisy
- pozwala budować komunikaty

```
String tekst = "Wynik to: " + 10;
System.out.println(tekst);
```

```
Wynik to: 10
```

5. DOBRE PRAKTYKI

- Komunikaty powinny być jasne.
- Użytkownik ma wiedzieć, co się dzieje.





Program komunikuje się z użytkownikiem przez **dane wejściowe i dane wyjściowe.**

18. NAJCZĘSTSZE BŁĘDY POCZĄTKUJĄCYCH W JAVIE

Rozpoznaj typowe problemy i ucz się szybciej.

1. BRAK ŚREDNIKA

Java często wymaga ; na końcu instrukcji.

```
int x = 10
Błąd kompilacji!
```

```
int x = 10;
```

2. LITERÓWKI W NAZWACH

- System to nie system.
- Wielkość liter ma znaczenie w Javie.

```
system.out.println("Hello");
// błąd - nie istnieje "system"
```

```
System.out.println("Hello");
// poprawnie
```

Java rozróżnia wielkość liter!

3. NIEZGODNOŚĆ TYPU DANYCH

Nie można przypisać wartości innego typu danych.

```
int liczba = "10";
// błąd - tekstu nie można przypisać do int
```

```
int liczba = 10;
// poprawnie
```

Typy danych muszą się zgadzać!

4. BŁĘDY W NAWIASACH I KLAMRACH

Brak } albo) psuje strukturę programu.

```
public class Main {
   public static void main(String[] args) {
      System.out.println("Witaj");
   } / brak zamknięcia klamry
}
```

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

5. ZŁA NAZWA PLIKU I KLASY

Jeśli klasa jest publiczna, jej nazwa powinna pasować do nazwy pliku.

plik: Program.java

```
public class Main {
   // ...
}
```

// błąd - nazwa pliku i klasy się różnią

plik: Main.java

```
public class Main {
   // ...
}
```

// poprawnie - nazwy są takie same

Main.java

```
public class Main {
   ...
}
```

6. CZYTANIE KOMUNIKATÓW BŁĘDÓW

- Nie ignoruj komunikatów błędów.
- Szukaj linii i wskazówki, co jest nie tak.

```
Main.java:3: error: ';' expected
     System.out.println("Witaj")
                               ^
1 error
```

Komunikat mówi, w którym pliku i w której linii jest błąd. i co się stało.

DOBRA WIADOMOŚĆ:

Błędy są normalne. Każdy programista uczy się przez poprawianie kodu.

Z każdą poprawką jesteś bliżej lepszego programisty!

19. START Z JAVĄ

Podsumowanie modułu 01

1. JAVA

- popularny język programowania
- dobry do nauki i praktyki

Java

2. JDK / JRE / JVM

- tworzenie
- uruchamianie
- wykonywanie

JDK → JRE → JVM

3. ŚRODOWISKO PRACY

- JDK
- IDE
- pierwszy projekt

4. STRUKTURA PROGRAMU

- klasa
- main
- println

```
public class Main {
    public static void
      main(String[] args) {
        System.out.println(
          "Witaj, Java!");
    }
}
```

5. ZMIENNE I TYPY

- int, double, char, boolean, String

int, double, char, boolean, String

6. OPERATORY

- arytmetyczne
- porównania
- logiczne

+ − × ÷
== != < >
&& || !

7. PRACA Z DANYMI

- Scanner
- wyświetlanie wyników

Scanner

Wynik: 123

8. DOBRE NAWYKI

- czytelność kodu
- komentarze
- analiza błędów

CO DALEJ?

W następnym module przejdziesz do logiki, instrukcji warunkowych i pętli.

MODUŁ 02

LOGIKA, WARUNKI I PĘTLE

Podejmuj decyzje w kodzie i automatyzuj powtarzalne zadania.



21. IF

Instrukcja if — wykonaj kod, gdy warunek jest spełniony.

1. DO CZEGO SŁUŻY?

- sprawdza warunek;
- wykonuje kod tylko wtedy, gdy warunek daje `true`;
- pozwala programowi podejmować decyzje.



2. PODSTAWOWA SKŁADNIA

```
if (warunek) {
    // kod
}
```



3. PROSTY PRZYKŁAD

```
int wiek = 20;

if (wiek >= 18) {
    System.out.println("Jesteś
      pełnoletni");
}
```



4. CO DZIEJE SIĘ KROK PO KROKU?



5. GDZIE SIĘ PRZYDAJE?

- sprawdzanie wieku;
- sprawdzanie wyniku;
- walidacja danych;
- podejmowanie decyzji w programie.





if = **JEŚLI** warunek jest prawdziwy, wykonaj instrukcję.

WARUNEK

TRUE → IF
FALSE → ELSE

1. SKŁADNIA

```
if (warunek) {
    // opcja A
} else {
    // opcja B
}
```

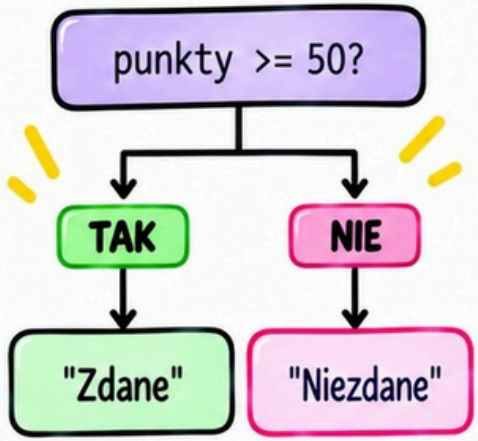
2. PRZYKŁAD

```
int punkty = 45;

if (punkty >= 50) {
    System.out.println("Zdane");
} else {
    System.out.println("Niezdane");
}
```

A+

3. JAK TO DZIAŁA?



4. KIEDY UŻYWAĆ?

Kiedy istnieją dokładnie dwie możliwości główne.

- zalogowany / niezalogowany;
- pełnoletni / niepełnoletni;
- dostęp / brak dostępu;
- poprawna / błędna odpowiedź.



5. WAŻNE

Wykona się tylko jeden z dwóch bloków.

`if` = pierwsza możliwość

`else` = wszystko inne

23. ELSE IF

gdy masz więcej niż dwie możliwości

1. IDEA

`else if` pozwala sprawdzić kolejny warunek, jeśli wcześniejszy nie został spełniony.

2. SKŁADNIA

```
if (warunek1) {
    // opcja 1
} else if (warunek2) {
    // opcja 2
} else {
    // opcja 3
}
```

`else if` może występować wielokrotnie.

3. PRZYKŁAD OCEN

```
int wynik = 82;

if (wynik >= 90) {
    System.out.println("Bardzo dobrze");
} else if (wynik >= 70) {
    System.out.println("Dobrze");
} else if (wynik >= 50) {
    System.out.println("Zdane");
} else {
    System.out.println("Niezdane");
}
```

A+

4. JAK JAVA SPRAWDZA?

wynik >= 90? NIE
wynik >= 70? NIE
TAK
wynik >= 50? NIE
"Dobrze"
koniec sprawdzania

Java zatrzymuje się przy pierwszym pasującym warunku.

5. DOBRA PRAKTYKA

- układaj warunki logicznie;
- najbardziej szczegółowe przypadki często sprawdzaj wcześniej;
- unikaj zbędnych warunków.

Przejdź dalej, aby zobaczyć kolejne konstrukcje sterujące w Javie.

24. ZAGNIEŻDŻONE `IF`

Warunek wewnątrz warunku

1. CO OZNACZA ZAGNIEŻDŻENIE?

- jeden `if` znajduje się wewnątrz drugiego;
- drugi warunek sprawdzamy dopiero po spełnieniu pierwszego.

```
if (warunek1) {
    if (warunek2) {
        ...
    }
}
```

2. PRZYKŁAD

```
int wiek = 22;
boolean maBilet = true;

if (wiek >= 18) {
   if (maBilet) {
      System.out.println("Możesz wejść");
   }
}
```

BILET

3. LOGIKA



4. KIEDY UŻYWAĆ?

- warunki zależą od siebie;
- drugi test ma sens tylko po pierwszym.

5. UWAGA

Zbyt wiele zagnieżdżeń = kod trudny do czytania.

Jeśli logika robi się zbyt głęboka, rozważ uproszczenie warunków lub podział kodu na metody.

OPERATOR TERNARNY

`?:`

krótka decyzja w jednej linii

1. CO ROBI?

Wybiera jedną wartość w zależności od tego, czy warunek jest prawdziwy czy fałszywy.



To skrócona forma prostego wyrażenia warunkowego.

2. SCHEMAT

```
warunek ? wartośćJeśliTrue : wartośćJeśliFalse
```

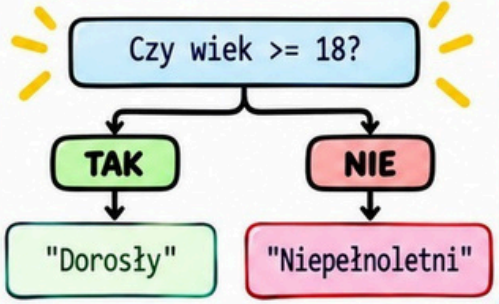
Jeśli warunek jest prawdziwy, zwracana jest pierwsza wartość, w przeciwnym razie druga.

3. PRZYKŁAD

```
int wiek = 20;
String status =
    wiek >= 18 ? "Dorosły" :
    "Niepełnoletni";
```

Zmienna status przyjmuje jedną z dwóch wartości w zależności od wieku.

4. CZYTAJ TO TAK



5. KIEDY WARTO UŻYWAĆ?

- bardzo prosta decyzja;
- przypisanie jednej z dwóch wartości;
- krótki i czytelny warunek.



Świetnie sprawdza się, gdy warunek jest prosty i nie wymaga wielu operacji.

6. KIEDY NIE UŻYWAĆ?

- gdy logika jest długa;
- gdy masz wiele warunków;
- gdy zapis staje się trudny do czytania.



Wtedy lepszym rozwiązaniem będzie klasyczne użycie if / else.

PORÓWNANIE



Proste → `?:`

```
int max = a > b ? a : b;
```

Bardziej rozbudowane → if / else

```
int max;
if (a > b) {
    max = a;
} else {
}   max = b;
```



SWITCH

Instrukcja switch – wybór jednej z wielu opcji.

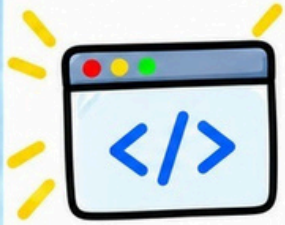
1. KIEDY UŻYWAĆ?

Gdy porównujemy jedną zmienną z wieloma konkretnymi wartościami.



2. PODSTAWOWA SKŁADNIA

```
switch (wartosc) {
    case 1:
        // kod
        break;
    case 2:
        // kod
        break;
    default:
        // kod
}
```



3. PRZYKŁAD

```
int dzien = 2;

switch (dzien) {
   case 1:
      System.out.println("Poniedziałek");
      break;
   case 2:
      System.out.println("Wtorek");
      break;
   default:
      System.out.println("Inny dzień");
}
```



4. ELEMENTY

- `switch` – wartość do sprawdzenia;
- `case` – możliwa opcja;
- `break` – kończy dany przypadek;
- `default` – opcja domyślna.

5. KIEDY JEST CZYTELNIEJSZY OD if?

Gdy masz wiele prostych porównań tej samej zmiennej.





Jedna wartość → wiele konkretnych możliwości.



27. SWITCH czy IF?

Dobierz narzędzie do problemu.

IF / ELSE

Najlepsze, gdy:

- sprawdzasz zakresy;
- masz złożone warunki;
- używasz &&, ||, <, >;
- warunki są różnego typu.



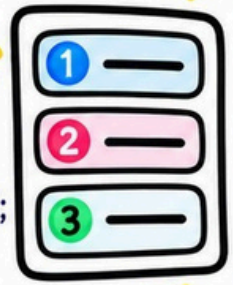
```
if (wiek >= 18 && maBilet) {
    ...
}
```

Możesz łączyć wiele warunków, sprawdzać zakresy i używać operatorów logicznych.

SWITCH

Najlepszy, gdy:

- jedna wartość ma wiele konkretnych opcji;
- porównujesz stałe przypadki;
- chcesz uniknąć długiego łańcucha `else if`.




```
switch (opcja) {
      case 1:
          ...
}
```

Idealny, gdy porównujesz jedną zmienną z wieloma stałymi wartościami (konkretne opcje).

SZYBKA ZASADA



zakres / logika → **if**

konkretne wartości → **switch**



FOR

Pętla for — powtarzaj kod określoną liczbę razy.

1. DO CZEGO SŁUŻY?

- powtarza kod;
- świetna, gdy znamy liczbę powtorzeń;
- często używana przy tablicach i listach.



2. SKŁADNIA

```
for (int i = 0; i < 5; i++) {
    System.out.println(i);
}
```

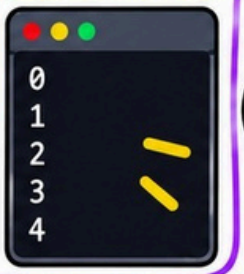
3. TRZY CZĘŚCI



4. CO ZOSTANIE WYPISANE?

```
for (int i = 0; i < 5; i++) {
    System.out.println(i);
}
```

Wynik w konsoli:



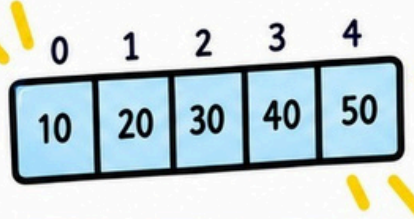
5. JAK DZIAŁA?



6. GDZIE UŻYWAĆ?

- powtórzenia;
- indeksy;
- tablice;
- przetwarzanie danych.





for jest świetny, gdy wiesz, ile razy chcesz coś wykonać.

29. WHILE

Pętla `while` — powtarzaj tak długo, jak warunek jest prawdziwy.

1. JAK DZIAŁA?

- Najpierw sprawdza warunek.
- Jeśli jest `true`, wykonuje blok.
- Potem wraca i sprawdza ponownie.

2. SKŁADNIA

```
while (warunek) {
    // kod
}
```

3. PRZYKŁAD

```
int licznik = 0;

while (licznik < 5) {
   System.out.println(licznik);
   licznik++;
}
```

0 1 2 3 4

4. SCHEMAT DZIAŁANIA



5. KIEDY UŻYWAĆ?

- gdy liczba powtórzeń nie jest znana;
- gdy czekamy na określony stan;
- gdy powtarzamy działanie do momentu spełnienia warunku.

6. UWAŻAJ NA PĘTLĘ NIESKOŃCZONĄ

Jeśli warunek nigdy nie stanie się fałszywy, pętla nie zakończy się.

```
while (true) {
    System.out.println("bez końca");
}
```

PORÓWNANIE Z PĘTLĄ FOR

`for`	→ znasz liczbę powtórzeń
`while`	→ znasz warunek zakończenia

30. DO-WHILE

Pętla do-while – wykonaj kod przynajmniej raz.

1. JAK DZIAŁA?

- Najpierw wykonuje kod;
- dopiero potem sprawdza warunek;
- dlatego wykona się przynajmniej jeden raz.

2. SKŁADNIA

```
do {
    // kod
} while (warunek);
```

3. PRZYKŁAD

```
int licznik = 0;

do {
    System.out.println(licznik);
    licznik++;
} while (licznik < 5);
```

4. KOLEJNOŚĆ DZIAŁANIA



5. WHILE KONTRA DO-WHILE

while

najpierw sprawdza warunek



do-while

najpierw wykonuje kod



6. GDZIE SIĘ PRZYDAJE?

- menu użytkownika;
- pytanie powtarzane do poprawnej odpowiedzi;
- operacja, która musi wystąpić przynajmniej raz.

PODSUMOWANIE

do-while = minimum jedno wykonanie.

31. BREAK

BREAK = wyjdź z pętli

1. CO ROBI?

- Natychmiast kończy pętlę.
- Program przechodzi do kodu znajdującego się po niej.

2. PRZYKŁAD

```
for (int i = 0; i < 10; i++) {
    if (i == 5) {
        break;
    }
    System.out.println(i);
}
```

3. WYNIK

```
0
1
2
3
4
```

4. DLACZEGO?

i == 5

↓

break

↓

pętla zostaje zakończona

5. TYPOWE ZASTOSOWANIA

- Znaleziono szukany element.
- Użytkownik wybrał wyjście.
- Dalsze wykonywanie nie ma sensu.
- Osiągnięto określony stan.

6. WAŻNE

break nie oznacza „pular uma volta”.
Ele termina a repetição inteira.

STOP

BREAK = WYJDŹ Z PĘTLI

32. CONTINUE

Continue pomija jedną iterację, ale pętla trwa dalej.

1. CO ROBI?

- Przerywa tylko aktualną iterację.
- Pętla nadal działa i wykonuje kolejne iteracje.



2. PRZYKŁAD

```
for (int i = 0; i < 5; i++) {
    if (i == 2) {
        continue;
    }
    System.out.println(i);
}
```

3. WYNIK

```
0
1
3
4
```

4. CO SIĘ STAŁO Z 2?



5. BREAK KONTRA CONTINUE

BREAK	CONTINUE
Kończy całą pętlę.	Pomija tylko jedną iterację.
break;	continue;
KONIEC PĘTLI	NASTĘPNA ITERACJA

6. KIEDY UŻYWAĆ?

- Ignorowanie niechcianych danych (np. pustych lub nieprawidłowych).
- Pomijanie wyjątkowego przypadku (np. konkretnej wartości).
- Filtrowanie elementów podczas iteracji (np. wypisywanie tylko wybranych).



PODSUMOWANIE

CONTINUE = POMIŃ I IDŹ DALEJ



33. PĘTLE ZAGNIEŻDŻONE

Pętla zewnętrzna może zawierać pętlę wewnętrzną.

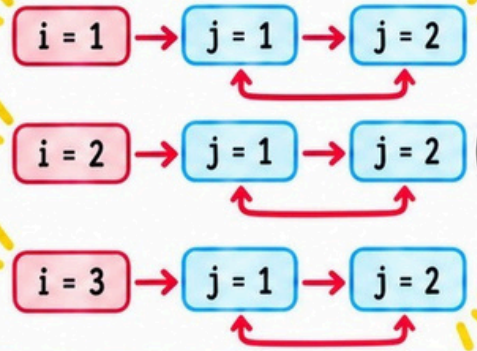
1. CO TO ZNACZY?

- Pętla zewnętrzna zawiera pętlę wewnętrzną.
- Kod wewnętrznej pętli wykonuje się wielokrotnie dla każdej iteracji pętli zewnętrznej.

2. PRZYKŁAD

```
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 2; j++) {
        System.out.println(i + " - " + j);
    }
}
```

3. JAK DZIAŁA?



4. GDZIE SIĘ PRZYDAJĄ?

- praca z tablicami 2D (macierzami);
- tworzenie tabel;
- obsługa siatek (np. w grach);
- kombinacje elementów;
- generowanie wzorów (np. gwiazdki, kwadraty).



5. WAŻNE

- Każda iteracja pętli zewnętrznej powoduje wykonanie wszystkich iteracji pętli wewnętrznej.
- Każdy dodatkowy poziom zagnieżdżenia zwiększa liczbę wykonań kodu.

Obliczanie liczby wykonań:

3 iteracje × 2 iteracje = **6 wykonań** wewnętrznego kodu.

34. NAJCZĘSTSZE BŁĘDY

Zobacz typowe pułapki w warunkach i pętlach.

1. `=` ZAMIAST `==`

`=` przypisuje wartość;
`==` porównuje wartości.

```
int a = 5;
if (a = 5) {      // BŁĄD!
    System.out.println("OK");
}
```

2. ZŁY OPERATOR LOGICZNY

&& gdy obie rzeczy muszą być prawdziwe (logiczne AND).

|| gdy wystarczy jedna (logiczne OR).

```
if (wiek > 18 && maBilet) {
    System.out.println("Możesz wejść");
}
```

3. PĘTLA NIESKOŃCZONA

```
int i = 0;
while (i < 5) {
    System.out.println(i);
    // brak zmiany i
}
```

i nigdy się nie zmienia,
dlatego warunek zawsze jest prawdziwy
i pętla nie kończy się.

4. ZŁY WARUNEK KOŃCOWY

```
int[] tablica = {10, 20, 30};
for (int i = 0; i <= tablica.length; i++) {
    System.out.println(tablica[i]);
}
```

i <= tablica.length może spowodować próbę dostępu do nieistniejącej pozycji.
Poprawnie powinno być i < tablica.length.

5. NIEPOTRZEBNE ZAGNIEŻDŻENIA

```
if (wiek > 18) {
    if (maBilet) {
        if (miejsceWolne) {
            System.out.println("Możesz wejść");
        }
    }
}
```

Wiele zagnieżdżonych if utrudnia czytanie kodu i zwiększa ryzyko błędów.
Często można to uprościć, łącząc warunki.

6. BRAK KLAMER LUB ZŁE WCIĘCIA

```
if (wynik > 0)
    System.out.println("Pozytywny");
    System.out.println("Koniec");
```

Tylko pierwsza linia należy do if.
Zła struktura może ukrywać błędy.

Poprawnie używaj klamer, nawet przy jednej linijce:

```
if (wynik > 0) {
    System.out.println("Pozytywny");
    System.out.println("Koniec");
}
```

GDY KOD ZACHOWUJE SIĘ DZIWNIE:

1. sprawdź warunek – czy jest poprawny;
2. sprawdź wartości zmiennych – czy mają oczekiwane wartości;
3. sprawdź, czy pętla je zmienia – czy zmienne aktualizują się;
4. prześledź program krok po kroku – użyj wydruków lub debuggera.

35. STEROWANIE PROGRAMEM

Podsumowanie modułu 02

1. if

Jedna decyzja oparta na warunku.

```
if (wiek >= 18) {
    System.out.println("Pełnoletni");
}
```

2. if / else

Dwie możliwe drogi.

```
if (punkty >= 50) {
    System.out.println("Zdał");
} else {
    System.out.println("Nie zdał");
}
```

3. else if

Kilka warunków sprawdzanych po kolei.

```
if (ocena >= 5) {
    System.out.println("Bardzo dobrze");
} else if (ocena >= 4) {
    System.out.println("Dobrze");
} else {
    System.out.println("Słabiej");
}
```

4. switch

Jedna wartość + wiele konkretnych przypadków.

```
switch (dzien) {
  case 1:
     System.out.println("Poniedziałek");
     break;
  case 2:
     System.out.println("Wtorek");
     break;
  default:
     System.out.println("Inny dzień");
}
```

5. for

Gdy zazwyczaj znamy liczbę powtórzeń.

```
for (int i = 0; i < 5; i++) {
    System.out.println("i = " + i);
}
```

6. while

Gdy powtarzamy kod, dopóki warunek jest prawdziwy.

```
int licznik = 0;
while (licznik < 5) {
     System.out.println(licznik);
     licznik++;
}
```

7. do-while

Gdy kod musi wykonać się przynajmniej raz.

```
int licznik = 0;
do {
     System.out.println(licznik);
     licznik++;
} while (licznik < 5);
```

8. break / continue

`break` → zakończ;
`continue` → pomiń iterację.

```
for (int i = 0; i < 10; i++) {
     if (i == 5) continue;  // pomiń 5
     if (i == 8) break;     // zakończ pętlę
}    System.out.println(i);
```

CHCESZ PODJĄĆ DECYZJĘ? → if / switch

CHCESZ POWTARZAĆ KOD? → for / while / do-while

CHCESZ KONTROLOWAĆ PĘTLĘ? → break / continue

Teraz potrafisz sterować przepływem programu. Następny krok: dzielenie kodu na metody.

MODUŁ 03

METODY, PARAMETRY I ORGANIZACJA KODU

Dziel duże problemy na małe, czytelne i wielokrotnego użytku fragmenty.

METODA

PARAMETRY

Metoda może przyjmować parametry wewnątrz nawiasów.

```
void pokaz(int liczba,
     String napis) {
     // kod
}
```

ARGUMENTY

Podczas wywołania metody przekazujemy argumenty.

```
pokaz(5, "Java");
```

RETURN

Metoda może zwracać wartość za pomocą słowa return.

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

TYP ZWRACANY

Każda metoda ma określony typ zwracany: int, String, void lub inny typ.

```
double srednia(int a, int b) {
    return (a + b) / 2.0;
}
```

PRZECIĄŻANIE

Można tworzyć metody o tej samej nazwie, ale z różnymi parametrami.

```
void pokaz() { ... }
void pokaz(int x) { ... }
void pokaz(String s) { ... }
```

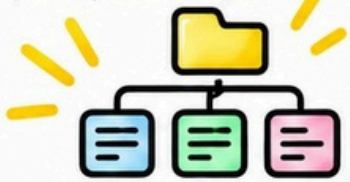
ZAKRES ZMIENNYCH

Zmienne zdefiniowane wewnątrz metody są widoczne tylko w jej zakresie.

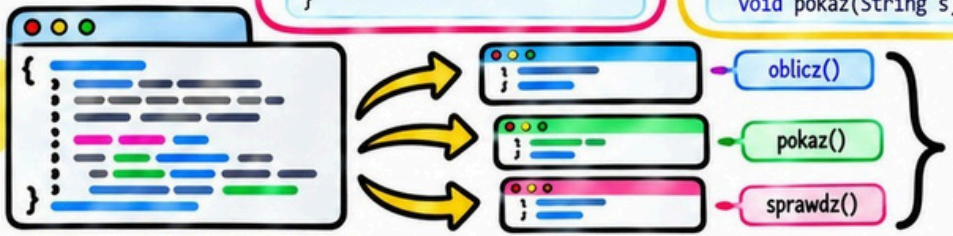
```
void oblicz() {
    int liczba = 10;
    // widoczna tylko tutaj
}
```

ORGANIZACJA KODU

Dzielimy kod na małe, logiczne metody, co ułatwia czytanie, rozwijanie i ponowne użycie.



Jeden duży blok kodu...



Mniej powtarzania. Więcej porządku.

37. METODA

Metoda to nazwany fragment kodu wykonujący konkretne zadanie.

1. CO TO JEST?

- Wydzielony fragment kodu.
- Wykonuje konkretne zadanie.
- Można go wywoływać wielokrotnie.



2. PROSTY PRZYKŁAD

```
static void przywitaj() {
   System.out.println("Cześć!");
}
```

3. JAK URUCHOMIĆ METODĘ?

```
// wywołanie metody
przywitaj();
```

4. DLACZEGO UŻYWAMY METOD?

- Mniej powtarzania kodu.
- Czytelniejszy kod.
- Łatwiejsze testowanie.
- Prostsze poprawianie programu.





5. DOBRA METODA

Powinna mieć jedno jasne zadanie.

Przykłady dobrych nazw metod:

```
obliczSume()
wyswietlMenu()
sprawdzHaslo()
```

PODSUMOWANIE



Metoda = nazwany fragment kodu wykonujący konkretne zadanie.

38. Z CZEGO SKŁADA SIĘ METODA?

1. static

Na fazie obecnej nauki pozwala wywołać metodę w obecnym kontekście bez tworzenia obiektu.

2. int

Typ wartości zwracanej.

3. dodaj

Nazwa metody.

4. int a, int b

Parametry.
Tu są dwa parametry typu int: a i b.

5. { ... }

Ciało metody.
Zawiera instrukcje, które zostaną wykonane po wywołaniu metody.

6. return a + b

Wartość zwracana przez metodę.
Oblicza sumę parametrów i zwraca wynik.

```
static int dodaj(int a, int b) {
    return a + b;
}
```

7. SCHEMAT BUDOWY METODY

MODYFIKATORY	+	TYP	+	NAZWA	+	PARAMETRY	+	CIAŁO
np. static		np. int		np. dodaj		np. (int a, int b)		np. { ... }

8. PRZYKŁAD UŻYCIA METODY

```
int wynik = dodaj(4, 6);
```

WYNIK DZIAŁANIA

wynik = 10

9. WAŻNA WSKAZÓWKA

Czytelna nazwa metody powinna mówić, co metoda robi.

39. PARAMETRY I ARGUMENTY

Poznaj różnicę między parametrem a argumentem.

1. PARAMETR

- To zmienna zadeklarowana w definicji metody.
- Określa, jakie dane metoda będzie otrzymywać.

```
static void przywitaj(String imie) {
  System.out.println("Cześć" + imie);
}
```

String imie = parametr

2. ARGUMENT

- To rzeczywista wartość przekazana przy wywołaniu metody.
- Może to być liczba, tekst, zmienna, wyrażenie itp.

```
przywitaj("Anna");
```

"Anna" = argument

ARGUMENT "Anna" → przekazujemy do → PARAMETRU `imie` → metoda korzysta z wartości

3. WIĘCEJ PARAMETRÓW

- Metoda może mieć wiele parametrów.
- Przy wywołaniu musimy podać tyle samo argumentów w tej samej kolejności.

```
static int dodaj(int a, int b) {
   return a + b;
}
```

Wywołanie metody:

```
dodaj(5, 3);
```

5 → a

3 → b

4. ZAPAMIĘTAJ:

- **parametr** → nazwa w definicji metody;
- **argument** → konkretna wartość przy wywołaniu.

40. RETURN

`return` — zwróć wynik z metody

1. CO ROBI `return`?

- kończy działanie metody;
- zwraca wartość do miejsca, z którego metoda została wywołana.

2. PRZYKŁAD

Metoda, która dodaje dwie liczby i zwraca ich sumę:

```
static int dodaj(int a, int b) {
    return a + b;
}
```

3. WYWOŁANIE

Wywołujemy metodę i zapisujemy zwróconą wartość w zmiennej:

```
int wynik = dodaj(5, 3);
```

5 + 3 → metoda oblicza → return 8 → wynik = 8

4. TYP ZWRACANY MUSI SIĘ ZGADZAĆ

Jeśli metoda ma zadeklarowany typ zwracany, np. `static int ...`, to musi zwrócić wartość tego typu (w tym przypadku `int`).

```
static String pobierzImie() {
    return "Anna";
}
```

Zwracana wartość musi być zgodna z typem określonym w nagłówku metody.

5. `void` KONTRA WARTOŚĆ ZWRACANA

void	metoda niczego nie zwraca.
int, double, String, itd.	metoda zwraca określony typ danych.

PODSUMOWANIE

`return` pozwala wykorzystać wynik metody w dalszej części programu.

41. VOID

Metody `void` – wykonaj działanie bez zwracania wyniku

1. CO OZNACZA `void` ?

- metoda nie zwraca wartości;
- wykonuje określone zadanie.

Słowo kluczowe `void` oznacza, że metoda nie zwraca żadnej wartości, ale może wykonywać różne operacje.

2. PRZYKŁAD

```
static void pokazKomunikat() {
    System.out.println("Witaj!");
}
```

Metoda `pokazKomunikat()` nie zwraca wartości, ale wyświetla tekst w konsoli.

3. WYWOŁANIE

```
pokazKomunikat();
```

Aby wykonać metodę typu `void`, wystarczy ją wywołać — tak jak każdą inną metodę.

4. TYPOWE ZASTOSOWANIA

- wyświetlanie informacji;
- zmiana stanu programu;
- zapisywanie danych;
- wykonywanie operacji pomocniczych.

5. `void` KONTRA METODA ZWRACAJĄCA

VOID

```
static void powitaj() {
    System.out.println("Cześć");
}
```

→ wykonuje akcję

Metoda typu `void` nie zwraca wartości, ale wykonuje określone działanie.

ZWROT WARTOŚCI

```
static int pobierzWiek() {
    return 20;
}
```

→ daje wynik

Metoda zwracająca wartość przekazuje wynik do miejsca, w którym została wywołana.

PODSUMOWANIE

Jeśli potrzebujesz wyniku — zwracaj wartość.

Jeśli potrzebujesz tylko akcji — `void` może wystarczyć.

42. METHOD OVERLOADING

Przeciążanie metod – ta sama nazwa, różne parametry

1. NA CZYM POLEGA?

Kilka metod może mieć tę samą nazwę, jeśli różnią się parametrami.

2. PRZYKŁAD

```
static int dodaj(int a, int b) {
    return a + b;
}
```

Dodaje dwie liczby całkowite.

```
static double dodaj(double a, double b) {
    return a + b;
}
```

Dodaje dwie liczby zmiennoprzecinkowe.

3. CO WYBIERA JAVA?

Java na podstawie sygnatury metody wybiera odpowiednią wersję, biorąc pod uwagę:

- liczbę argumentów;
- typy argumentów;
- kolejność typów.

4. INNY PRZYKŁAD

```
static void pokaz(String tekst) {
    System.out.println(tekst);
}
```

```
static void pokaz(String tekst, int liczba) {
    System.out.println(tekst + liczba);
}
```

5. CO NIE WYSTARCZY?

Nie wystarczy zmienić tylko typu zwracanego, jeśli parametry pozostają takie same.

```
static int wartosc(int a) { return a; }
static double wartosc(int a) { return a; }
```

Taka deklaracja spowoduje błąd kompilacji, ponieważ lista parametrów jest identyczna.

TA SAMA NAZWA, INNA LISTA PARAMETKÓW

Ta sama nazwa
+
inna lista parametrów
=
przeciążanie

6. DLACZEGO WARTO?

- spójne nazewnictwo;
- wygodniejsze API;
- kod jest bardziej naturalny w użyciu.

43. ZAKRES ZMIENNEJ

Zakres zmiennych — gdzie zmienna jest dostępna?

1. ZMIENNA LOKALNA

Tworzona wewnątrz metody lub bloku.

```
static void test() {
    int liczba = 10;
}
```

`liczba` jest dostępna tylko wewnątrz tej metody.

2. ZMIENNA W BLOKU

```
if (true) {
    int x = 5;
}
```

`x` przestaje być dostępne po wyjściu z klamr.

3. DLACZEGO ZAKRES JEST WAŻNY?

- zmniejsza ryzyko błędów;
- utrzymuje porządek;
- ogranicza przypadkowe użycie danych.

4. PROSTA ZASADA



Im bardziej wewnętrzna zmienna, tym mniejszy jej zakres.

5. TYPOWY BŁĄD

```
if (true) {
    int wynik = 5;
}

System.out.println(wynik);
```

BŁĄD — `wynik` nie istnieje poza blokiem.

44. ZMIENNE LOKALNE A POLA KLASY

Poznaj różnicę.

ZMIENNA LOKALNA

- deklarowana w metodzie;
- istnieje tylko podczas działania tej części kodu;
- dostępna w ograniczonym zakresie.

```
void oblicz() {
    int wynik = 10;
}
```

Zmienna „wynik" istnieje tylko wewnątrz metody oblicz().

POLE KLASY

- deklarowane w klasie;
- opisuje stan obiektu lub klasy;
- może być używane przez różne metody zgodnie z dostępem.

```
class Uczen {
  String imie;

  void pokazImie() {
    System.out.println(imie);
  }
}
```

Pole „imie" należy do klasy Uczen i może być używane przez różne metody (np. pokazImie()).

KIEDY UŻYWAĆ LOKALNEJ?

- wynik tymczasowy;
- licznik;
- dane pomocnicze.

Zmienne lokalne świetnie sprawdzają się do przechowywania wartości, które są potrzebne tylko w danym miejscu kodu.

NAJWAŻNIEJSZA RÓŻNICA

lokalna → należy do metody/bloku

pole → należy do klasy/obiektu

KIEDY POLA?

- dane opisujące obiekt;
- wartości potrzebne wielu metodom.

Pola klasy przechowują stan obiektu (np. imię ucznia) i mogą być używane w różnych metodach zgodnie z zasadami dostępu (np. public, private).

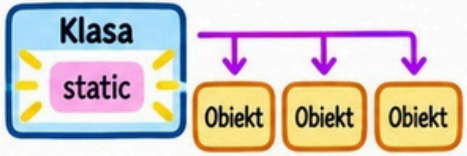
Nie zamieniaj każdej zmiennej w pole klasy — używaj najmniejszego potrzebnego zakresu.

45. STATIC

'static' — element należący do klasy

1. CO OZNACZA?

Element oznaczony jako `static` należy do klasy, a nie do konkretnego obiektu.



2. PRZYKŁAD METODY

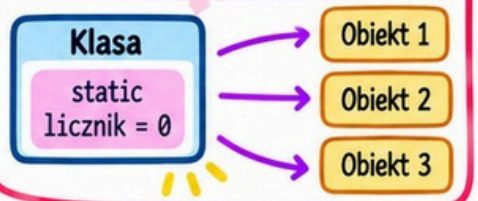
```
static void powitaj() {
    System.out.println("Cześć");
}
```

Możemy ją wywołać w odpowiednim kontekście bez tworzenia osobnego obiektu.

3. PRZYKŁAD POLA

```
static int licznik = 0;
```

Ta zmienna jest współdzielona na poziomie klasy.



4. BEZ 'static'

STATIC → wspólne dla klasy



NIESTATIC → związane z konkretnym obiektem



5. GDZIE POCZĄTKUJĄCY SPOTYKA 'static'?

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

Metoda main jest metodą statyczną, dzięki czemu program może zostać uruchomiony bez tworzenia obiektu klasy.

6. NA TYM ETAPIE ZAPAMIĘTAJ

Nie musisz jeszcze znać wszystkich szczegółów. Najważniejsza idea to:

klasa kontra obiekt



Temat wróci szerzej w module OOP.

46. REKURENCJA

metoda, która wywołuje samą siebie.

1. IDEA

Metoda wywołuje samą siebie.

2. MUSI MIEĆ DWA ELEMENTY

PRZYPADEK BAZOWY

zatrzymuje rekurencję.

KROK REKURENCYJNY

wywołuje metodę ponownie.

3. PRZYKŁAD

```
static int silnia(int n) {
    if (n <= 1) {
        return 1;
    }
    return n * silnia(n - 1);
}
```

4. JAK DZIAŁA `silnia(4)`?

silnia(4)

4 × silnia(3)

4 × 3 × silnia(2)

4 × 3 × 2 × silnia(1)

24

5. NAJWAŻNIEJSZE OSTRZEŻENIE

Bez warunku stopu rekurencja będzie trwała dalej i może doprowadzić do błędu.

6. REKURENCJA CZY PĘTLA?

- niektóre problemy naturalnie pasują do rekurencji;
- często zwykła pętla jest prostsza;
- wybieraj rozwiązanie czytelniejsze.

Rekurencja = rozwiąż mniejszą wersję tego samego problemu.

47. DOBRA NAZWA METODY

Jak nazywać metody, żeby kod był czytelny.

1. NAZWA POWINNA OPISYWAĆ DZIAŁANIE

Nazwa metody powinna jasno mówić, co metoda robi. Dzięki temu kod jest łatwiejszy do zrozumienia.

Dobrze:

```
obliczSume()
pobierzImie()
sprawdzHaslo()
wyswietlMenu()
```

Słabo:

```
x()
test2()
abc()
zrobCos()
```

2. UŻYWAJ CZASOWNIKÓW

Metody zwykle wykonują akcje. Dlatego ich nazwy powinny być czasownikami (w formie rozkazującej w liczbie pojedynczej).

```
dodajProdukt()
usunProdukt()
znajdzUzytkownika()
```

Czasowniki od razu mówią, że metoda coś robi.

3. STYL CAMELCASE

Nazwy metod w Javie zapisujemy w stylu camelCase.

```
obliczSredniaOcene()
```

oblicz | Srednia | Ocene

pierwsze słowo małą literą — kolejne słowa z wielkiej litery

Pierwsze słowo małą literą, kolejne z wielkiej.

4. NAZWA KONTRA ODPOWIEDZIALNOŚĆ

Jeśli trudno nazwać metodę, być może robi zbyt wiele rzeczy.

Dobrze nazwana metoda zazwyczaj ma jedną, jasno określoną odpowiedzialność i wykonuje tylko jedną rzecz.

5. PORÓWNANIE

Ta sama funkcjonalność, ale różne nazwy metod – duża różnica w czytelności!

`m1()` ✗	Co robi? Trudno zgadnąć.
`obliczPodatek()` ✓	Wiadomo od razu.

Dobra nazwa zmniejsza potrzebę dodawania komentarzy.

48. PODZIAŁ PROGRAMU NA METODY

Nie pisz wszystkiego w `main` – dziel duży problem na mniejsze zadania.

WSZYSTKO W `main`

```
public class Kalkulator {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Podaj pierwszą liczbę: ");
    double a = scanner.nextDouble();
    System.out.print("Podaj drugą liczbę: ");
    double b = scanner.nextDouble();
    System.out.print("Wybierz operację (+ - * /): ");
    String op = scanner.next();
    double wynik = 0;
    if (op.equals("+")) {
      wynik = a + b;
    } else if (op.equals("-")) {
      wynik = a - b;
    } else if (op.equals("*")) {
      wynik = a * b;
    } else if (op.equals("/")) {
      wynik = a / b;
    } else {
      System.out.println("Nieznana operacja!");
    }
    System.out.println("Wynik: " + wynik);
  }
}
```

- ✗ trudno czytać – wszystko w jednym miejscu,
- ✗ trudno poprawiać – łatwo wprowadzić błąd,
- ✗ trudno testować – nie da się sprawdzić osobno części,
- ✗ dużo powtórzeń – podobny kod pojawia się wiele razy.

PODZIAŁ NA METODY

pobierzDane()

```
public static void pobierzDane() {
  // pobranie liczb i operacji od użytkownika
}
```

obliczWynik()

```
public static double obliczWynik() {
  // wykonanie obliczenia na podstawie
  // wcześniej pobranych danych
  return 0;
}
```

wyswietlWynik()

```
public static void wyswietlWynik(double wynik) {
  // wyświetlenie wyniku na ekranie
  System.out.println("Wynik: " + wynik);
}
```

Program składa się z kilku małych metod, które każda ma jedno konkretne zadanie.

PRZYKŁAD MYŚLENIA

Problem: Program kalkulatora

1. pobierz liczby;
2. pobierz operację;
3. wykonaj obliczenie;
4. pokaż wynik.

możemy zamienić to na:

WYWOŁANIA METOD

```
public static void main(String[] args) {
  pobierzLiczby();
  wykonajObliczenie();
  wyswietlWynik();
}
```

JEDNA METODA = JEDNO ZADANIE

Jedna metoda = jedno jasne zadanie.

- ✓ Metoda wykonuje tylko jedną rzecz.
- ✓ Ma czytelną nazwę.
- ✓ Może być użyta w różnych miejscach.
- ✓ Łatwo ją testować i modyfikować.

DUŻY PROBLEM → MAŁE ZADANIA → OSOBNE METODY

49. MINI KALKULATOR

Praktyka – prosty kalkulator z metodami

1. METODA DODAWANIA

Metoda dodaje dwie liczby i zwraca ich sumę.

```
static double dodaj(double a, double b) {
    return a + b;
}
```

Metoda przyjmuje dwa parametry typu double, a następnie zwraca wynik za pomocą słowa return.

2. METODA ODEJMOWANIA

Metoda odejmuje drugą liczbę od pierwszej i zwraca wynik.

```
static double odejmij(double a, double b) {
    return a - b;
}
```

Działa podobnie jak dodawanie, ale wykonuje operację odejmowania.

3. METODA WYŚWIETLANIA

Metoda wyświetla wynik na ekranie (nie zwraca wartości).

```
static void pokazWynik(double wynik) {
    System.out.println("Wynik: " + wynik);
}
```

Metoda nie zwraca wartości, ma typ void. Jej zadaniem jest tylko wyświetlenie wyniku.

4. UŻYCIE

Tak wywołujemy metody w programie.

```
double wynik = dodaj(10, 5);
pokazWynik(wynik);
```

Najpierw wywołujemy metodę dodaj(), przekazując argumenty, a następnie wyświetlamy wynik za pomocą pokazWynik().

5. CO TUTAJ ĆWICZYSZ?

- metody;
- parametry;
- argumenty;
- return;
- void;
- podział odpowiedzialności.

PRZEPŁYW DZIAŁANIA

10 + 5	Przekazujemy liczby do metody dodaj().
↓ dodaj()	Metoda dodaj() dodaje liczby i zwraca wynik.
↓ 15	Otrzymujemy wynik (15).
↓ pokazWynik()	Przekazujemy wynik do metody pokazWynik().
↓ Wynik: 15	Wynik zostaje wyświetlony na ekranie.

SPRÓBUJ SAM:

Dodaj metody:

```
pomnoz()
podziel()
```

Napisz metody do mnożenia i dzielenia dwóch liczb. Następnie przetestuj ich działanie, wywołując je i wyświetlając wyniki za pomocą pokazWynik().

Metody pozwalają budować program z małych, uporządkowanych elementów.

DOBRA METODA

Pisz kod, który łatwo rozwijać.

JEDNO GŁÓWNE ZADANIE

Metoda powinna robić jedną, konkretną rzecz. Nie próbujmy wykonywać wszystkiego naraz.

```
obliczCene();
```

Dobrze — jedna odpowiedzialność (tylko oblicza cenę).

```
obliczCeneIWyswietlIZapisz();
```

Źle — robi zbyt wiele rzeczy (oblicza, wyświetla i zapisuje).

CZYTELNA NAZWA

Nazwa metody powinna jasno mówić, co ona robi. Używaj czasowników i unikaj skrótów.

```
obliczSrednia();
sprawdzLogin();
wyswietlRaport();
```

MAŁO PARAMETRÓW

Im mniej parametrów, tym łatwiej zrozumieć, jak używać metody.

```
obliczCene(double cena, int ilosc)
```

2–3 parametry → łatwe do zrozumienia.

```
stworzZamowienie(String nazwa, int ilosc, double cena, boolean rabat, int typ, String klient, String adres)
```

Bardzo dużo parametrów → sprawdź projekt kodu.

UNIKAJ POWTARZANIA

Jeśli ten sam blok kodu pojawia się w wielu miejscach, to znak, że warto wydzielić z niego metodę.

```
System.out.println("Start");
// ...
System.out.println("Koniec");
```

```
System.out.println("Start");
// ...
System.out.println("Koniec");
```

Zamiast powtarzać kod — rozważ osobną metodę.

ZWRACAJ DANE, GDY SĄ POTRZEBNE

Jeśli metoda ma obliczyć wynik, lepiej go zwrócić, zamiast tylko wypisywać.

```
static double obliczVAT(double cena) {
    return cena * 0.23;
}
```

METODA POWINNA BYĆ ŁATWA DO TESTOWANIA

Dobrze zaprojektowana metoda ma jasne wejście i przewidywalny wynik. Dzięki temu łatwo ją przetestować.

Wejście jasne → wynik przewidywalny.

Czytelna metoda = jasna nazwa + jedno zadanie + prosty przepływ danych.

METODY

Podsumowanie — metody i organizacja kodu

DEFINICJA METODY

Nazwany fragment kodu wykonujący określone zadanie.

```
public static void powitanie() {
    System.out.println("Cześć!");
}
```

PARAMETRY

Dane oczekiwane przez metodę.

```
public static void powitanie(String imie) {
    System.out.println("Cześć, " + imie + "!");
}
```

String imie – parametr metody typu String o nazwie imie.

ARGUMENTY

Rzeczywiste wartości przekazywane podczas wywołania.

```
powitanie("Ania");
```

"Ania" – argument przekazany do parametru imie.

return

Zwraca wynik.

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

```
int wynik = dodaj(3, 5);
```

Metoda zwraca wartość typu int, którą możemy wykorzystać dalej.

void

Wykonuje zadanie bez zwracania wartości.

```
public static void pokazKomunikat() {
    System.out.println("To jest komunikat.");
}
```

Metoda z typem void nie zwraca wartości, wykonuje tylko określone czynności.

PRZECIĄŻANIE

Ta sama nazwa + różne listy parametrów.

```
public static int pole(int bok) {
    return bok * bok;
}
public static int pole(int a, int b) {
    return a * b;
}
```

Ta sama nazwa metody (pole), różne parametry – różne wersje metody.

ZAKRES

Określa, gdzie dane są dostępne.

```
public static void przyklad() {
    int x = 10;   // zmienna lokalna
    System.out.println(x);
}
// System.out.println(x); // BŁĄD – x
// jest dostępne tylko wewnątrz metody.
```

Zmienna lokalna istnieje tylko w obrębie metody, w której została utworzona.

ORGANIZACJA KODU

Duży problem dzielimy na mniejsze części.

```
public static void oblicz() {
    wczytajDane();
    int wynik = wykonajObliczenia();
    wyswietlWynik(wynik);
}

// każda metoda ma jedno zadanie
```

Program składa się z wielu małych metod, co ułatwia czytanie i rozwijanie kodu.

PROBLEM – Duży problem do rozwiązania.

→ **PODZIEL NA ZADANIA** – Rozbij problem na mniejsze części.

→ **UTWÓRZ METODY** – Dla każdej części stwórz osobną metodę.

→ **PRZEKAŻ DANE** – Użyj parametrów i argumentów.

→ **ZWRÓĆ WYNIK** – Zwróć wynik lub wykonaj zadanie bez zwracania wartości.

Teraz Twój kod może być krótszy, czytelniejszy i łatwiejszy do rozwijania.

CO DALEJ?

Stringi, tablice i przetwarzanie większej ilości danych.

MODUŁ 04

STRINGI, TABLICE I PRZETWARZANIE DANYCH

Pracuj z tekstem, wieloma wartościami i uporządkowanymi zbiorami danych.

DANE

String

Przechowuje tekst w cudzysłowach.

```
String tekst =
  "Witaj, Java!";
```

„Witaj, Java!"

char

Przechowuje pojedynczy znak.

```
char litera = 'A';
```

A

tablice

Przechowują wiele wartości tego samego typu.

```
int[] liczby = {10, 20, 30};
```

10	20	30
0	1	2

indeks

Elementy tablicy mają indeksy zaczynające się od 0.

```
int[] liczby = {5, 8, 11, 14};
```

5	8	11	14
0	1	2	3

.length

Zwraca liczbę elementów tablicy lub długość stringa.

```
System.out.println(tekst.length());
System.out.println(liczby.length);
```

pętle

Umożliwiają przetwarzanie wszystkich elementów.

```
for (int i = 0; i < liczby.length; i++) {
    System.out.println(liczby[i]);
}
```

tablice 2D

Przechowują dane w wierszach i kolumnach.

```
int[][] macierz = {
    {1, 2, 3},
    {4, 5, 6}
};
```

przetwarzanie

Możemy analizować, wyszukiwać, sortować i modyfikować dane.

STRING

CO PRZECHOWUJE?

String przechowuje tekst, który składa się z wielu znaków.

- imię
- wiadomość
- adres
- opis
- nazwa produktu

TWORZENIE STRINGA

Stringa tworzymy, przypisując tekst w podwójnych cudzysłowach:

```
String imie = "Anna";
String jezyk = "Java";
```

TEKST ZAPISUJEMY W PODWÓJNYCH CUDZYSŁOWACH

" "

ŁĄCZENIE TEKSTÓW

Możemy łączyć teksty za pomocą operatora +:

```
String tekst = "Cześć " + imie;
```

Wynik: **Cześć Anna**

STRING MOŻE BYĆ PRZECHOWYWANY W ZMIENNEJ

String możemy zapisać w zmiennej i używać go w programie:

```
String miasto = "Warszawa";
```

WAŻNE

String nie jest typem prymitywnym jak int czy boolean; jest klasą używaną bardzo często w Javie.

String = tekst, z którym możemy wykonywać wiele gotowych operacji.

STRING METHODS

Najważniejsze metody String — pracuj z tekstem szybciej.

.length()

Liczba znaków.

```
"Java".length() → 4
```

Zwraca liczbę znaków w łańcuchu znaków.

.toUpperCase()

Zmienia litery na wielkie.

```
"java".toUpperCase() → "JAVA"
```

Zwraca nowy łańcuch znaków z wszystkimi literami zamienionymi na wielkie.

.toLowerCase()

Zmienia litery na małe.

```
"JAVA".toLowerCase() → "java"
```

Zwraca nowy łańcuch znaków z wszystkimi literami zamienionymi na małe.

.contains()

Sprawdza, czy tekst zawiera fragment.

```
"Java jest super"
  .contains("Java") → true
```

Zwraca true, jeśli łańcuch znaków zawiera podany fragment (wielkość liter ma znaczenie).

.startsWith()

Sprawdza początek tekstu.

```
"Java jest super"
  .startsWith("Java") → true
```

Zwraca true, jeśli łańcuch znaków zaczyna się od podanego fragmentu.

.endsWith()

Sprawdza koniec tekstu.

```
"Java jest super"
  .endsWith("super") → true
```

Zwraca true, jeśli łańcuch znaków kończy się podanym fragmentem.

String jest obiektem, dlatego oferuje wiele gotowych metod do pracy z tekstem.

.trim()

Usuwa zbędne spacje z początku i końca.

```
"  Java  " .trim() → "Java"
```

Zwraca nowy łańcuch znaków bez spacji na początku i końcu.

JAK POPRAWNIE PORÓWNYWAĆ STRINGI

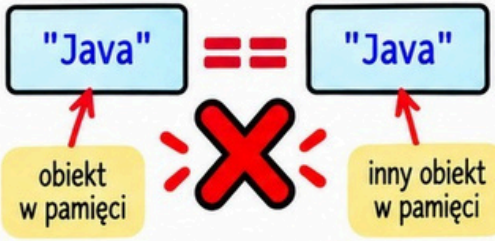
– equals() zamiast ==

NIE TAK

```
String a = "Javã";
String b = "Java";

a == b;
```

== nie jest właściwym sposobem porównywania treści Stringów.



Nawet jeśli teksty wyglądają identycznie, mogą to być różne obiekty w pamięci, więc wynik porównania może być false.

TAK

a.equals(b) → porównuje treść

equals()

Sprawdza, czy teksty mają dokładnie taką samą treść.

```
"Java".equals("Java") → true
```

equalsIgnoreCase()

Ignoruje wielkość liter.

```
"JAVA".equalsIgnoreCase("java") → true
```

Praktyczny przykład

```
String haslo = "java123";
if (haslo.equals("java123")) {
    System.out.println("Poprawne hasło");
}
```





Dla treści Stringów zapamiętaj: equals()

ZNAKI I INDEKSY – ZAJRZYJ DO ŚRODKA STRINGA



INDEKSY ZACZYNAJĄ SIĘ OD 0

- Pierwszy znak ma indeks 0, drugi 1, trzeci 2 i tak dalej.
- Indeksy zwiększają się o 1.
- To bardzo ważne w Javie.

charAt()

Metoda charAt() zwraca znak (char) z podanego indeksu.

```
String tekst = "Java";
char znak = tekst.charAt(0);
```

Wynik: 'J'

OSTATNI ZNAK

Indeks ostatniego znaku w Stringu obliczamy tak:

```
tekst.length() - 1
```



Dla napisu „Java" o długości 4, ostatni znak ma indeks 3 (bo 4 - 1 = 3).

J	a	v	a
indeks 0	indeks 1	indeks 2	indeks 3

PRZYKŁAD

Pobierzemy ostatni znak z napisu „Java".

```
String tekst = "Java";
char ostatni =
tekst.charAt(tekst.length() - 1);
```

Wynik: 'a'

UWAGA



Jeśli podasz indeks spoza zakresu, program zgłosi błąd (wyjątek StringIndexOutOfBoundsException).

```
String tekst = "Java";
tekst.charAt(10);
```

Dla „Java" → indeks poza zakresem. Dozwolone są indeksy od 0 do 3.

Indeksowanie od 0 pojawi się także w tablicach i kolekcjach.

substring() – pobierz fragment tekstu

PODSTAWOWA IDEA

substring() tworzy fragment oryginalnego tekstu.

Nie tworzy nowego tekstu od zera, tylko zwraca część istniejącego łańcucha znaków.

PRZYKŁAD TEKSTU I INDEKSÓW

P	R	O	G	R	A	M	O	W	A	N	I	E
0	1	2	3	4	5	6	7	8	9	10	11	12

indeksy zaczynają się od 0

długość tekstu = 13

OD POZYCJI DO KOŃCA

```
String tekst = "Programowanie";
String fragment = tekst.substring(7);
```

Wynik: "WANIE"

P	R	O	G	R	A	M	W	W	A	N	I	E
0	1	2	3	4	5	6	7	8	9	10	11	12

od pozycji 7 do końca

OD POZYCJI DO POZYCJI

```
String tekst = "Programowanie";
String fragment = tekst.substring(0, 7);
```

Wynik: "Program"

P	R	O	G	R	A	M	O	W	A	N	I	E
0	1	2	3	4	5	6	7	8	9	10	11	12

od pozycji 0 do pozycji 7 (nie wliczając 7)

- pierwszy indeks → gdzie zaczynamy
- drugi indeks → gdzie kończymy
- drugi indeks nie należy do wyniku

PRZYKŁADOWE ZASTOSOWANIA

- skracanie tekstu
- pobieranie prefiksu — np. "Pro..."
- pobieranie fragmentu kodu
- analizowanie danych tekstowych

ŁĄCZENIE Z INNYMI METODAMI

```
String tekst = "Programowanie";
String wynik = tekst.substring(0, 4).toUpperCase();
```

Wynik: "PROG"

P	R	O	G	R	A	M	O	W	A	N	I	E
0	1	2	3	4	5	6	7	8	9	10	11	12

substring(0, 4) → następnie toUpperCase()

Indeksy są kluczem do świadomej pracy z tekstem.

TABLICA

Wiele wartości w jednej zmiennej.

PO CO TABLICA?

Bez tablicy:

```
int a = 10;
int b = 20;
int c = 30;
```

Z tablicą:

```
int[] liczby = {10, 20, 30};
```

Zamiast wielu osobnych zmiennych możemy użyć jednej tablicy.

CO PRZECHOWUJE?

Wiele wartości tego samego typu.

liczby → | 10 | 20 | 30 |

W tablicy możemy przechowywać wiele wartości, ale wszystkie muszą być tego samego typu (np. `int`).

TWORZENIE TABLICY

Tablicę tworzymy, podając jej elementy w nawiasach klamrowych.

```
int[] punkty = {10, 20, 30, 40};
```

W tej tablicy przechowujemy 4 liczby typu `int`.

punkty → | 10 | 20 | 30 | 40 |

INDEKSY

Każda pozycja w tablicy ma swój indeks zaczynający się od 0.

10	[0]
20	[1]
30	[2]
40	[3]

punkty

10	20	30	40
0	1	2	3

POBIERANIE ELEMENTU

Element tablicy odczytujemy, używając indeksu w nawiasach kwadratowych.

```
System.out.println(punkty[1]);
```

Na ekranie zostanie wypisana wartość:

20

ZMIANA ELEMENTU

Możemy zmienić wartość wybranego elementu, przypisując nową wartość pod danym indeksem.

```
punkty[1] = 25;
```

Po zmianie tablica wygląda tak:

10	25	30	40

TABLICA MA STAŁĄ DŁUGOŚĆ

Długość tablicy (liczba elementów) jest ustalana podczas jej tworzenia i nie może się później zmienić.



punkty → | 10 | 20 | 30 | 40 |

długość: 4

Nie możemy dodać nowych elementów ani zmniejszyć długości tablicy.

TWORZENIE TABLIC
i właściwość length

TABLICA Z GOTOWYMI WARTOŚCIAMI

Możemy utworzyć tablicę od razu z podanymi wartościami. Elementy są zapisywane w podanej kolejności, a rozmiar tablicy jest dopasowany automatycznie.

```
String[] imiona = {
    "Anna",
    "Jan",
    "Ola"
};
```

Powstaje tablica z 3 elementami:

0	1	2
Anna	Jan	Ola

TABLICA O OKREŚLONYM ROZMIARZE

Możemy utworzyć tablicę o określonym rozmiarze, podając liczbę elementów w nawiasach kwadratowych.

```
int[] liczby = new int[5];
```

0	1	2	3	4
0	0	0	0	0

Java tworzy pozycje z wartościami domyślnymi.

DŁUGOŚĆ TABLICY

Właściwość **length** zwraca liczbę elementów w tablicy.

```
liczby.length
```

Wynik: **5**

Tablica → .length **bez nawiasów.**

OSTATNI INDEKS

Ostatni indeks tablicy jest zawsze o 1 mniejszy od jej długości.

OSTATNI INDEKS = length - 1

Przykład: length = 5

0	1	2	3	4

PRAKTYCZNY PRZYKŁAD

Tworzymy tablicę, wypisujemy jej długość i sprawdzamy wynik.

```
int[] punkty = {10, 20, 30};
System.out.println(punkty.length);
```

Wynik: **3**

0	1	2
10	20	30

Tablica ma 3 elementy, więc length zwraca 3.

WAŻNE!

length mówi, ile elementów ma tablica — **nie jaki jest ostatni indeks.**

FOR + TABLICA

Pętla for i tablica — przejdź przez wszystkie elementy.

PRZYKŁADOWA TABLICA

Mamy tablicę z 4 elementami typu `int`.

```
int[] liczby = {10, 20, 30, 40};
```

0	1	2	3
10	20	30	40

Każda pozycja ma swój indeks, który zaczyna się od 0.

PRZEJŚCIE PO INDEKSACH

Aby przejść przez wszystkie elementy tablicy, używamy pętli `for` i kolejnych indeksów.

```
for (int i = 0; i < liczby.length; i++) {
    System.out.println(liczby[i]);
}
```

Pętla wykonuje się dla kolejnych indeksów: 0, 1, 2, 3, aż do `length - 1`.

CO OZNACZA „i"?

Zmienna `i` to aktualny indeks, po którym się poruszamy.

i (indeks)		liczby[i] (wartość)
i = 0	→	10
i = 1	→	20
i = 2	→	30
i = 3	→	40

W każdej iteracji pętli `i` ma inną wartość, a `liczby[i]` zwraca element z tego indeksu.

DLACZEGO UŻYWAMY length?

Warunek w pętli:

```
i < liczby.length
```

Dzięki temu pętla wykona się tylko dla istniejących elementów tablicy. Nie dojdziemy do indeksu spoza zakresu, co zapobiega błędom (np. ArrayIndexOutOfBoundsException).

`liczby.length` zwraca rzeczywistą liczbę elementów w tablicy. Ostatni poprawny indeks to `length - 1`.

CO MOŻNA ROBIĆ W PĘTLI?

- wyświetlać wartości tablicy;
- sumować liczby;
- szukać elementu;
- zmieniać dane w tablicy;
- sprawdzać warunki (np. czy element spełnia określone kryterium).

Pętla daje nam pełną kontrolę nad każdym elementem tablicy.

PRZYKŁAD – SUMA

Obliczmy sumę wszystkich elementów tablicy.

```
int suma = 0;

for (int i = 0; i < liczby.length; i++) {
    suma += liczby[i];
}
```

W każdej iteracji dodajemy bieżący element do zmiennej `suma`.

Po wykonaniu pętli `suma = 100`.

Pętla + indeks = pełna kontrola nad tablicą.

FOR-EACH

Prostsze przechodzenie po elementach.

DO CZEGO SŁUŻY?

- Pozwala przejść po wszystkich elementach tablicy w prosty i czytelny sposób.
- Nie musimy używać indeksów.
- Świetnie sprawdza się, gdy chcemy tylko odczytać wartości tablicy.

for-each przechodzi po każdym elemencie automatycznie.

SKŁADNIA

Używamy następującej składni pętli for-each:

```
for (typ element : tablica) {
    // kod do wykonania dla każdego elementu
}
```

typ element to taki sam typ, jak elementy tablicy (np. int, double, String).

PRZYKŁAD

Przejdźmy po tablicy z liczbami i wypiszmy każdą wartość:

```
int[] liczby = {10, 20, 30};

for (int liczba : liczby) {
    System.out.println(liczba);
}
```

Wynik działania programu:

```
10
20
30
```

CZYTAJ TO TAK

Pętla for-each można czytać w prosty sposób:

```
for (int liczba : liczby) { ... }
```

Dla każdego elementu tablicy liczby... → ...przypisz jego wartość do zmiennej liczba i wykonaj blok kodu.

for KONTRA for-each

KLASYCZNA PĘTLA for	PĘTLA for-each
Masz indeks.	Prostszy zapis.
Możesz zmieniać konkretne pozycje.	Świetny do odczytu wartości.
Większa kontrola nad tablicą.	Nie pracujesz bezpośrednio z indekem.
	Nie służy do zmieniania wartości w tablicy.

KIEDY UŻYWAĆ?

Używaj pętli for-each, gdy:

- chcesz przejść przez wszystkie elementy tablicy,
- interesuje Cię tylko odczyt wartości,
- nie potrzebujesz indeksów,
- zależy Ci na prostszym i bardziej czytelnym kodzie.

Jeśli musisz modyfikować konkretne pozycje w tablicy, lepiej użyj klasycznej pętli for.

Nie potrzebujesz indeksu? for-each jest często czytelniejszy.

SZUKANIE

Jak znaleźć element w tablicy.

DANE

Mamy tablicę z liczbami oraz wartość, której szukamy.

```
int[] liczby = {4, 7, 10, 15};
int szukana = 10;
```

0	1	2	3
4	7	10	15

Szukaną wartość możemy zmienić na dowolną inną liczbę.

PRZEJDŹ PRZEZ TABLICĘ

Używamy pętli for-each, aby sprawdzić każdy element tablicy.

```
for (int liczba : liczby) {
    if (liczba == szukana) {
        System.out.println("Znaleziono!");
    }
}
```

Pętla sprawdza kolejne elementy, a w środku porównujemy je z wartością szukaną.

LOGIKA

Działanie krok po kroku:

- Weź element tablicy (liczba)
- Porównaj z szukana (liczba == szukana?)
  - TAK → **Znaleziono!** Wykonaj akcję (np. wypisz komunikat).
  - NIE → Sprawdź następny element.

WERSJA Z BREAK

Po znalezieniu elementu nie musimy już dalej szukać, więc używamy break.

```
for (int liczba : liczby) {
    if (liczba == szukana) {
        System.out.println("Znaleziono!");
        break;
    }
}
```

Instrukcja break przerywa pętlę, gdy element został znaleziony. Dzięki temu nie sprawdzamy już kolejnych elementów.

GDZIE SIĘ PRZYDAJE?

- wyszukiwanie ID;
- sprawdzanie wartości;
- odnajdywanie nazw;
- proste filtrowanie danych.

Szukanie w tablicy to podstawa wielu praktycznych zadań, np. sprawdzenia, czy dana wartość w ogóle istnieje w naszych danych.

WAŻNA IDEA

Szukanie w tablicy to połączenie dwóch rzeczy:

pętla + warunek

- Pętla przechodzi przez wszystkie elementy tablicy (po kolei).
- Warunek sprawdza, czy aktualny element jest równy wartości szukanej.

Razem pozwalają nam znaleźć element wśród wielu wartości.

Szukaj → porównuj → reaguj.

MIN + MAX

Znajdź najmniejszą i największą liczbę w tablicy.

MAX

Znajdujemy największą wartość w tablicy, przechodzac po wszystkich elementach.

```
int[] liczby = {8, 3, 12, 5};

int max = liczby[0];  // na początku
                      // zakładamy pierwszą wartość

for (int liczba : liczby) {
    if (liczba > max) {
        max = liczba;
    }
}
```

Zmienna `max` pamięta największą wartość znalezioną do tej pory.

Po wykonaniu pętli:

`max = 12`

MIN

Znajdujemy najmniejszą wartość w tablicy, przechodzac po wszystkich elementach.

```
int[] liczby = {8, 3, 12, 5};

int min = liczby[0];  // na początku
                      // zakładamy pierwszą wartość

for (int liczba : liczby) {
    if (liczba < min) {
        min = liczba;
    }
}
```

Zmienna `min` pamięta najmniejszą wartość znalezioną do tej pory.

Po wykonaniu pętli:

`min = 3`

JAK TO DZIAŁA?

1. Zacznij od pierwszej wartości w tablicy.
2. Porównuj kolejne elementy tablicy.
3. Jeśli znajdziesz lepszą wartość (większą dla max lub mniejszą dla min) — zapamiętaj ją.

Przechodzimy po wszystkich elementach i na bieżąco aktualizujemy wynik.

TYPOWY WZORZEC

Zarówno dla `min`, jak i dla `max` używamy tego samego schematu:

wartość początkowa
↓
pętla po wszystkich elementach
↓
porównanie i ewentualna aktualizacja wyniku

GDZIE SIĘ PRZYDAJE?

- najwyższy wynik;
- najniższa cena;
- najwiekszy pomiar;
- najmniejsza wartość.

Nie musisz sortować tablicy, żeby znaleźć `min` lub `max`.

SUMA + ŚREDNIA

Podstawowe przetwarzanie danych.

TABLICA

Mamy tablicę z ocenami (punktami).

```
int[] punkty = {10, 20, 30, 40};
```

0	1	2	3
10	20	30	40

Każda pozycja ma swój indeks, który zaczyna się od 0.

ZMIENNA DO SUMOWANIA

Tworzymy zmienną, która będzie przechowywać sumę wszystkich wartości z tablicy.

```
int suma = 0;
```

Na początku suma wynosi 0.

DODAWANIE KOLEJNYCH WARTOŚCI

Przechodzimy przez wszystkie elementy tablicy i dodajemy każdy z nich do zmiennej `suma`.

```
for (int punkt : punkty) {
    suma += punkt;
}
```

Pętla for-each przechodzi przez kolejne wartości tablicy.

Po zakończeniu pętli: `suma = 100`

SCHEMAT

Krok Ro kroku: jak obliczamy sumę i średnią.

1. **DANE** (tablica)
2. **PĘTLA** (przejdź przez elementy)
3. **SUMA** (dodawaj wartości)
4. **PODZIEL PRZEZ LICZBĘ ELEMENTÓW**
5. **ŚREDNIA** (wynik)

ŚREDNIA

Aby obliczyć średnią, dzielimy sumę przez liczbę elementów w tablicy.

```
double srednia =
    (double) suma / punkty.length;
```

Dla naszej tablicy średnia wynosi:

`srednia = 25.0`

WZORZEC AKUMULATORA

Używamy jednej zmiennej (akumulatora), która jest wielokrotnie aktualizowana w pętli.

```
int suma = 0;
for (int punkt : punkty) {
    suma += punkt;
}
```

Za każdym razem do zmiennej `suma` dodajemy kolejną wartość z tablicy.

DLACZEGO (double)?

Jeśli podzielimy dwie liczby typu `int`, otrzymamy dzielenie całkowite (bez części dziesiętnej). Rzutowanie na `double` powoduje, że wynik jest liczbą zmiennoprzecinkową.

```
int a = 5, b = 2;
int wynik = a / b;  // 2

double wynik2 = (double) a / b; // 2.5
```

Dzięki `(double)` otrzymujemy dokładny wynik że średnią.

Akumulator pomaga krok po kroku przetwarzać dane.

TABLICA 2D

Wiersze i kolumny

JAK WYGLĄDA?

Tablica 2D to tabela wartości ułożonych w wierszach i kolumnach.

1	2	3
4	5	6
7	8	9

Możemy ją sobie wyobrazić jako tabelę, tak jak na rysunku.

TWORZENIE

Tablicę 2D tworzymy, podając wartości dla kolejnych wierszy.

```
int[][] macierz = {
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
```

Każdy wiersz to osobna tablica (zawierająca wartości dla danej kolumny).

DWA INDEKSY

Do elementu tablicy 2D odwołujemy się za pomocą dwóch indeksów:

`macierz[1][2]`

- pierwszy indeks → wiersz;
- drugi indeks → kolumna.

Dla naszej tablicy wartość elementu `macierz[1][2]` to:

6

INDEKSY ZACZYNAJĄ SIĘ OD 0

Podobnie jak w zwykłych tablicach, indeksy wierszy i kolumn zaczynają się od 0.

`macierz[0][0]`

oznacza element z pierwszego wiersza i pierwszej kolumny.

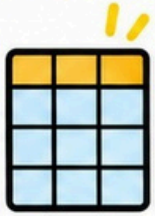
Dla naszej tablicy wartość `macierz[0][0]` to:

1

GDZIE SIĘ PRZYDAJĄ?

Tablice 2D używamy, gdy chcemy przechowywać dane w formie tabeli.

- plansze (np. gry);
- tabele (np. wyniki);
- macierze (np. matematyczne);
- siatki (np. obrazy);
- dane w wierszach i kolumnach.



JAK PRZEJŚĆ PO WSZYSTKICH DANYCH?

Aby przejść przez wszystkie elementy tablicy 2D, zazwyczaj używamy jednej pętli wewnątrz drugiej.

```
for (int i = 0; i < macierz.length; i++) {
    for (int j = 0; j < macierz[i].length; j++) {
        System.out.print(macierz[i][j] + " ");
    }
}
```

Zewnętrzna pętla przechodzi po wszystkich wierszach, a wewnętrzna po wszystkich kolumnach danego wiersza.

`macierz.length` zwraca liczbę wierszy, a `macierz[i].length` zwraca liczbę kolumn w i-tym wierszu.

Tablica 2D = tablica zawierająca inne tablice.

2 PĘTLE = 2 WYMIARY

Pętle i tablica 2D – przejdź przez całą tabelę.

PRZYKŁAD DANYCH

Mamy tablicę 2D (3 wiersze i 2 kolumny) z przykładowymi wartościami.

```
int[][] dane = {
    {1, 2},
    {3, 4},
    {5, 6}
};
```

	kolumna 0	kolumna 1
wiersz 0	1	2
wiersz 1	3	4
wiersz 2	5	6

PĘTLA PO WIERSZACH

Zewnętrzna pętla przechodzi po wszystkich wierszach tablicy 2D. Używamy właściwości `length` dla tablicy.

```
for (int i = 0; i < dane.length; i++) {
    // tutaj będzie pętla po kolumnach
}
```

`dane.length` zwraca liczbę wierszy w tablicy 2D (w tym przypadku 3).

PĘTLA PO KOLUMNACH

Wewnętrzna pętla przechodzi po wszystkich kolumnach w aktualnym wierszu.

```
for (int j = 0; j < dane[i].length; j++) {
    // tutaj używamy dane[i].length
}
```

`dane[i].length` zwraca liczbę kolumn w wierszu o indeksie `i` (w tym przypadku 2).

JAK TO DZIAŁA?

Pętle wykonują się w takiej kolejności:

- wiersz 0 → kolumna 0 → kolumna 1
- wiersz 1 → kolumna 0 → kolumna 1
- wiersz 2 → kolumna 0 → kolumna 1

Najpierw wybieramy wiersz (pętla zewnętrzna), a potem przechodzimy po wszystkich kolumnach (pętla wewnętrzna).

PEŁNY PRZYKŁAD

Używamy dwóch zagnieżdżonych pętli: zewnętrznej po wierszach i wewnętrznej po kolumnach.

```
for (int i = 0; i < dane.length; i++) {
    for (int j = 0; j < dane[i].length; j++) {
        System.out.println(dane[i][j]);
    }
}
```

Ten kod wypisze kolejno wszystkie elementy tablicy 2D, wiersz po wierszu, od lewej do prawej.

CO OZNACZAJĄ ZMIENNE?

Używamy dwóch zmiennych sterujących pętlami.

- `i` → wiersz — Zmienna `i` określa, który wiersz aktualnie przetwarzamy.
- `j` → kolumna — Zmienna `j` określa, którą kolumnę w aktualnym wierszu przeglądamy.

Zewnętrzna pętla wybiera wiersz.

Wewnętrzna przechodzi po jego elementach.

DANE W JAVIE

STRING

Przechowuje tekst.

```
String tekst = "Java";
String imie = "Anna";
```

String to obiekt, który przechowuje ciąg znaków (tekst).

METODY STRING

Najczęściej używane metody klasy String.

```
tekst.length();            // liczba znaków
tekst.contains("Java");    // czy zawiera
tekst.toUpperCase();       // na wielkie litery
tekst.trim();              // usuwa spacje
tekst.substring(0, 4);     // część tekstu
```

PORÓWNYWANIE TEKSTU

Do porównywania zawartości stringów używamy metody equals().

```
String a = "Java";
String b = "Java";
boolean wynik = a.equals(b);
```

equals() porównuje zawartość tekstów, a nie ich adres w pamięci.

INDEKSY

Zaczynają się od 0.

```
String tekst = "Java";
```

0	1	2	3
J	a	v	a

Każdy znak ma swój indeks, który zaczyna się od 0.

TABLICE

Wiele wartości tego samego typu.

```
int[] liczby = {1, 2, 3, 4};
String[] imiona = {"Jan", "Anna"};
```

Tablica przechowuje wiele wartości tego samego typu.

LENGTH

Określa liczbę elementów.

```
int[] liczby = {10, 20, 30};
int rozmiar = liczby.length;
```

Działa dla tablic oraz dla stringów.

PĘTLE

Pozwalają przetwarzać kolejne elementy.

```
for (int i = 0; i < liczby.length; i++) {
    System.out.println(liczby[i]);
}
```

TABLICE 2D

Wiersze + kolumny.

```
int[][] liczby = {
    {1, 2, 3},
    {4, 5, 6}
};
```

	0	1	2
0	1	2	3
1	4	5	6

Tablica 2D to tablica tablic (wiersze i kolumny).

NAJCZĘSTSZE WZORCE

- znajdź element (np. o danej wartości);
- policz sumę elementów (np. wszystkie liczby);
- oblicz średnią;
- znajdź min i max (najmniejszą i największą wartość);
- przejdź po wszystkich danych i wykonaj na nich operację.

Teraz potrafisz pracować z większą ilością danych. Następny krok: dynamiczne kolekcje.

MODUŁ 05

LISTY, KOLEKCJE I GENERYKI

Przechowuj dane elastycznie i korzystaj z najważniejszych struktur kolekcji w Javie.

COLLECTIONS

List

- Uporządkowana kolekcja.
- Może zawierać duplikaty.
- Elementy mają indeksy (zaczynają się od 0).
- Najczęściej używana kolekcja w Javie.



```
List<String> lista =
new ArrayList<>();
```

ArrayList

- Implementacja interfejsu List.
- Dynamicznie rośnie.
- Szybki dostęp po indeksie (get()).
- Najlepszy wybór w większości przypadków.



```
List<String> lista =
 new ArrayList<>();
```

Map

- Przechowuje pary klucz → wartość.
- Klucze są unikalne.
- Bardzo przydatna do szybkiego wyszukiwania po kluczu.




```
Map<String, Integer> oceny =
new ArashMap<>();
```

LinkedList

- Implementacja interfejsu List.
- Dwukierunkowa lista.
- Szybkie dodawanie i usuwanie elementów na początku/końcu.
- Przydatna, gdy często modyfikujemy kolekcję.



```
List<String> lista =
new LinkedList<>();
```

Set

- Kolekcja bez duplikatów.
- Nie gwarantuje kolejności elementów (HashSet).
- Utrzymuje unikalne wartości.



```
Set<Integer> liczby =
new HashSet<>();
```

Collections

- Klasa z użytecznymi metodami dla kolekcji.
- np. sortowanie, wyszukiwanie, zamiana, tworzenie niemodyfikowalnych kolekcji.



```
Collections.sort(lista);
Collections.reverse(lista);
```

Generyki <T>

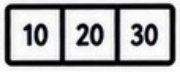
- Pozwalają określić typ elementów w kolekcjach.
- Dzięki nim mamy bezpieczeństwo typów (brak rzutowania).
- Możemy przechowywać dowolne typy, np. String, Integer, własne klasy.




```
List<String> nazwy = new ArrayList<>();
Map<Integer, String> mapa = new HashMap<>();
```

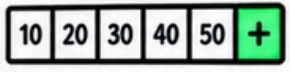
TABLICA

Stały rozmiar.
Po utworzeniu nie można go zmienić.



LISTA

Dynamicznie rośnie.
Możemy dodawać i usuwać elementy w trakcie działania programu.



PODSTAWOWE OPERACJE



DODAWANIE	USUWANIE	WYSZUKIWANIE	SORTOWANIE
add()	remove()	contains()	Collections.sort()

KOLEKCJE

w Javie — po co są i czym różnią się od tablic

PO CO SĄ?

- Pozwalają przechowywać i organizować wiele obiektów.
- Ułatwiają dodawanie, usuwanie, wyszukiwanie i przetwarzanie danych.
- Są częścią Java Collection Framework (dostępne w pakiecie `java.util`).
- Pozwalają pracować na danych w sposób elastyczny i wygodny.

Kolekcje to narzędzia stworzone do pracy z dynamicznymi zbiorami danych.

DLACZEGO NIE ZAWSZE TABLICA?

TABLICA

- ma stały rozmiar;
- mniej wygodna przy dodawaniu i usuwaniu.

```
int[] tab = {1, 2, 3};
```

1	2	3

KOLEKCJA

- może dynamicznie rosnąć;
- oferuje gotowe metody;
- ułatwia zarządzanie danymi.

```
List<Integer> lista =
new ArrayList<>();
```

1	2	3	4	...

NAJWAŻNIEJSZE RODZINY

LIST (Lista)

- Uporządkowana kolekcja.
- Elementy mają swoje pozycje (indeksy).
- Może zawierać duplikaty.

SET (Zbiór)

- Skupia się na unikalnych elementach.
- Brak powtórzeń (według zasad danej implementacji).

MAP (Mapą)

klucz → wartość

- Przechowuje pary: klucz → wartość.
- Klucze są unikalne.
- Pozwala szybko wyszukiwać wartość po kluczu.

CO BĘDZIEMY ROBIĆ?

- dodawać elementy;
- usuwać elementy;
- wyszukiwać elementy;
- iterować po elementach;
- sortować elementy;
- przechowywać różne typy obiektów.

Kolekcje mają gotowe metody, które ułatwiają wszystkie te operacje.

PRZYKŁADY IMPLEMENTACJI

ArrayList

```
List<String> lista
= new ArrayList<>();
```

- implementacja `List`
- szybki dostęp po indeksie
- często używana

LinkedList

```
List<String> lista
= new LinkedList<>();
```

- implementacja `List`
- wydajna przy dodawaniu i usuwaniu na początku/końcu
- elementy połączone w węzły

HashSet

```
Set<String> zbior
= new HashSet<>();
```

- implementacja `Set`
- brak duplikatów
- szybkie operacje dodawania, usuwania i wyszukiwania

HashMap

```
Map<String, Integer>
mapa = new HashMap<>();
```

- implementacja `Map`
- przechowuje pary klucz → wartość
- szybkie wyszukiwanie po kluczu

TABLICA — stały rozmiar

```
int[] tab = new int[3];
```

10	20	30

- rozmiar określony przy tworzeniu;
- nie można łatwo zmienić rozmiaru;
- ręczne zarządzanie elementami.

VS

KOLEKCJA — elastyczne zarządzanie danymi

```
List<Integer> lista = new ArrayList<>();
```

10	20	30	40	...

- może dynamicznie rosnąć;
- łatwe dodawanie i usuwanie;
- gotowe metody;
- wygodne zarządzanie danymi.

Kolekcje to jedne z najczęściej używanych narzędzi w codziennym kodzie Java.

LIST + ARRAYLIST

Dynamiczna lista elementów.

CO TO JEST `LIST`?

- uporządkowana kolekcja;
- elementy mają indeksy;
- może przechowywać duplikaty;
- rozmiar może się zmieniać.

TWORZENIE LISTY

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

List<String> imiona = new ArrayList<>();
```

DODAWANIE ELEMENTÓW

```
imiona.add("Anna");
imiona.add("Jan");
imiona.add("Ola");
```

```
["Anna", "Jan", "Ola"]
```

POBIERANIE ELEMENTU

```
imiona.get(1);
```

Wynik:

```
"Jan"
```

0	"Anna"
1	"Jan"
2	"Ola"

LICZBA ELEMENTÓW

```
imiona.size();
```

Wynik:

```
3
```

0	"Anna"
1	"Jan"
2	"Ola"

NAJWAŻNIEJSZA RÓŻNICA WZGLĘDEM TABLICY

TABLICA



rozmiar ustalony wcześniej

ARRAYLIST



może dynamicznie rosnąć i maleć

LIST VS ARRAYLIST

`List` opisuje sposób pracy, a `ArrayList` jest konkretną implementacją tej listy.

DODATKOWE INFORMACJE

- Używamy ArrayList, aby pracować z dynamiczną listą elementów.
- Elementy są uporządkowane i mają indeksy (od 0).
- Można dodawać, usuwać i pobierać elementy.
- Lista może przechowywać duplikaty.
- ArrayList znajduje się w pakiecie `java.util`.

ARRAYLIST METHODS

Najważniejsze operacje na liście.

add() – DODAJ

Metoda `add()` dodaje element na koniec listy.

```
lista.add("Java");
```

Wynik:

0	"Java"
1	"C++"

→

0	"Java"
1	"C++"
2	"Python"

get() – POBIERZ

Metoda `get()` zwraca element o podanym indeksie.

```
lista.get(0);
```

Wynik:

"Java"

0	"Java"
1	"C++"
2	"Python"

set() – ZMIEŃ

Metoda `set()` zastępuje element na podanym indeksie nową wartością.

```
lista.set(0, "Python");
```

Wynik:

0	"Java"
1	"C++"
2	"Python"

→

0	"Python"
1	"C++"
2	"Python"

remove() – USUŃ

Metoda `remove()` usuwa element z listy. Możemy podać indeks lub samą wartość.

```
lista.remove(0);          // usuwa pierwszy
lista.remove("Java"); // usuwa wartość
```

Wynik:

0	"Java"
1	"C++"
2	"Python"

→

0	"C++"
1	"Python"

size() – LICZBA ELEMENTÓW

Metoda `size()` zwraca aktualną liczbę elementów w liście.

```
lista.size();
```

Wynik:

3

0	"Java"
1	"C++"
2	"Python"

contains() – SPRAWDŹ OBECNOŚĆ

Metoda `contains()` sprawdza, czy lista zawiera podaną wartość.

```
lista.contains("Java");
```

Wynik:

true lub false

0	"Java"
1	"C++"
2	"Python"

PRZYKŁAD DZIAŁANIA

Dodajemy nowy element do listy.

["Java", "C++"]

↓

add("Python")

↓

["Java", "C++", "Python"]

clear() – USUŃ WSZYSTKO

Metoda `clear()` usuwa wszystkie elementy z listy. Po jej wywołaniu lista jest pusta.

```
lista.clear();
```

Wynik:

Lista oferuje gotowe operacje, których tablica nie udostępnia tak wygodnie.

LISTA + PĘTLE

Jak przechodzić po elementach `List`.

1. FOR Z INDEKSEM

Przechodzimy po elementach listy za pomocą indeksu od 0 do size() - 1.

```
for (int i = 0; i < imiona.size(); i++) {
    System.out.println(imiona.get(i));
}
```

Zalety:

- dostęp do indeksu;
- można pobrać konkretną pozycję.

2. FOR-EACH

Przechodzimy po elementach listy bez użycia indeksu.

```
for (String imie : imiona) {
    System.out.println(imie);
}
```

Zalety:

- prostszy;
- czytelniejszy;
- idealny do odczytu.

3. forEach()

Używamy metody forEach(), która wykonuje podaną operację dla każdego elementu listy.

```
imiona.forEach(imie ->
    System.out.println(imie)
);
```

Wersja nowoczesna oparta na operacji wykonywanej dla każdego elementu.

PRZYKŁADOWA LISTA

W przykładach używamy listy imion:

```
List<String> imiona = new ArrayList<>();
imiona.add("Anna");
imiona.add("Jan");
imiona.add("Ola");
```

NA CO UWAŻAĆ?

- Zaczynamy od indeksu 0.
- Warunek to `< imiona.size()`.
- Nie przekraczamy zakresu listy.

Dostęp do nieistniejącego indeksu powoduje wyjątek `IndexOutOfBoundsException`.

KIEDY KTÓRĄ PĘTLĘ?

Potrzebujesz indeksu? →	for
Chcesz tylko czytać elementy? →	for-each
Chcesz bardziej funkcyjny zapis? →	forEach()

WAŻNA UWAGA

Lista i pętla bardzo często występują razem.

PODSUMOWANIE

- Są 3 sposoby na przechodzenie po liście.
- `for` daje dostęp do indeksu.
- `for-each` jest najprostszy do odczytu.
- `forEach()` to nowoczesna, funkcyjna wersja.

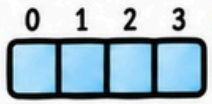
ARRAYLIST VS LINKEDLIST

Poznaj podstawową różnicę.

ARRAYLIST

Jak myśleć?

Elementy są przechowywane w strukturze przypominającej dynamiczną tablicę.



Mocne strony:

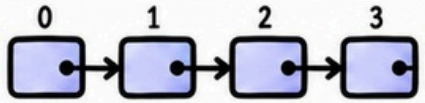
- szybki dostęp po indeksie;
- bardzo popularna;
- dobry wybór w większości prostych przypadków.

```
List<String> dane = new ArrayList<>();
```

LINKEDLIST

Jak myśleć?

Elementy tworzą połączony łańcuch.



Może być użyteczna, gdy:

- często pracujesz z początkiem lub końcem struktury;
- potrzebujesz zachowania listy powiązanej.

```
List<String> dane = new LinkedList<>();
```

OBIE:

- implementują `List`;
- mają `add()`;
- mają `remove()`;
- mają `get()`;
- pozwalają przechowywać duplikaty.

WAŻNA WSKAZÓWKA DLA POCZĄTKUJĄCYCH

Jeśli nie masz szczególnego powodu — `ArrayList` jest zwykle prostym wyborem startowym.

SET

`Set` i `HashSet` – kolekcja bez powtarzających się elementów.

NAJWAŻNIEJSZA CECHA

Nie przechowuje **duplikatów** zgodnie z zasadami porównania elementów.

"Java" | "Python" | "Java" ✗

↓

"Java" | "Python"

TWORZENIE

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

Set<String> jezyki = new HashSet<>();
```

Używamy interfejsu `Set` i jego najpopularniejszej implementacji `HashSet`.

DODAWANIE

```
jezyki.add("Java");
jezyki.add("Python");
jezyki.add("Java");
```

Wynik: "Java" | "Python"

Drugi raz dodajemy "Java", ale w zbiorze pojawi się tylko raz.

SPRAWDZENIE ELEMENTU

```
jezyki.contains("Java");
```

Wynik: `true`

Metoda `contains()` sprawdza, czy dany element znajduje się w zbiorze.

USUWANIE

```
jezyki.remove("Python");
```

Metoda `remove()` usuwa element ze zbioru. Jeśli element nie istnieje, nie zgłosi błędu.

WAŻNE

`Set` nie działa jak lista indeksowana. Nie używamy zwykle:

```
jezyki.get(0);
```

✗

`Set` nie ma indeksów i nie gwarantuje określonej kolejności elementów (w przypadku **HashSet** kolejność jest nieokreślona).

GDZIE SIĘ PRZYDAJE?

- ✓ unikalne adresy;
- ✓ tagi;
- ✓ identyfikatory;
- ✓ zestaw unikalnych nazw.

LIST vs SET

List	Set
• kolejność i pozycje;	• unikalność;
• może przechowywać duplikaty;	• nie przechowuje duplikatów;
• dostęp przez indeks (np. `get(0)`).	• brak indeksów (i zwykle brak określonej kolejności).

KLUCZ → WARTOŚĆ

`Map` i `HashMap` — przechowuj dane jako klucz i wartość.

IDEA

- Każda wartość jest przechowywana pod kluczem.
- Klucz identyfikuje wartość i pozwala ją później odczytać.

Przykład:

"Anna"	→	25
"Jan"	→	31
"Ola"	→	22

TWORZENIE

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

Map<String, Integer> wiek =
    new HashMap<>();
```

Tworzymy obiekt Map (najczęściej HashMap), który przechowuje pary klucz → wartość.

DODAWANIE DANYCH

```
wiek.put("Anna", 25);
wiek.put("Jan", 31);
```

Metoda put() dodaje parę klucz → wartość do mapy.

POBIERANIE WARTOŚCI

```
wiek.get("Anna");
```

Wynik:

25

Klucz	Wartość
"Anna"	25
"Jan"	31
"Ola"	22

SPRAWDZANIE KLUCZA

```
wiek.containsKey("Anna");
```

Metoda containsKey() sprawdza, czy dany klucz istnieje w mapie.

USUWANIE

```
wiek.remove("Jan");
```

Metoda remove() usuwa parę klucz → wartość o podanym kluczu.

KLUCZ POWINIEN IDENTYFIKOWAĆ WARTOŚĆ

- ID → użytkownik;
- kod produktu → cena;
- kraj → stolica;
- nazwa → wynik.

Klucz	Wartość
1	użytkownik
A123	cena
PL	Warszawa
test	85

WAŻNA INFORMACJA

Map nie przechowuje pojedynczych elementów — przechowuje **pary klucz → wartość.**

HASHMAP

Najważniejsze operacje na `HashMap`.

PUT()

Dodaje lub aktualizuje wartość.

```
punkty.put("Anna", 90);
```

Po wykonaniu:

Klucz	Wartość
"Anna"	90

GET()

Pobiera wartość dla podanego klucza.

```
punkty.get("Anna");
```

Wynik:

90

Klucz	Wartość
"Anna"	90
"Jan"	75

GETORDEFAULT()

Zwraca wartość dla podanego klucza lub wartość domyślną, jeśli klucz nie istnieje.

```
punkty.getOrDefault("Jan", 0);
```

Jeśli klucz nie istnieje:

0

Klucz	Wartość
"Anna"	90

CONTAINSKEY()

Sprawdza, czy mapa zawiera dany klucz.

```
punkty.containsKey("Anna");
```

Wynik:

true

Klucz	Wartość
"Anna"	90
"Jan"	75

REMOVE()

Usuwa element o podanym kluczu.

```
punkty.remove("Anna");
```

Po wykonaniu:

Klucz	Wartość
"Jan"	75

ITEROWANIE

Możemy przejść przez wszystkie elementy mapy.

```
for (String klucz : punkty.keySet()) {
  System.out.println(
    klucz + ": " + punkty.get(klucz)
  );
}
```

DZIAŁANIE MAPY

KLUCZ

↓

wyszukaj w mapie

↓

WARTOŚĆ

UWAGA!

Ponowne użycie tego samego klucza może zastąpić poprzednią wartość.

Klucz	Wartość
"Anna"	90

→

Klucz	Wartość
"Anna"	100

LIST vs SET vs MAP

Wybierz właściwą strukturę.

LIST

Myśl: kolejność

- elementy mają pozycje;
- duplikaty dozwolone;
- dostęp przez indeks.

Przykład:

```
["Java", "Java", "Python"]
```

Używaj gdy:

- kolejność ma znaczenie;
- potrzebujesz listy elementów.

SET

Myśl: unikalność

- brak duplikatów;
- brak klasycznego indeksowania.

Przykład:

```
{"Java", "Python"}
```

Używaj gdy:

- interesują Cię unikalne elementy.

MAP

Myśl: klucz → wartość

"Java" → 150

Przykład:

```
"Java" → 150
```

Używaj gdy:

- chcesz odnajdywać dane po kluczu.

KTÓRĄ STRUKTURĘ WYBRAĆ?



Dobór struktury danych wpływa na **prostotę całego programu.**

COLLECTIONS

Sortowanie i gotowe operacje na listach.

SORTOWANIE

Metoda `sort()` sortuje elementy listy w kolejności rosnącej (naturalnej).

```
Collections.sort(lista);
```

Przed: | 5 | 2 | 8 | 1 |

Po: | 1 | 2 | 5 | 8 |

IMPORT

Aby używać klasy Collections, trzeba ją zaimportować.

```
import java.util.Collections;
```

Klasa Collections znajduje się w pakiecie `java.util`.

ODWRÓCENIE KOLEJNOŚCI

Metoda `reverse()` odwraca kolejność elementów na liście.

```
Collections.reverse(lista);
```

Przed: | 1 | 2 | 3 | 4 |

Po: | 4 | 3 | 2 | 1 |

MINIMUM

Metoda `min()` zwraca najmniejszy element listy.

```
Collections.min(lista);
```

Dla listy: | 5 | 2 | 8 | 1 |

Wynik: | 1 |

MAXIMUM

Metoda `max()` zwraca największy element listy.

```
Collections.max(lista);
```

Dla listy: | 5 | 2 | 8 | 1 |

Wynik: | 8 |

TASOWANIE

Metoda `shuffle()` losowo przestawia elementy listy.

```
Collections.shuffle(lista);
```

Przed: | A | B | C | D |

shuffle

Po: | C | A | D | B |



WAŻNE ROZRÓŻNIENIE

Collection / List / Set
→ struktury i interfejsy

- to interfejsy reprezentujące kolekcje;
- określają, jakie operacje są dostępne;
- m.in. List pozwala przechowywać duplikaty, a Set ich nie.

Collections
→ klasa z pomocniczymi metodami

- to klasa z gotowymi metodami;
- działa na obiektach implementujących interfejsy kolekcji;
- zawiera metody takie jak `sort()`, `reverse()`, `min()`, `max()`, `shuffle()` i inne.

PODSUMOWANIE

- Collections to klasa z gotowymi metodami do pracy z kolekcjami.
- Umożliwia sortowanie, odwracanie, tasowanie, oraz znajdowanie minimum i maksimum.
- Działa na listach i innych kolekcjach z pakietu `java.util`.

Collections daje gotowe narzędzia do pracy z kolekcjami.

79. <T>

Generyki – podstawy.

GDZIE JUŻ JE WIDZIAŁEŚ?

Generyki są używane w wielu klasach w bibliotece Java, m.in.:

```
List<String> imiona = new ArrayList<>();
```
Lista tekstów.

```
List<Integer> liczby = new ArrayList<>();
```
Lista liczb.

```
Map<String, Integer> oceny = new HashMap<>();
```
Mapa par klucz-wartość.

CO MÓWIĄ GENERYKI?

Informują kompilator, jaki typ obiektu ocekujemy przechowywać lub używać.

```
List<String> imiona = new ArrayList<>();
```

Ta lista oczekuje elementów typu:

String

a nie na przykład:

Integer

DLACZEGO TO WAŻNE?

- większe bezpieczeństwo typów;
- mniej błędów;
- czytelniejszy kod;
- mniej ręcznych konwersji.

CO OZNACZA `T`?

Litera `T` to nazwa parametru typu. Oznacza po prostu **„Type"** (typ).

T → **Type**

Można używać także innych nazw, np. E (Element), K (Key), V (Value), ale `T` jest najczęściej spotykane.

BEZPIECZEŃSTWO

Dzięki generykom kompilator sprawdza, czy dodawane elementy mają odpowiedni typ.

```
imiona.add("Anna");
```
Poprawnie! Zgadza się z List<String>.

```
imiona.add(123);
```
Błąd! Nie pasuje do List<String>.

NA TYM ETAPIE ZAPAMIĘTAJ

GENERYK =
KOD, KTÓRY MOŻE PRACOWAĆ Z RÓŻNYMI TYPAMI W KONTROLOWANY SPOSÓB

W kolejnych mapach zobaczysz, jak tworzyć własne klasy i metody generyczne.

Własna klasa generyczna – twórz kod działający z różnymi typami

1. Podstawowy zapis

Klasa generyczna ma parametr typu w nawiasach ostrych <>.

```
class Pudelko<T> {
    T wartosc;
}
```



T oznacza typ, który określimy później.

2. Użycie ze String

Tworzymy obiekt klasy generycznej dla typu String.

```
Pudelko<String> tekst =
    new Pudelko<>();
```



3. Użycie z Integer

Tworzymy obiekt klasy generycznej dla typu Integer.

```
Pudelko<Integer> liczba =
    new Pudelko<>();
```



KLASA GENERYCZNA

Jeden kod, wiele typów

4. Ta sama klasa – różne typy

Ta sama klasa generyczna może być używana z różnymi typami danych.

`Pudelko<String>` → tekst



`Pudelko<Integer>` → liczby



`Pudelko<Double>` → wartości zmiennoprzecinkowe



5. Dlaczego warto?

- Jeden kod działa z wieloma typami.
- Bezpieczeństwo typów.
- Mniej powtarzania.
- Łatwiejsze rozwijanie kodu.

6. Co oznacza <T>?

T jest symbolem typu, który zostanie określony podczas użycia klasy.



Generyki pozwalają pisać elastyczny kod bez rezygnowania z kontroli typów.

Metody generyczne – jedna metoda dla różnych typów

1. Przyklad

Metoda generyczna z parametrem typu <T> w nawiasach ostrych.

```
static <T> void pokaz(T wartosc) {
    System.out.println(wartosc);
}
```

2. Użycie

Tę samą metodę możemy wywołać dla różnych typów danych.

```
pokaz("Java");
pokaz(123);
pokaz(3.14);
```

String, Integer, Double → ta sama metoda

3. Gdzie znajduje się <T>?

Parametr typu metody znajduje się przed typem zwracanym.

```
static <T> void pokaz(...)
```

<T> przed typem zwracanym deklaruje parametr typu metody.

METODA <T>

4. Co zyskujemy?

- Ponowne wykorzystanie kodu.
- Mniej duplikacji.
- Kontrolę typów.
- Bardziej uniwersalny kod.

5. Przyklad z wartością zwracaną

Metoda generyczna, która zwraca wartość typu T.

```
static <T> T zwroc(T wartosc) {
    return wartosc;
}
```

6. Prosta idea

Jak działa metoda generyczna?

Dane wejściowe typu T → Metoda pracuje z T → Może zwrócić T

Generyczna metoda nie musi być przywiązana do jednego konkretnego typu.

Sortowanie list — liczby, tekst i własny porządek

1. Lista liczb

Tworzymy listę liczb całkowitych w określonej kolejności.

```
List<Integer> liczby =
    new ArrayList<>(List.of(5, 2, 8, 1));
```



2. Sortowanie rosnące

Sortujemy listę liczb w kolejności rosnącej (od najmniejszej).

```
Collections.sort(liczby);
```

Wynik: `[1, 2, 5, 8]`



3. Odwrócona kolejność

Sortujemy listę liczb w odwrotnej kolejności (od największej).

```
Collections.sort(
    liczby,
    Collections.reverseOrder()
);
```

Wynik: `[8, 5, 2, 1]`



4. Tekst też można sortować

Sortujemy listę napisów (String) w kolejności alfabetycznej.

```
List<String> imiona =
    new ArrayList<>(List.of(
        "Ola", "Anna", "Jan"
    ));
Collections.sort(imiona);
```



Wynik: `["Anna", "Jan", "Ola"]`

SORTOWANIE

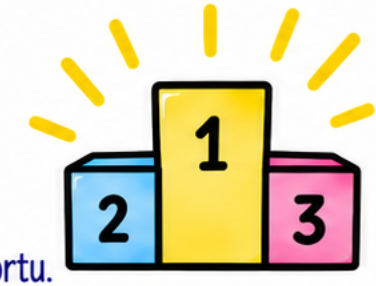
5. Ważna idea

Sortowanie zmienia kolejność elementów listy.



6. Gdzie się przydaje?

- Ranking.
- Ceny.
- Wyniki.
- Nazwy.
- Dane do raportu.



Sortowanie pomaga uporządkować dane przed dalszym przetwarzaniem lub prezentacją.

Podsumowanie – listy, kolekcje i generyki

1. List

- Kolejność.
- Indeksy.
- Duplikaty.



2. ArrayList

- Elastyczny rozmiar.
- Częsty wybór domyślny.



3. LinkedList

Alternatywna implementacja List.



4. Set

Unikalne elementy.



5. Map

Klucz → wartość.

klucz	wartość
"id"	1
"nazwa"	"Java"
"aktywny"	true



COLLECTIONS FRAMEWORK

6. Collections

- Sortowanie.
- Odwracanie.
- Min / max.
- Tasowanie.



7. Generyki

```
List<String>
Map<String, Integer>
```

Zapewniają kontrolę typów.



8. Własne <T>

Pozwalają tworzyć uniwersalne klasy i metody.



Potrzebujesz indeksów i kolejności? → **List** | Potrzebujesz unikalności? → **Set** | Potrzebujesz klucza? → **Map**

Teraz potrafisz przechowywać i organizować dane dynamicznie. Następny krok: klasy i obiekty.

MODUŁ 06

KLASY, OBIEKTY I PODSTAWY OOP

Połącz dane i zachowania w struktury, które opisują prawdziwe obiekty.

1. klasa

Szablon, który określa pola i metody obiektów.



2. obiekt

Konkretny egzemplarz klasy z własnymi danymi.



3. pole

Przechowuje dane obiektu (np. liczby, teksty, inne obiekty).



4. metoda

Określa zachowania obiektu – operacje, które może wykonać.



OOP

5. konstruktor

Fragment klasy wywoływany przy tworzeniu obiektu; inicjalizuje jego pola.



6. this

Odwołuje się do bieżącego obiektu, np. do jego pól lub metod.



7. enkapsulacja

Ukrywa wewnętrzne dane obiektu i udostępnia je przez publiczne metody (gettery i settery).



KLASA

↓

na jej podstawie powstaje

OBIEKT

Klasa: Samochód

- kolor
- marka
- jedz()



Programowanie obiektowe – jak myśleć w OOP

1. Podstawowa idea

Łączymy:

DANE + **ZACHOWANIA**

w jeden obiekt.

2. Przykład – samochód

Dane	Zachowania
• marka	• jedz()
• kolor	• zatrzymajSie()
• prędkość	• przyspiesz()



3. Klasa

To szablon lub projekt obiektów.

KLASA = PLAN



OOP

4. Obiekt

To instancja utworzona na podstawie klasy.

OBIEKT = KONKRETNY EGZEMPLARZ



5. Przykład

Klasa: Samochod

Obiekty:

- czerwone BMW
- biały Ford
- czarna Toyota



6. Dlaczego OOP?

- Porządek w kodzie.
- Ponowne wykorzystanie kodu.
- Modelowanie problemów.
- Rozwijanie większych systemów.

Klasa opisuje. Obiekt istnieje i posiada konkretny stan.

Klasa — szablon dla obiektów

1. Prosta klasa

Definiujemy klasę, która opisuje ucznia.

```
class Uczen {
   String imie;
   int wiek;

   void przedstawSie() {
      System.out.println(
        "Mam na imię " + imie
      );
   }
}
```



2. Pola

Przechowują dane obiektu.

```
String imie;
int wiek;
```



3. Metody

Opisują zachowania obiektu.

```
void przedstawSie()
```



CLASS

4. Jak myśleć o klasie?



Dane: imię, wiek

Działanie: przedstaw się

5. Klasa sama nie jest jeszcze konkretnym uczniem

To dopiero definicja.



6. Dopiero obiekt otrzyma konkretne dane

`class Uczen` → `new Uczen()` → konkretny obiekt



Klasa grupuje informacje i zachowania dotyczące jednego typu obiektu.

Obiekt – utwórz konkretną instancję klasy

1. Mamy klasę

Definiujemy klasę Uczen z polami i metodą.

```
class Uczen {
   String imie;
   int wiek;
   void przedstawSie() {
      System.out.println(
        "Mam na imię " + imie
      );
   }
}
```



2. Tworzymy obiekt

Tworzymy nową instancję klasy za pomocą słowa new.

```
Uczen uczen1 = new Uczen();
```

Tworzy nową instancję.



3. Nadajemy dane

Ustawiamy wartości pól obiektu.

```
uczen1.imie = "Anna";
uczen1.wiek = 20;
```



OBIEKT

4. Drugi obiekt

Tworzymy drugi obiekt i nadajemy mu dane.

```
Uczen uczen2 = new Uczen();
uczen2.imie = "Jan";
uczen2.wiek = 24;
```



uczen1	uczen2
Anna	Jan
20	24

Oba obiekty powstały na podstawie tej samej klasy.

5. Każdy obiekt może mieć własny stan

Oznacza to własne wartości przechowywane w polach.



6. Obiekt może korzystać z metod klasy

Obiekt może wywoływać metody zdefiniowane w klasie.

```
uczen1.przedstawSie();
```



Jedna klasa może tworzyć wiele różnych obiektów.

Stan i zachowanie obiektu – pola oraz metody

STAN – POLA

Obiekt Konto Bankowe ma pola, które przechowują jego stan.

```
String wlasciciel;
double saldo;
```



Określają:

- Kto jest właścicielem;
- ile pieniędzy znajduje się na koncie.



ZACHOWANIE – METODY

Obiekt Konto Bankowe ma metody, które definiują jego zachowanie.

```
void wplac(double kwota) {
    saldo += kwota;
}
```




```
void pokazSaldo() {
    System.out.println(saldo);
}
```



1. Stan może się zmieniać

Jeśli saldo = 100 i wywołamy wplac(50), wtedy saldo = 150.

saldo = 100 → wplac(50) → saldo = 150

2. Metoda może zmieniać pola obiektu

METODA → operuje na → STANIE OBIEKTU

3. Dlaczego to ważne?

Obiekt nie jest tylko zbiorem wartości. Zawiera także logikę związaną z tymi wartościami.



OBIEKT = STAN + ZACHOWANIE

Konstruktor — przygotuj obiekt już podczas tworzenia

1. Co to jest?

Specjalny fragment klasy wywoływany podczas tworzenia obiektu.



2. Nazwa konstruktora

Ma taką samą nazwę jak klasa.

```
class Uczen {
   Uczen(...) {
      // konstruktor
   }
}
```



3. Przykład

Pełna definicja klasy z konstruktorem.

```
class Uczen {
   String imie;
   int wiek;

   Uczen(String imie, int wiek) {
      this.imie = imie;
      this.wiek = wiek;
   }
}
```



KONSTRUKTOR

4. Tworzenie obiektu

Konstruktor jest wywoływany automatycznie podczas tworzenia obiektu.

```
Uczen uczen =
      new Uczen("Anna", 20);
```



5. Dlaczego konstruktor jest przydatny?

- Nadaje danym wartości początkowe.
- Pomaga uniknąć niekompletmych obiektów.
- Upraszcza tworzenie obiektów.
- Skupia reguły inicjalizacji.



6. Konstruktor nie ma typu zwracanego

Nie dodajemy typu zwracanego, nawet `void`.

```
void Uczen(...) { ...}
```

NIE

Konstruktor ≠ zwykła metoda

Dobrze zaprojektowany konstruktor pomaga tworzyć obiekty od razu w poprawnym stanie.

this — odwołaj się do bieżącego obiektu

1. Co oznacza?

this = bieżący obiekt, na którym pracuje metoda lub konstruktor.



2. Najczęstsze zastosowanie

```
class Uczen {
    String imie;
    Uczen(String imie)
    {
        this.imie = imie;
    }
}
```

this.imie → pole obiektu

imie → parametr konstruktora

3. Czytaj to tak

`this.imie = imie;`

pole obiektu

parametr konstruktora

Przypisz wartość parametru imie do pola imie tego obiektu.

this

4. Możesz też używać this w metodach

```
void pokazImie() {
    System.out.println(this.imie);
}
```



5. Po co to robić?

- Usuwa niejednoznaczność.
- Poprawia czytelność.
- Często pojawia się w konstruktorach.
- Jasno wskazuje pola obiektu.

6. Prosta zasada

this. coś = coś należącego do bieżącego obiektu



this nie oznacza klasy — oznacza konkretny aktualny obiekt.

Przeciążanie konstruktorów – kilka sposobów tworzenia obiektu

1. Jeden konstruktor

```
class Uczen {
  String imie;
  int wiek;

  Uczen(String imie, int wiek) {
    this.imie = imie;
    this.wiek = wiek;
  }
}
```



2. Drugi konstruktor

Dodajemy go w tej samej klasie.

```
Uczen(String imie) {
  this.imie = imie;
  this.wiek = 0;
}
```



3. Teraz możemy tworzyć obiekty na dwa sposoby

```
Uczen a = new Uczen("Anna", 20);
```

lub:

```
Uczen b = new Uczen("Jan");
```



WIELE KONSTRUKTORÓW

4. Co musi się różnić?

Lista parametrów.

Może zmienić się:

- liczba parametrów;
- typy parametrów;
- kolejność typów.



5. Dlaczego warto?

- Elastyczne tworzenie obiektów.
- Wartości domyślne.
- Wygodniejsze API.



6. this(...)

Jeden konstruktor może wywołać drugi.

Alternatywna wersja konstruktora z bloku 2:

```
Uczen(String imie) {
  this(imie, 0);
}
```

Uczen("Jan") → this("Jan", 0) → konstruktor z dwoma parametrami

Przeciążanie konstruktorów = ta sama klasa, różne sposoby rozpoczęcia życia obiektu.

public, private i dostęp do elementów klasy

1. public

Pozwala korzystać z elementu z innych klas, jeśli sama klasa jest dostępna.

```
public void pokazDane() {
    ...
}
```



2. private

Element dostępny wewnątrz klasy.

```
private double saldo;
```



3. Po co ograniczać dostęp?

- Ochrona stanu wewnętrznego.
- Mniej przypadkowych zmian.
- Czytelniejsze API.
- Większa kontrola nad danymi.



DOSTĘP

4. Przykład

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

POZA KLASĄ	→ `konto.wplac(100)`	✓	dozwolone
POZA KLASĄ	→ `konto.saldo`	✗	brak bezpośredniego dostępu

5. Jak o tym myśleć?

`private`	→	wnętrze obiektu
`public`	→	to, co obiekt udostępnia na zewnątrz



6. Uwaga

Java ma także inne poziomy dostępu.
Na tym etapie poznajemy przede wszystkim:

`public` kontra `private`



Nie wszystko w klasie powinno być dostępne bezpośrednio z zewnątrz.

Enkapsulacja – chroń stan obiektu

1. Idea

Nie pozwalamy, aby każdy fragment programu dowolnie zmieniał pola obiektu.



2. Zamiast tego

- Pola często są `private`.
- Dostęp odbywa się przez kontrolowane metody.



3. Bez enkapsulacji

Gdy pole jest publiczne:

```
konto.saldo = -100000;
```

Dowolna zmiana stanu.



4. Z enkapsulacją

```
konto.wplac(100);
```

Metoda może sprawdzać reguły.



ENKAPSULACJA

5. Przykład

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



6. Co zyskujemy?

- Ochronę danych.
- Walidację.
- Reguły w jednym miejscu.
- Mniejsze ryzyko niepoprawnego stanu.
- Łatwiejsze utrzymanie kodu.

PRYWATNE DANE + **PUBLICZNE OPERACJE** → **KONTROLOWANY OBIEKT**

Enkapsulacja nie polega tylko na private.
Chodzi o kontrolowanie sposobu korzystania ze stanu obiektu.

Gettery i settery – kontrolowany dostęp do pól

GETTER

Służy do odczytu wartości.

```
private String imie;
public String getImie() {
    return imie;
}
```



pole prywatne → getImie() → odczyt

GETTERY I SETTERY

SETTER

Służy do zmiany wartości.

```
public void setImie(String imie) {
    this.imie = imie;
}
```



nowa wartość → setImie() → pole obiektu

1. Setter może sprawdzać dane

```
private int wiek;
public void setWiek(int wiek) {
    if (wiek >= 0) {
        this.wiek = wiek;
    }
}
```



2. Dlaczego to lepsze niż publiczne pole?

Klasa może kontrolować sposób zmiany wartości.



3. Nie każde pole musi mieć setter

Jeśli wartości nie należy swobodnie zmieniać, nie musimy udostępniać settera.



4. Szybkie zapamiętanie

get...() → pobierz

set...() → ustaw



Getter daje dostęp do danych. Setter może kontrolować ich zmianę.

Referencje – zmienna wskazuje na obiekt

1. Tworzymy obiekt

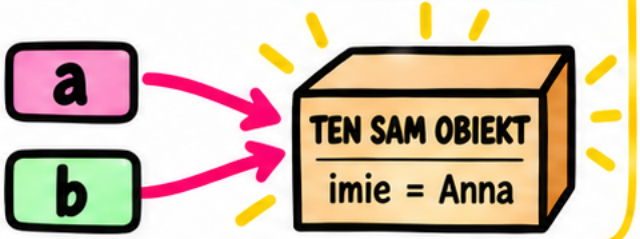
Klasa Uczen ma konstruktor Uczen(String) oraz metody getImie() i setImie(...).

```
Uczen a = new Uczen("Anna");
```



2. Przypisujemy referencję

```
Uczen b = a;
```



3. Nie powstał drugi obiekt

Instrukcja Uczen b = a; kopiuje referencję.



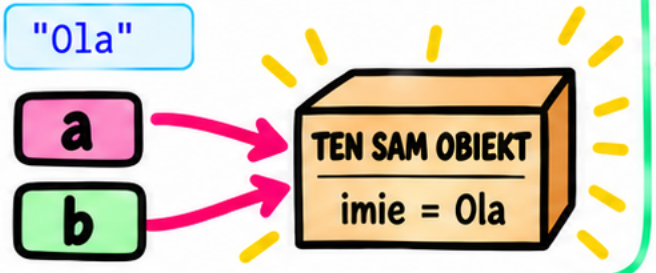
REFERENCJA

4. Zmiana przez b

```
b.setImie("Ola");
```

a.getImie() również zwraca:

```
"Ola"
```

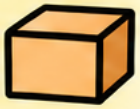


5. Dlaczego?

Zmienne a i b wskazują na ten sam obiekt.



6. Obiekt kontra referencja



OBIEKT → konkretne dane w pamięci



REFERENCJA → sposób dotarcia do obiektu

Dwie zmienne mogą wskazywać na ten sam obiekt.

null – gdy referencja nie wskazuje na obiekt

1. Co oznacza?

Zmienna referencyjna nie wskazuje obecnie na żaden obiekt.

```
Uczen uczen = null;
```

uczen → BRAK OBIEKTU

2. Co się stanie tutaj?

```
uczen.getImie();
```

Próba użycia nieistniejącego obiektu.

3. Typowy efekt

`NullPointerException`

NULL

4. Jak sprawdzić?

```
if (uczen != null) {
    uczen.getImie();
}
```

5. Kiedy może pojawić się null?

- Pole referencyjne nie otrzymało jeszcze obiektu.
- Wyszukiwanie nie znalazło wyniku.
- Celowo oznaczono brak wartości.
- Metoda zwróciła null.

6. Dobra praktyka

Zawsze sprawdzaj, co null oznacza w danym kontekście. Nie traktuj go bez zastanowienia jak zwykłej wartości.

null ≠ pusty String

`null` → brak obiektu

`""` → istniejący String o długości 0

Zanim wywołasz metodę na możliwym null, upewnij się, że obiekt istnieje.

toString() – czytelna reprezentacja obiektu

1. Problem

Mamy obiekt:

```
Uczen uczen = new Uczen("Anna", 20);
```

Chcemy czytelnie wyświetlić jego dane.



2. Definiujemy toString()

```
@Override
public String toString() {
    return "Uczen{imie='" + imie
        + "', wiek=" + wiek + "}";
}
```

3. Teraz

```
System.out.println(uczen);
```

Wynik:

```
Uczen{imie='Anna', wiek=20}
```



toString()

4. Po co?

- Debugowanie.
- Logi.
- Podgląd danych.
- Czytelniejsze wypisywanie obiektów.

5. @Override

Adnotacja informuje, że nadpisujemy odziedziczoną metodę.
Więcej przy dziedziczeniu.

@Override

6. Dobra reprezentacja tekstowa

Pokazuje przydatne informacje i nie ujawnia niepotrzebnie danych wrażliwych.

toString() pomaga szybko zobaczyć, co znajduje się w obiekcie.

Praktyka – zbuduj klasę Produkt

1. Pola

Klasa Produkt ma trzy pola z odpowiednimi typami.

```
private String nazwa;
private double cena;
private int ilosc;
```



2. Konstruktor

Konstruktor przyjmuje wartości dla pól i przypisuje je do obiektu.

```
Produkt(String nazwa,
        double cena, int ilosc) {
    this.nazwa = nazwa;
    this.cena = cena;
    this.ilosc = ilosc;
}
```

3. Metoda obliczająca wartość

Metoda obliczWartosc() zwraca wartość magazynową produktu.

```
public double obliczWartosc() {
    return cena * ilosc;
}
```



4. Getter

Metoda getNazwa() zwraca nazwę produktu.

```
public String getNazwa() {
    return nazwa;
}
```



PRODUKT

Pola, konstruktor i metody tworzą klasę Produkt.

5. Tworzenie obiektu

Tworzymy obiekt klasy Produkt, podając wartości w konstruktorze.

```
Produkt produkt =
    new Produkt("Klawiatura", 199.99, 2);
```



6. Użycie

Wywołujemy metodę obliczWartosc() dla utworzonego obiektu.

```
System.out.println(
    produkt.obliczWartosc()
);
```





7. Co ćwiczysz?

klasa • obiekt • private • konstruktor
this • metoda • getter • stan + zachowanie



SPRÓBUJ SAM

Dodaj metodę dodajDoMagazynu(int sztuki), która zwiększa ilosc.

Podsumowanie – klasy, obiekty i podstawy OOP

1. Klasa

Szablon opisujący dane i zachowania.



2. Obiekt

Konkretna instancja klasy.



3. Pole

Przechowuje stan obiektu.



4. Metoda

Opisuje zachowanie.



OOP – PODSTAWY

KLASA → new → OBIEKT

OBIEKT = STAN + ZACHOWANIE

5. Konstruktor

Przygotowuje obiekt podczas tworzenia.



6. this

Odnosi się do bieżącego obiektu.



7. private

Chroni dane wewnętrzne.



8. Enkapsulacja

Kontroluje sposób dostępu do stanu.



9. Gettery i settery

Udostępniają kontrolowany odczyt i zmianę danych.



10. Referencja

Wskazuje na obiekt.



Najważniejsza zmiana sposobu myślenia

Dotąd: „Jakie instrukcje ma wykonać program?”

Teraz także: „Jakie obiekty istnieją w problemie i za co każdy z nich odpowiada?”

Co dalej?

DZIEDZICZENIE • INTERFEJSY • POLIMORFIZM

W kolejnym module zobaczysz, jak klasy mogą współpracować i dzielić wspólne zachowania.

MODUŁ 07

DZIEDZICZENIE, INTERFEJSY I POLIMORFIZM

Buduj elastyczne klasy, które współpracują przez wspólne zachowania.

OOP – RELACJE

- extends
- klasa bazowa
- klasa potomna
- super
- nadpisywanie
- polimorfizm
- abstract
- interface

ZWIERZĘ



- PIES
- KOT



INTERFEJS: LATANIE



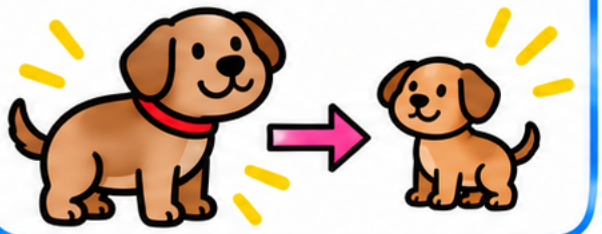
- Ptak
- Samolot
- Dron



Dziedziczenie – jedna klasa może rozszerzać drugą

1. Idea

Klasa potomna przejmuje część zachowań klasy bazowej.



2. Przykład klasy bazowej

```
class Zwierze {
  void jedz() {
    System.out.println("Zwierzę je");
  }
}
```



3. Klasa potomna

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



DZIEDZICZENIE

4. Co może zrobić Pies?

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



5. Słowo kluczowe extends

Oznacza dziedziczenie między klasami.

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



6. Dlaczego używać?

- Ponowne wykorzystanie kodu.
- Uporządkowana struktura klas.
- Modelowanie relacji „jest rodzajem”.



Pies jest zwierzęciem, dlatego może dziedziczyć wspólne zachowania.

Relacja „IS-A” – kiedy dziedziczenie ma sens

1. Jak czytać?

Słowo kluczowe extends oznacza relację IS-A.

```
Pies extends Zwierze
```

oznacza:

„Pies jest zwierzęciem.”



2. Dobre przykłady

- Pies → Zwierzę
- Samochód → Pojazd
- Manager → Pracownik
- Kot → Zwierzę



3. Zły przykład

```
Silnik extends Samochod
```

Silnik nie jest samochodem.

Poprawna relacja: samochód ma silnik.

IS-A

4. Dziedziczenie kontra kompozycja

IS-A → dziedziczenie

HAS-A → kompozycja

5. Przykład HAS-A

```
class Silnik {}

class Samochod {
   Silnik silnik;
}
```



6. Dlaczego to ważne?

Źle dobrana relacja sprawia, że kod staje się sztuczny i trudny w utrzymaniu.

Pytanie kontrolne: „Czy obiekt klasy potomnej naprawdę jest rodzajem klasy bazowej?”

super — odwołaj się do klasy bazowej

1. Konstruktor klasy bazowej

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



2. Klasa potomna

```
class Uczen extends Osoba {
  int klasa;
  Uczen(String imie, int klasa) {
      super(imie);
      this.klasa = klasa;
  }
}
```



3. Co robi super(imie)?

Wywołuje konstruktor klasy bazowej.

`new Uczen("Anna", 2)`

↓

`super("Anna")`

↓

konstruktor klasy Osoba



super

4. super może też odwołać się do metody rodzica

W metodzie klasy Uczen:

```
super.pokazDane();
```



5. this kontra super

`this` → bieżący obiekt

`super` → dostępne elementy klasy bazowej



6. Ważne

W tym przykładzie najpierw wywołujemy `super(imie)`, potem przypisujemy `this.klasa`.



super pomaga zbudować część obiektu odziedziczoną z klasy bazowej.

Nadpisywanie metod – zmień zachowanie odziedziczonej metody

1. Klasa bazowa

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



2. Klasa potomna

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



3. Co się zmieniło?

- Ta sama nazwa.
- Te same parametry.
- Nowa implementacja.



OVERRIDING

4. @Override

Informuje, że nadpisujemy odziedziczoną metodę. Kompilator sprawdza poprawność nadpisania.



5. Przykład działania

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

Wynik: Hau!



6. Przeciążanie kontra nadpisywanie

OVERLOADING →	ta sama nazwa, różne parametry
OVERRIDING →	metoda odziedziczona, nowa implementacja

Nadpisywanie pozwala klasie potomnej zachowywać się po swojemu.

Polimorfizm – jeden typ, różne zachowania

1. Klasa bazowa

Klasa bazowa definiuje metodę, którą klasy potomne mogą nadpisać.

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



2. Dwie klasy potomne

Klasy potomne nadpisują metodę wydajDzwiek().

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



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



3. Wspólny typ

Obiekty klas potomnych mogą być przechowywane w zmiennej typu bazowego.

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



POLIMORFIZM

4. Ta sama metoda

Wywołujemy tę samą metodę na obu obiektach.

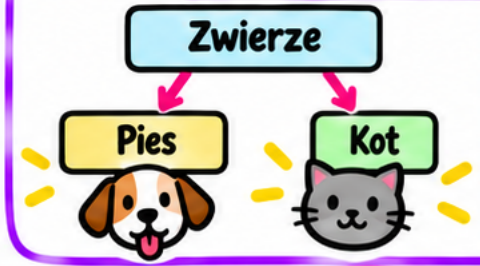
```
a.wydajDzwiek();
b.wydajDzwiek();
```

Wyniki:

```
Hau!
Miau!
```



5. Jak to rozumieć?



To samo wywołanie: wydajDzwiek() — różne zachowania.

6. Dlaczego to ważne?

- Elastyczność.
- Uniwersalny kod.
- Łatwiejsze dodawanie nowych typów.
- Mniej warunków zależnych od typu.



Polimorfizm = wspólny typ, różne konkretne zachowania.

Klasa abstrakcyjna — wspólna baza bez bezpośredniego obiektu

1. Tworzenie

Definiujemy klasę abstrakcyjną za pomocą słowa kluczowego abstract.

```
abstract class Figura {
    abstract double pole();
}
```



2. Co oznacza abstract?

Klasa abstrakcyjna opisuje ogólną koncepcję. Nie można bezpośrednio utworzyć jej obiektu.



3. Nie robimy

Nie możemy utworzyć obiektu klasy abstrakcyjnej.

```
Figura figura =
    new Figura();
```



4. Klasa potomna

Klasa potomna dziedziczy po klasie abstrakcyjnej i implementuje jej metody.

```
class Kwadrat extends Figura {
    double bok;

    Kwadrat(double bok) {
        this.bok = bok;
    }

    @Override
    double pole() {
        return bok * bok;
    }
}
```



ABSTRACT CLASS

5. Co daje klasa abstrakcyjna?

- Wspólną bazę.
- Wspólne pola.
- Gotowe metody.
- Metody abstrakcyjne wymagające implementacji w konkretnej klasie potomnej.



6. Jak o niej myśleć?

Figura = ogólna koncepcja

Kwadrat / Koło = konkretne wersje



Klasa abstrakcyjna mówi: „te klasy są ze sobą spokrewnione i mają wspólną podstawę".

Interfejs – umowa opisująca wspólne zachowanie

1. Przykład

Interfejs opisuje wspólne zachowanie, które muszią zrealizować klasy.

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

2. Implementacja

Klasa implementuje interfejs, dostarczając konkretną implementację metod.

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

3. Słowo kluczowe implements

Oznacza, że klasa realizuje kontrakt interfejsu.

implements

4. Inna klasa może implementować ten sam interfejs

Wiele klas może implementować ten sam interfejs, każda na swój sposób.

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

INTERFACE

Latajacy

Ptak

Dron

5. Co łączy Ptak i Dron?

Reprezentują różne rodzaje obiektów, ale oba udostępniają zachowanie lec().

6. Dlaczego interfejsy są ważne?

- Określają kontrakt.
- Pozwalają pracować z różnymi typami.
- Ułatwiają polimorfizm.
- Zmniejszają zależność od konkretnych klas.

Interfejs mówi „co potrafisz zrobić", a nie „czym jesteś".

Interfejs czy klasa abstrakcyjna?

KLASA ABSTRAKCYJNA

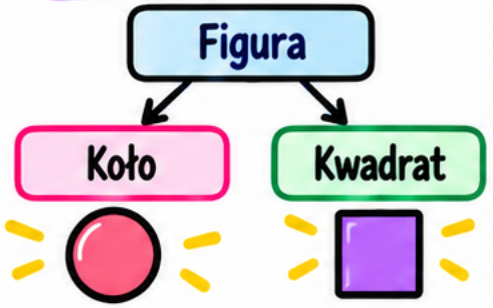
Myśl: wspólna baza

Może:

- mieć pola;
- mieć konstruktor;
- mieć zwykłe metody;
- mieć metody abstrakcyjne.

Sprawdza się, gdy klasy mają wspólne pola i należą do jednej rodziny.

Przykład:



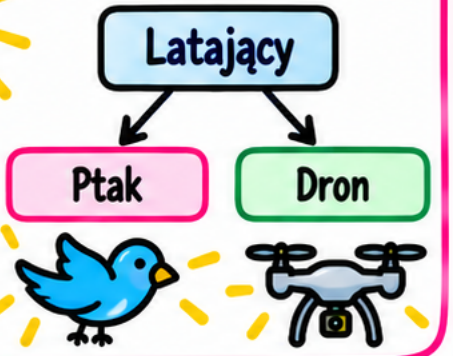
INTERFEJS

Myśl: wspólna umiejętność / kontrakt

Określa przede wszystkim oczekiwane zachowanie.

Sprawdza się, gdy różne typy mają oferować tę samą umiejętność.

Przykład:



Najprostsza heurystyka

„Czym jest?”	→	klasa bazowa / abstrakcyjna
„Co potrafi?”	→	interfejs

Ważne

To pomocna wskazówka na początek, nie sztywna reguła.

Klasa abstrakcyjna modeluje wspólną bazę. Interfejs modeluje wspólny kontrakt.

Polimorfizm przez interfejs – programuj do kontraktu

1. Interfejs

Interfejs określa kontrakt, który muszą spełniać implementacje.

```
interface Platnosc {
    void zaplac(double kwota);
}
```



2. Implementacja 1

Klasa Karta implementuje interfejs Platnosc.

```
class Karta implements Platnosc {
  @Override
  public void zaplac(double kwota) {
    System.out.println(
      "Płatność kartą");
  }
}
```

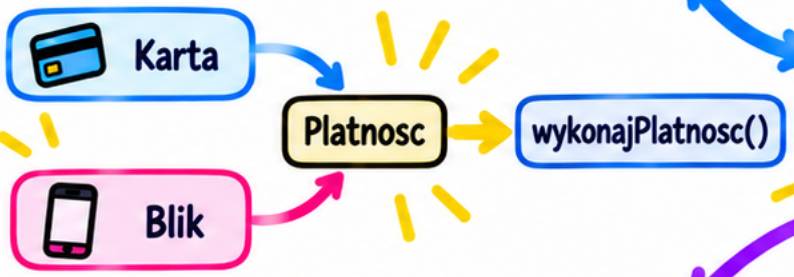


3. Implementacja 2

Klasa Blik implementuje interfejs Platnosc.

```
class Blik implements Platnosc {
  @Override
  public void zaplac(double kwota) {
    System.out.println(
      "Płatność BLIK");
  }
}
```





INTERFEJS + POLIMORFIZM

4. Wspólna metoda

Metoda pomocnicza w klasie:

```
static void wykonajPlatnosc(
    Platnosc platnosc, double kwota
) {
    platnosc.zaplac(kwota);
}
```



5. Użycie

Wywołujemy tę samą metodę dla różnych implementacji.

```
wykonajPlatnosc(new Karta(), 100);
wykonajPlatnosc(new Blik(), 100);
```



6. Najważniejsza korzyść

Metoda nie musi znać szczegółów każdej formy płatności. Korzysta tylko z kontraktu Platnosc.



7. Dlaczego to jest potężne?

- Łatwo dodać nowe implementacje.
- Mniej instrukcji if i switch.
- Mniejsza zależność od konkretnych klas.
- Łatwiejsze ponowne użycie kodu.

PROGRAMUJ DO INTERFEJSU, NIE DO KONKRETNEJ IMPLEMENTACJI

Wiele interfejsów – jedna klasa może mieć wiele umiejętności

1. Dwa kontrakty

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

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

2. Jedna klasa implementuje oba

```
class Dron implements Latajacy, Ladujacy {
    @Override
    public void lec() {
        System.out.println("Dron leci");
    }
    @Override
    public void laduj() {
        System.out.println("Dron się ładuje");
    }
}
```

3. Jak to czytać?

Dron potrafi latać i potrafi się ładować.



WIELE INTERFEJSÓW

4. Ważna różnica

Klasa w Javie może rozszerzać jedną klasę i implementować wiele interfejsów.

Nie może rozszerzać wielu klas jednocześnie.

5. Przykład nagłówka klasy

Sam nagłówek – metody jak w bloku 2.

```
class Dron extends Urzadzenie
        implements Latajacy, Ladujacy
```

Dziedziczy po Urzadzenie i realizuje dwa kontrakty.

6. Dlaczego to jest przydatne?

- Łączenie wielu umiejętności.
- Elastyczny projekt.
- Mniej sztucznych hierarchii.
- Łatwiejsze modelowanie zachowań.

Klasa może mieć jedną klasę bazową, ale realizować wiele kontraktów.

instanceof — sprawdź typ obiektu

1. Po co?

Sprawdza, czy obiekt jest instancją danego typu.



2. Przykład

```
Zwierze zwierze = new Pies();
if (zwierze instanceof Pies) {
  System.out.println("To jest pies");
}
```

3. Wynik

Wynik sprawdzenia instanceof:

true

Komunikat w konsoli:

To jest pies



instanceof

4. Gdzie może się przydać?

- Sprawdzanie konkretnego typu.
- Bezpieczniejsze rzutowanie.
- Obsługa różnych obiektów.



5. Przykład z rzutowaniem

```
if (zwierze instanceof Pies) {
   Pies pies = (Pies) zwierze;
   pies.szczekaj();
}
```

6. Uwaga projektowa

Wiele warunków instanceof może sygnalizować, że warto lepiej wykorzystać polimorfizm.

DOBRZE UŻYTE → pojedyncze sprawdzenie typu

NADUŻYTE → długie łańcuchy sprawdzeń typów



Najpierw korzystaj z polimorfizmu. instanceof stosuj, gdy naprawdę potrzebujesz znać konkretny typ.

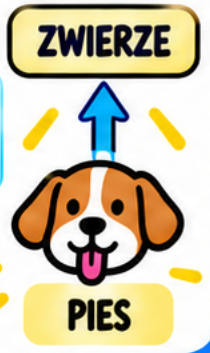
Rzutowanie obiektów – przechodzenie między typami w hierarchii

1. Upcasting

Typ potomny → typ bazowy.

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

Bezpieczne przejście do ogólniejszego typu, bez jawnego rzutowania.



2. Downcasting

Typ bazowy → typ potomny.

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



3. Dlaczego ostrożnie?

Nie każde Zwierze jest obiektem klasy Pies.

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

ClassCastException — błąd podczas działania programu.

CASTING

4. Jak się zabezpieczyć?

```
if (zwierze instanceof Pies) {
    Pies pies = (Pies) zwierze;
}
```



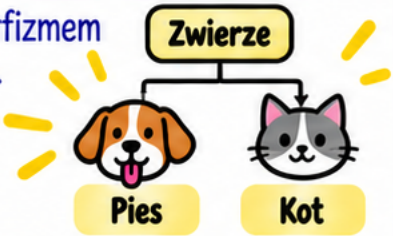
5. Szybkie zapamiętanie

UPCASTING → bardziej ogólny typ

DOWNCASTING → bardziej szczegółowy typ

6. Po co to wszystko?

Do pracy z polimorfizmem i hierarchiami klas.



Rzutowanie zmienia sposób traktowania referencji, nie sam obiekt.

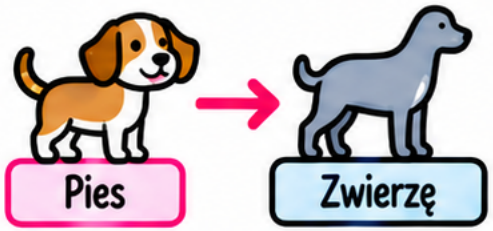
Nie rzutuj w dół bez pewności, jaki obiekt naprawdę znajduje się pod referencją.

Kompozycja czy dziedziczenie? „Ma” kontra „jest”.

DZIEDZICZENIE

Relacja: IS-A

Pies jest zwierzęciem.



Używaj, gdy:

- Istnieje prawdziwa relacja typu.
- Klasa potomna naprawdę jest odmianą klasy bazowej.

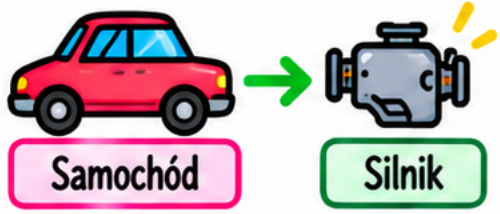
Przykład:

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

KOMPOZYCJA

Relacja: HAS-A

Samochód ma silnik.



Używaj, gdy:

- Istnieje relacja posiadania.
- Obiekt ma inny obiekt jako część składową.
- Nie ma prawdziwej relacji typu.

Przykład:

```
class Samochod {
    private Silnik silnik;
}
```

Dlaczego kompozycja jest ważna?

- Pozwala składać obiekty z innych obiektów.
- Może zmniejszać zależności.
- Daje większą elastyczność.
- Pomaga unikać sztucznych hierarchii.



Przykłady

- Komputer ma procesor.
- Dom ma adres.
- Zamówienie ma produkty.



Pytanie kontrolne

Czy zdanie brzmi naturalnie?

„X jest Y” → rozważ dziedziczenie

„X ma Y” → rozważ kompozycję





Nie używaj dziedziczenia tylko dla kilku metod. Relacja musi mieć sens.

Praktyka – system powiadomień z interfejsem

1. Interfejs

```
interface Powiadomienie {
    void wyslij(String tresc);
}
```

2. E-mail

```
class Email implements Powiadomienie {
    @Override
    public void wyslij(String tresc) {
      System.out.println("Email: " + tresc);
    }
}
```

3. SMS

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

4. Wspólna metoda

Metoda pomocnicza w klasie:

```
static void powiadom(
    Powiadomienie kanal, String tresc
) {
    kanal.wyslij(tresc);
}
```

SYSTEM POWIADOMIEŃ

EMAIL

SMS

POWIADOMIENIE

powiadom()

5. Użycie

```
powiadom(new Email(),
         "Twoje zamówienie wysłano");
powiadom(new Sms(), "Kod: 1234");
```

6. Co tutaj wykorzystujemy?

- Interfejs
- Implements
- @Override
- Polimorfizm
- Programowanie do kontraktu

7. Rozszerzenie

Możemy dodać klasę PushNotification implementującą Powiadomienie bez zmiany wspólnej metody powiadom().

Dobry kontrakt pozwala dodawać nowe implementacje bez przepisywania całej logiki.

Podsumowanie — dziedziczenie, interfejsy i polimorfizm

1 extends



Dziedziczenie po jednej klasie bazowej.

2 IS-A



Pomaga sprawdzić, czy dziedziczenie ma sens.

3 super



Odwołanie do konstruktora lub dostępnych elementów klasy bazowej.

4 @Override



Pozwala kompilatorowi sprawdzić nadpisanie metody.

ELASTYCZNE OOP

5 Polimorfizm



Wspólny typ + różne zachowania.

6 Klasa abstrakcyjna



Wspólna baza dla powiązanych klas.

7 Interfejs



Kontrakt opisujący zachowanie.

8 implements



Klasa realizuje kontrakt interfejsu.

9 Kompozycja



Obiekt zawiera inne obiekty.

IS-A → dziedziczenie



HAS-A → kompozycja



CAN-DO → często interfejs



Co dalej?



WYJĄTKI, PLIKI I OPERACJE NA DANYCH



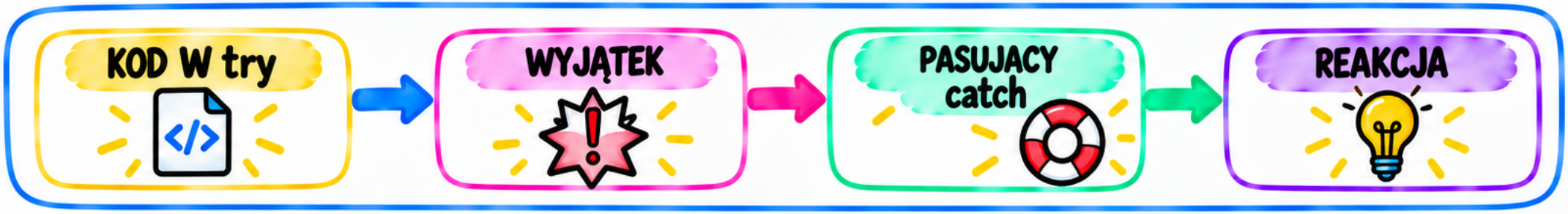
Teraz potrafisz budować klasy, które współpracują bez sztywnego uzależnienia od jednej konkretnej implementacji.

MODUŁ 08

WYJĄTKI, PLIKI I OPERACJE NA DANYCH

Obsługuj błędy, czytaj i zapisuj pliki oraz twórz programy odporniejsze na problemy.





Wyjątki – gdy podczas działania programu pojawia się problem

1. Co to jest?

Wyjątek informuje, że podczas działania programu wystąpił problem.

2. Przykłady sytuacji

- Dzielenie całkowite przez zero.
- Odwołanie do nieprawidłowego indeksu.
- Brak pliku.
- Niepoprawne dane wejściowe.
- Wywołanie metody przez referencję null.

3. Przykład

```
int wynik = 10 / 0;
```

Wyjątek: ArithmeticException

EXCEPTION

4. Inny przykład

```
int[] liczby = {1, 2, 3};
System.out.println(liczby[10]);
```

Wyjątek: ArrayIndexOutOfBoundsException



5. Co dzieje się bez obsługi?

NORMALNY KOD → WYJĄTEK → PRZERWANY BIEŻĄCY PRZEPŁYW

6. Po co obsługiwać wyjątki?

- Zapobiec nagłemu przerwaniu działania.
- Wyświetlić przydatny komunikat.
- Zapisać informację o problemie.
- Spróbować innego rozwiązania.
- Zwolnić zasoby.

Wyjątek nie zawsze oznacza błąd programisty. Czasem opisuje problem, który program musi umieć obsłużyć.

try-catch — przechwyć problem i zareaguj

1. try

Zawiera kod, który może zgłosić wyjątek.

```
try {
    int wynik = 10 / 0;
    System.out.println(wynik);
}
```

2. catch

Zawiera reakcję na pasujący typ wyjątku.

```
catch (ArithmeticException e) {
    System.out.println(
      "Nie można dzielić przez zero");
}
```

3. Przykład

```
try {
    int wynik = 10 / 0;
    System.out.println(wynik);
} catch (ArithmeticException e) {
    System.out.println(
      "Nie można dzielić przez zero");
}
```

Wynik: Nie można dzielić przez zero

TRY → CATCH

4. Przepływ



Brak pasującego catch → wyjątek przekazywany dalej.

5. Co oznacza e?

e to obiekt reprezentujący wyjątek. Możemy odczytać jego komunikat:

```
e.getMessage();
```

6. Co warto przechwytywać?

Obsługuj problemy, na które naprawdę potrafisz odpowiednio zareagować.

7. Dobra reakcja

Nie powinna jedynie ukrywać błędu. Może:

- Poinformować użytkownika.
- Zapisać log.
- Użyć wartości zastępczej.
- Zakończyć operację bezpiecznie.

try-catch nie usuwa przyczyny problemu — pozwala kontrolować reakcję programu.

Wiele catch – różne problemy, różne reakcje

1. Przykład

```
try {
    int[] liczby = {10, 20};
    int indeks = 5;
    int wynik = liczby[indeks] / 0;
} catch (ArrayIndexOutOfBoundsException e) {
    System.out.println(
        "Niepoprawny indeks");
} catch (ArithmeticException e) {
    System.out.println(
        "Błąd obliczenia");
}
```



Dla indeks = 5: Niepoprawny indeks.

2. Jak Java wybiera?

Wybiera pierwszy catch zgodny z typem zgłoszonego wyjątku.

WYJĄTEK

- ArrayIndexOutOfBoundsException
- ArithmeticException

Błąd indeksu występuje przed dzieleniem.

3. Dlaczego różne catch?

Różne problemy mogą wymagać różnych reakcji.

Brak pliku	→	poinformuj o pliku.
Błędny numer	→	poproś o nową wartość.
Błąd indeksu	→	popraw zakres.



JEDEN try → WIELE catch

4. Kolejność ma znaczenie

Najpierw bardziej szczegółowe typy wyjątków, potem ogólniejsze.

ArrayIndexOutOfBoundsException

↓

Exception



5. Jeden catch dla kilku typów

Osobny przykład multi-catch:

```
catch (ArithmeticException |
       IllegalArgumentException e) {
    System.out.println("Błędne dane");
}
```



6. Dobra zasada

Obsługuj wyjątek na tym poziomie programu, na którym naprawdię wiesz, jak na niego zareagować.



Różne wyjątki mogą oznaczać różne problemy – dlatego reakcja też może być różna.

finally – kod końcowy po wykonaniu operacji

1. Struktura

```
try {
    // kod, który może zgłosić wyjątek
} catch (Exception e) {
    // obsługa problemu
} finally {
    // kod końcowy
}
```

2. Jak działa?

Zwykle wykona się po sukcesie i po wyjątku.



Także gdy wyjątek jest przekazywany dalej.

Zakończenie JVM może uniemożliwić wykonanie finally.

3. Przykład

```
try {
    System.out.println("Praca z danymi");
} catch (Exception e) {
    System.out.println("Wystąpił błąd");
} finally {
    System.out.println("Koniec operacji");
}
```

FINALLY

4. Do czego służy?

- Końcowe porządkowanie.
- Ręczne zamykanie zasobów w starszym kodzie.
- Czynności po próbie wykonania operacji.

5. Ważne

Do zamykania zasobów AutoCloseable zwykle wybierz `try-with-resources`.

Poznasz je w dalszej części modułu.

6. Jak zapamiętać?

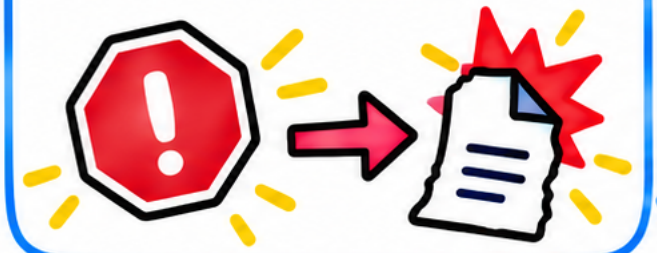
`try`	→ spróbuj
`catch`	→ obsłuż problem
`finally`	→ wykonaj krok końcowy

`finally` nie naprawia wyjątku – służy do wykonania kodu końcowego.

throw – sam zgłoś wyjątek

1. Co robi?

Przerywa bieżący przepływ i zgłasza konkretny wyjątek.

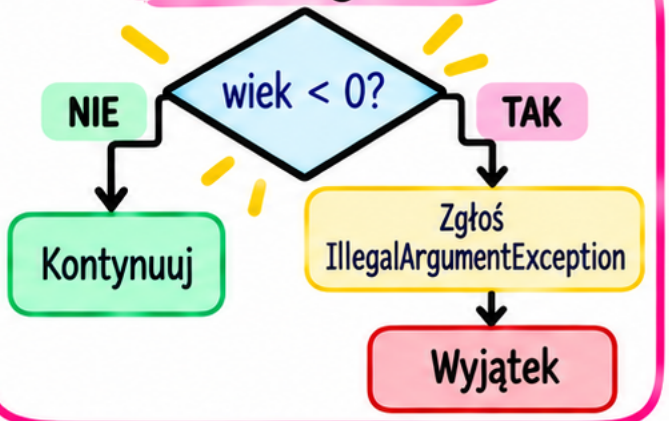


2. Przykład

Walidacja argumentu w metodzie:

```
static void ustawWiek(int wiek) {
    if (wiek < 0) {
        throw new IllegalArgumentException(
            "Wiek nie może być ujemny");
    }
}
```

3. Logika



THROW

4. Dlaczego warto?

- Zatrzymuje niepoprawne dane.
- Jasno komunikuje problem.
- Chroni stan programu.
- Przekazuje informację o błędzie wyżej.



5. Typowe przypadki

- Niepoprawny wiek.
- Ujemna cena.
- Brak wymaganego argumentu.
- Niedozwolony stan.



6. throw a return

return → normalnie kończy metodę i może zwrócić wynik

throw → zgłasza wyjątek i przerywa bieżący przepływ

throw używamy, gdy program wykrywa sytuację, której nie chce zaakceptować.

throws – poinformuj, że metoda może zgłosić wyjątek

1. Gdzie występuje?

W deklaracji metody.

Przykład nagłówka metody:

```
static String wczytajPlik()
        throws IOException
```

IOException pochodzi z pakietu java.io.

2. Co oznacza?

Metoda informuje:

„Ta operacja może zakończyć się tym wyjątkiem."

3. Kod wywołujący musi zdecydować

Dla wyjątku checked, np. IOException:

- Obsłuż wyjątek przez try-catch.
- Albo zadeklaruj go dalej w throws.

THROWS

4. Obsługa przez try-catch

```
try {
    String tekst = wczytajPlik();
} catch (IOException e) {
    System.out.println("Błąd pliku");
}
```

5. throw kontra throws

throw	throws
W kodzie:	W nagłówku metody:
`throw new IllegalArgumentException();`	`void metoda() throws IOException`
Faktycznie zgłasza wyjątek.	Deklaruje możliwość przekazania wyjątku dalej.

6. Szybkie zapamiętanie

THROW = ZGŁOŚ

THROWS = ZADEKLARUJ

Checked i unchecked exceptions — dwa ważne rodzaje wyjątków

CHECKED EXCEPTIONS

- Kompilator wymaga obsługi przez try-catch albo deklaracji throws.
- Przykład: IOException.
- Może dotyczyć operacji wejścia/wyjścia.



try-catch lub throws

Przykład nagłówka metody:

```
void wczytaj() throws IOException
```

UNCHECKED EXCEPTIONS

- Kompilator nie wymaga catch ani throws.

Przykłady:

- NullPointerException
- IllegalArgumentException
- ArithmeticException
- IndexOutOfBoundsException



Często sygnalizują:

- Błąd logiki.
- Niepoprawne dane.
- Zły stan programu.



Najprostsza różnica

CHECKED	kompilator wymaga decyzji
UNCHECKED	brak obowiązku catch lub throws

Oba rodzaje mogą wystąpić podczas działania programu.

Czy każdy unchecked trzeba łapać?



Nie. Często lepiej poprawić przyczynę problemu.

Przykład



Przy NullPointerException sprawdź, dlaczego referencja ma wartość null. Nie ukrywaj tylko objawu.



Obsługa wyjątków nie zastępuje poprawnego projektowania kodu.

Własny wyjątek – nazwij problem w języku swojej aplikacji

1. Po co własny wyjątek?

Zamiast ogólnego typu wyjątku, np. IllegalArgumentException, możemy nadać problemowi własną, konkretną nazwę.



2. Przykład

```
class BrakSrodkowException
        extends RuntimeException {
    BrakSrodkowException(String message) {
        super(message);
    }
}
```

extends RuntimeException → wyjątek unchecked.

3. Użycie

Fragment metody wyplac:

```
if (kwota > saldo) {
    throw new BrakSrodkowException(
        "Za mało środków na koncie");
}
```



CUSTOM EXCEPTION

4. Obsługa

```
try {
    konto.wyplac(500);
} catch (BrakSrodkowException e) {
    System.out.println(e.getMessage());
}
```



5. Co zyskujemy?

- Jasne znaczenie.
- Łatwiejsze rozpoznawanie problemów.
- Kod bliższy językowi domeny.
- Precyzyjniejsza obsługa.



6. Nie twórz własnego wyjątku bez potrzeby

Używaj go wtedy, gdy poprawia czytelność aplikacji.



Dobra nazwa wyjątku powinna od razu mówić, jaki problem wystąpił.

Pliki i ścieżki – jak Java odnajduje dane na dysku

1. Co to jest ścieżka?

Informacja, gdzie znajduje się plik lub katalog.

`dane.txt` lub `folder/dane.txt`



2. Tworzenie Path

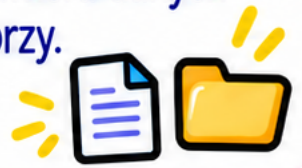
```
import java.nio.file.Path;

Path sciezka = Path.of("dane.txt");
```



3. Co przechowuje Path?

Reprezentuje lokalizację pliku lub katalogu. Nie zawiera danych pliku ani go nie tworzy.



PATH

4. Path w operacjach plikowych



5. Ścieżka względna

```
Path.of("dane.txt");
```

Względem bieżącego katalogu roboczego programu.



6. Ścieżka absolutna

Opisuje pełną lokalizację w systemie plików.



7. Dobra praktyka

Unikaj wpisywania na stałe ścieżek zależnych od jednego komputera.



Path mówi GDZIE. Files pomaga wykonać OPERACJĘ.

Odczyt pliku – pobierz tekst z dysku

1. Importy

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
```

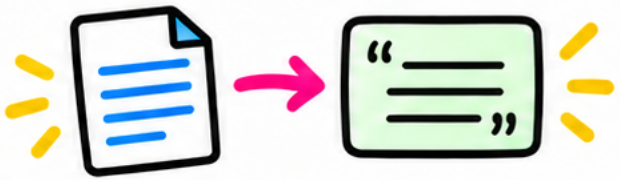
2. Ścieżka

```
Path plik = Path.of("notatki.txt");
```



3. Odczyt

```
String tekst = Files.readString(plik);
```



READ FILE

4. Przykład w metodzie

```
try {
    Path plik = Path.of("notatki.txt");
    String tekst = Files.readString(plik);
    System.out.println(tekst);
} catch (IOException e) {
    System.out.println(
      "Nie udało się odczytać pliku");
}
```

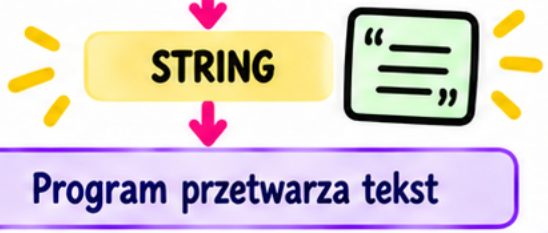
5. Dlaczego try-catch?

Operacje na plikach mogą się nie powieść.

- Brak pliku.
- Brak dostępu.
- Problem z odczytem.



6. Przepływ



Odczyt pliku zamienia dane zapisane na dysku w dane dostępne dla programu.

Zapis do pliku – zachowaj dane poza programem

1. Dane do zapisania

```
String tekst =
   "Moja pierwsza notatka z Javy";
```



2. Ścieżka

```
Path plik = Path.of("notatki.txt");
```



3. Zapis

```
Files.writeString(plik, tekst);
```



WRITE FILE

4. Przykład w metodzie

Importy:

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
```



Kod:

```
try {
   Path plik = Path.of("notatki.txt");
   Files.writeString(plik,
      "Java jest coraz prostsza!");
} catch (IOException e) {
   System.out.println("
      "Nie udało się zapisać pliku");
}
```

5. Co się dzieje?

STRING

↓

Files.writeString()

↓

PLIK NA DYSKU



6. Gdzie się przydaje?

- Notatki.
- Proste konfiguracje.
- Wyniki programu.
- Raporty.
- Eksport danych.



7. Uwaga



Bez dodatkowych opcji writeString tworzy plik lub zastępuje jego dotychczasową treść.

Wybierz świadomie: nadpisanie, dopisanie lub utworzenie nowego pliku.

Zmienne żyją podczas działania programu. Plik pozwala zachować dane na później.

readAllLines() – wczytaj plik jako listę wierszy

1. Przykładowy plik

imiona.txt

```
Anna
Jan
Ola
Marek
```

2. Odczyt wszystkich linii

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.List;
```

Fragmenty kodu na tej stronie należą do jednej metody z throws IOException.

```
List<String> linie = Files.readAllLines(
        Path.of("imiona.txt"));
```

3. Co otrzymujemy?

```
["Anna", "Jan", "Ola", "Marek"]
```

"Anna"	"Jan"	"Ola"	"Marek"

4. Teraz możemy używać kolekcji

```
for (String linia : linie) {
    System.out.println(linia);
}
```

PLIK → LISTA

5. Możemy też filtrować

```
for (String linia : linie) {
    if (linia.startsWith("A")) {
        System.out.println(linia);
    }
}
```

Wynik: Anna

6. Co tutaj łączymy?

Pliki | List | String | Pętle | Warunki

7. Typowe zastosowania

- Listy danych.
- Konfiguracje.
- Wiersze CSV do dalszej analizy.
- Raporty tekstowe.

Odczyt wierszy nie zastępuje parsera CSV.

Plik tekstowy może stać się kolekcją danych gotową do dalszego przetwarzania.

try-with-resources – automatycznie zamykaj zasoby

1. Problem

Niektóre obiekty korzystają z zasobów zewnętrznych i trzeba je prawidłowo zamknąć.

- Strumienie.
- Czytniki.
- Obiekty do zapisu.
- Połączenia.

2. Tradycyjny problem

Bez prawidłowego zamknięcia zasób może pozostać otwarty.



TRY-WITH-RESOURCES

3. Rozwiązanie

Importy:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
```

Kod w metodzie:

```
try (BufferedReader reader =
        Files.newBufferedReader(
            Path.of("dane.txt"))) {
    String linia = reader.readLine();
    System.out.println(linia);
} catch (IOException e) {
    System.out.println("Błąd odczytu");
}
```

Przykład odczytuje jedną linię.

4. Co robi Java?

OTWÓRZ ZASÓB → WYKONAJ KOD → AUTOMATYCZNIE ZAMKNIJ

Także wtedy, gdy wystąpi wyjątek.

5. Dlaczego warto?

- Mniej kodu.
- Mniejsze ryzyko pominięcia zamknięcia.
- Bezpieczniejsze zarządzanie zasobami.
- Nowoczesny sposób pracy.

6. Kiedy używać?

Gdy zasób implementuje interfejs AutoCloseable, bezpośrednio lub pośrednio.

AutoCloseable

7. finally kontra try-with-resources

finally → wykonuje kod końcowy

try-with-resources → automatycznie zamyka zasoby

Jeśli zasób trzeba zamknąć, sprawdź najpierw, czy możesz użyć try-with-resources.

CSV — zamień tekst z pliku na dane

1. Przykładowy plik

```
Anna,20,Warszawa
Jan,25,Kraków
Ola,22,Gdańsk
```

2. Każda linia opisuje jeden rekord

Anna,20,Warszawa

podział

- imię → Anna
- wiek → 20
- miasto → Warszawa

3. Odczyt linii

Importy:

```
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.util.List;
```

Fragmenty kodu należą do jednej metody z throws IOException.

```
List<String> linie = Files.readAllLines(
                     Path.of("osoby.csv"));
```

CSV

4. Podział tekstu

```
for (String linia : linie) {
    String[] dane = linia.split(",");
    String imie = dane[0];
    int wiek = Integer.parseInt(dane[1]);
    String miasto = dane[2];
}
```

5. Co tutaj łączymy?

- ✓ Pliki
- ✓ List<String>
- ✓ split()
- ✓ Tablice
- ✓ Konwersję typów
- ✓ Pętle

6. Ważne

Prosty przykład: trzy niepuste pola, brak nagłówka i poprawny wiek. split(",") nie obsługuje pełnego CSV, np. przecinków w polach ujętych w cudzysłowy.

7. Schemat

PLIK CSV
↓
WIERSZE
↓
split(",")
↓
POLA DANYCH
↓
OBIEKTY / ANALIZA

Tekst w pliku może być początkiem prawdziwego procesu przetwarzania danych.

Podsumowanie – wyjątki, pliki i operacje na danych

1 Wyjątek
Sygnalizuje problem podczas działania programu.

2 try
Kod, który może zakończyć się wyjątkiem.

3 catch
Reakcja na określony problem.

4 finally
Kod końcowy przy wyjściu z try/catch.

BEZPIECZNY PROGRAM

5 throw
Ręcznie zgłasza wyjątek.

6 throws
Deklaruje wyjątki, które metoda może przekazać wyżej.

7 Path
Opisuje lokalizację pliku lub katalogu.

8 Files
Pomaga czytać i zapisywać dane.

9 try-with-resources
Automatycznie zamyka zasoby.

Proces pracy z danymi

1. DANE ZEWNĘTRZNE
2. ODCZYT
3. WALIDACJA
4. PRZETWARZANIE
5. OBSŁUGA PROBLEMÓW
6. ZAPIS

Najważniejsza zasada

Nie ignoruj błędów bezmyślnie. Program powinien reagować na problemy w sposób przewidywalny.

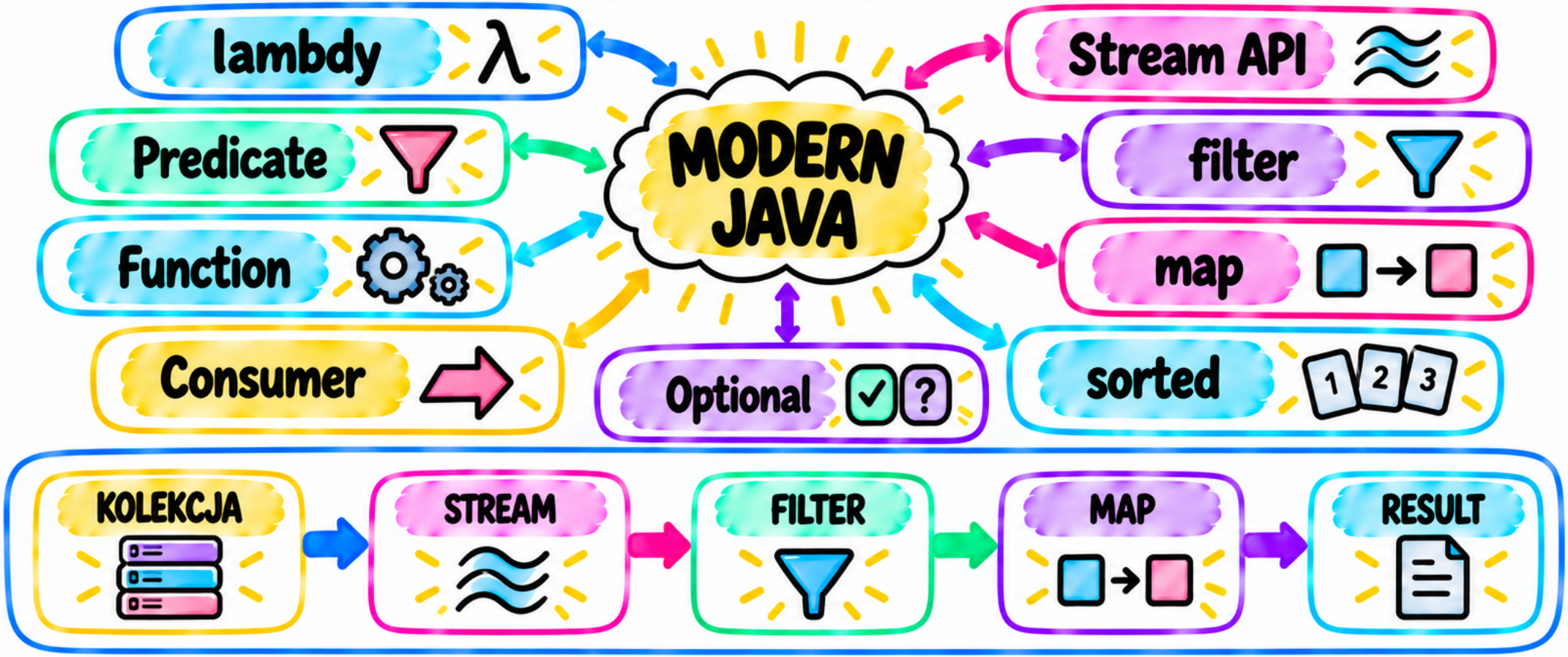
Co dalej?

LAMBDY • STREAM API • OPTIONAL • NOWOCZESNA JAVA

MODUŁ 09

NOWOCZESNA JAVA I DOBRE PRAKTYKI

Lambdy, Stream API, Optional i czytelniejsze przetwarzanie danych.



Lambda — krótki zapis zachowania

1. Podstawowa forma

Schemat zapisu:

```
(parametry) -> działanie
```

Typ lambdy wynika z kontekstu interfejsu funkcyjnego.

2. Prosty przykład

Fragment lambdy:

```
x -> x * 2
```

weź x → zwróć x * 2

3. Więcej parametrów

Fragment lambdy:

```
(a, b) -> a + b
```

4. Większy blok

Fragment lambdy:

```
x -> {
    int wynik = x * 2;
    return wynik;
}
```

LAMBDA

5. Gdzie spotkasz lambdy?

- forEach
- filter
- map
- Sortowanie
- Interfejsy funkcyjne

6. Przykład z listą

```
import java.util.List; // typ List

List<String> imiona =
    List.of("Anna", "Jan", "Ola");
imiona.forEach(
    imie -> System.out.println(imie)
);
```

"Anna"
"Jan"
"Ola"

7. Jak o tym myśleć?

Lambda opisuje, CO ZROBIĆ.
Dla małej operacji nie musisz tworzyć osobnej klasy.

Lambda = krótki sposób zapisania zachowania.

Interfejs funkcyjny – kontrakt z jedną metodą abstrakcyjną

FUNCTIONAL INTERFACE

1. Idea

Interfejs funkcyjny ma jedną metodę abstrakcyjną reprezentującą jego kontrakt.



2. Lambda jako implementacja

```
import java.util.function.*;

Predicate<Integer> dodatnia =
        x -> x > 0;
```



3. Najważniejsze typy

Predicate<T>

Wejście: T
Wynik: boolean
Przykład: Predicate<Integer>

```
x -> x > 10
```



Function<T, R>

Wejście: T
Wynik: R
Przykład: Function<String, String>

```
x -> x.toUpperCase()
```



Consumer<T>

Wejście: T
Wynik: brak
Przykład: Consumer<String>

```
x -> System.out.println(x)
```



Supplier<T>

Wejście: brak
Wynik: T
Przykład: Supplier<String>

```
() -> "Java"
```



4. Jak zapamiętać?

- **Predicate** → sprawdza
- **Function** → przekształca
- **Consumer** → używa
- **Supplier** → dostarcza

Lambdy współpracują z interfejsami funkcyjnymi.

Stream API — przetwarzaj kolekcje krok po kroku

1. Dane wejściowe

```
import java.util.List;

List<Integer> liczby =
    List.of(1, 2, 3, 4, 5);
```

2. Tworzenie streamu

Fragment potoku:

```
liczby.stream()
```



3. Operacje

```
liczby.stream()
      .filter(x -> x > 2)
      .forEach(System.out::println);
```

Wynik:

```
3
4
5
```

STREAM

4. Jak o tym myśleć?

Schemat ogólny:



5. Stream nie jest kolekcją

Nie przechowuje danych jak kolekcja.
Opisuje sposób przetwarzania źródła danych.

6. Dlaczego warto?

- Czytelny kod.
- Łączenie operacji.
- Łatwe filtrowanie i przekształcanie.
- Mniej jawnych pętli w wielu przypadkach.

7. Ważna idea

- W tym przykładzie lista źródłowa pozostaje bez zmian.
- Przetwarzanie uruchamia operacja końcowa, np. `forEach()`.

Stream API pozwala opisać CO ma się stać z danymi.

filter() — zostaw tylko elementy spełniające warunek

1. Lista

```
import java.util.List;

List<Integer> liczby =
    List.of(3, 8, 12, 5, 20);
```

2. Tylko wartości większe od 10

```
liczby.stream()
      .filter(x -> x > 10)
      .forEach(System.out::println);
```

Wynik:

```
12
20
```

3. Co robi filter()?

Dla każdego elementu:

Sprawdź warunek

- **true → zostaw**
- **false → odrzuć**

FILTER

4. Inny przykład

```
List<String> imiona =
    List.of("Anna", "Jan", "Aleksandra");

imiona.stream()
    .filter(imie -> imie.startsWith("A"))
    .forEach(System.out::println);
```

Wynik:

```
Anna, Aleksandra
```

5. filter() oczekuje warunku

Logiki, która dla elementu zwraca `true` albo `false`.

true | **false**

6. Typowe zastosowania

- Pełnoletni użytkownicy.
- Produkty powyżej określonej ceny.
- Aktywne konta.
- Nazwy zaczynające się od wybranej litery.
- Poprawne rekordy.

FILTER = WYBIERZ ELEMENTY

map() – przekształć każdy element

1. Lista wejściowa

```
import java.util.List;

List<String> imiona =
    List.of("anna", "jan", "ola");
```

2. Transformacja

```
imiona.stream()
      .map(String::toUpperCase)
      .forEach(System.out::println);
```

Wynik:

```
ANNA
JAN
OLA
```

3. Jak działa?

"anna" → transformuj → "ANNA"

"jan" → transformuj → "JAN"

MAP

4. Może zmienić typ

```
List<String> teksty =
    List.of("Java", "Kod");
teksty.stream()
      .map(String::length)
      .forEach(System.out::println);
```

"Java" → 4

"Kod" → 3

Elementy typu String stają się wartościami typu Integer.

5. filter() kontra map()

FILTER	MAP
wybiera	zmienia

6. Typowe zastosowania

- Przekształcanie imion.
- Obliczanie wartości.
- Wyodrębnianie pól obiektów.
- Zmiana formatów.

MAP = WEŹ ELEMENT I ZAMIEŃ GO W COŚ INNEGO

sorted() i distinct() – uporządkuj i usuń duplikaty

SORTED

1 Dane

Przykładowa lista liczb w nieposortowanej kolejności.

```
import java.util.List;

List<Integer> liczby =
    List.of(5, 2, 8, 1);
```

2 Sortowanie

Używamy metody sorted(), aby uporządkować elementy rosnąco.

```
liczby.stream()
  .sorted()
  .forEach(System.out::println);
```

Wynik:

```
1
2
5
8
```

DISTINCT

3 Dane z duplikatami

Przykładowa lista napisów z powtórzeniami.

```
List<String> jezyki =
    List.of("Java", "Python",
            "Java", "C++");
```

4 Usuwanie powtórzeń

Używamy metody distinct(), aby usunąć duplikaty.

```
jezyki.stream()
  .distinct()
  .forEach(System.out::println);
```

Wynik:

```
Java
Python
C++
```

5 Możemy łączyć operacje

Metody możemy łączyć, np. usuwać duplikaty, a następnie sortować wynik.

```
liczby.stream()
  .distinct()
  .sorted()
  .forEach(System.out::println);
```

6 Kolejność operacji

Dobieraj kolejność operacji do celu. Tutaj najpierw distinct(), potem sorted().

DANE → distinct() → sorted() → WYNIK

7 Dobra praktyka

Potok powinien być czytelny i jasno pokazywać etapy przetwarzania danych.

Stream pozwala składać małe operacje w jeden czytelny przepływ.

Zapisz wynik streamu z powrotem do listy

1. Dane

```
import java.util.List;

List<Integer> liczby =
    List.of(1, 2, 3, 4, 5, 6);
```

1	2	3	4	5	6

2. Chcemy liczby parzyste

```
List<Integer> parzyste =
    liczby.stream()
        .filter(x -> x % 2 == 0)
        .toList();
```

Wynik: `[2, 4, 6]`

Stream.toList(): Java 16+, lista niemodyfikowalna.

3. Co się stało?



4. Oryginalne dane

Lista liczby nadal istnieje
i zawiera [1, 2, 3, 4, 5, 6].

STREAM → NOWA LISTA

5. Starszy styl spotykany w kodzie

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

Fragment końca potoku:

```
.collect(Collectors.toList())
```

Ten zapis możesz spotkać w istniejących projektach.

6. Typowy przykład

```
List<String> imiona =
    List.of("Anna", "Jan", "Aleksandra");

List<String> dlugieImiona =
    imiona.stream()
        .filter(imie -> imie.length() > 4)
        .toList();
```

Wynik: `[Aleksandra]`

7. Najważniejsza idea

Strumienie mogą filtrować, przekształcać i sortować dane, a następnie tworzyć nowy wynik.

PIPELINE → WYNIK

Sprawdzaj kolekcję bez ręcznego pisania pętli

anyMatch()

Czy przynajmniej jeden element spełnia warunek?

```
boolean wynik = liczby.stream()
    .anyMatch(x -> x > 10);
```

Wynik: `true`

Ponieważ 12 > 10.



allMatch()

Czy wszystkie elementy spełniają warunek?

```
boolean dodatnie = liczby.stream()
    .allMatch(x -> x > 0);
```

Wynik: `true`



noneMatch()

Czy żaden element nie spełnia warunku?

```
boolean brakUjemnych = liczby.stream()
    .noneMatch(x -> x < 0);
```

Wynik: `true`



MATCH

Dane wspólne dla trzech przykładów

```
import java.util.List;

List<Integer> liczby =
    List.of(3, 8, 12, 5);
```



1 Szybkie porównanie

anyMatch()	przynajmniej jeden
allMatch()	wszystkie
noneMatch()	żaden

2 Typowe zastosowania

- Sprawdzanie uprawnień
- Walidacja danych
- Kontrola cen
- Statusy użytkowników
- Poprawność rekordów

3 Najważniejsza idea

Te operacje zwracają wartość typu

`boolean`

Nie zawsze potrzebujesz znaleźć element. Czasem wystarczy wiedzieć, czy warunek został spełniony.

findFirst() — znajdź pierwszy pasujący element bez ryzykownego null

1. Dane

```
import java.util.List;
import java.util.Optional;

List<String> imiona =
   List.of("Jan", "Anna", "Adam", "Ola");
```

2. Pierwsze imię na A

Fragment kodu:

```
Optional<String> wynik =
   imiona.stream()
      .filter(imie -> imie.startsWith("A"))
      .findFirst();
```

W tym przykładzie: **Anna**

3. Co otrzymujemy?

Nie otrzymujemy bezpośrednio wartości String.

Otrzymujemy:

`Optional<String>`

FIND FIRST

4. Dlaczego?

Możliwe są dwie sytuacje:
znaleziono element albo nic nie znaleziono.

Optional → **WARTOŚĆ** / **BRAK WARTOŚCI**

5. Bezpieczne użycie

Fragment kodu:

```
wynik.ifPresent(
    System.out::println
);
```

Wypisze: **Anna**

6. Wartość domyślna

Fragment kodu:

```
String imie =
   wynik.orElse("Brak");
```

Tu: Anna.
Gdy brak wyniku: Brak.

7. Dlaczego to pomaga?

- Jasno modeluje brak wartości.
- Zmniejsza ryzyko przypadkowych błędów.
- Zachęca do świadomej obsługi braku wyniku.

findFirst() zwraca Optional, ponieważ wynik może istnieć lub nie.

Optional — obsługuj brak wartości świadomie

1. Optional z wartością

```
Optional<String> wynik =
    Optional.of("Java");
```

of() wymaga wartości różnej od null.

2. Optional bez wartości

```
Optional<String> wynik =
    Optional.empty();
```

3. Gdy wartość może być null

```
String tekst = null;
Optional<String> wynik =
    Optional.ofNullable(tekst);
```

4. isPresent()

```
if (wynik.isPresent()) {
    System.out.println(wynik.get());
}
```

Ten zapis działa, ale często istnieje czytelniejsza forma.

OPTIONAL

5. ifPresent()

```
wynik.ifPresent(
    System.out::println
);
```

6. orElse()

```
String tekst =
    wynik.orElse("Brak danych");
```

```
import java.util.Optional;
```

Przykłady na tej stronie są niezależne.
W kartach 4–7 wynik ma typ Optional<String>.

7. orElseThrow()

```
String tekst =
    wynik.orElseThrow();
```

Brak wartości → NoSuchElementException.

Użyj, gdy brak wyniku powinien przerwać operację.

8. Jak myśleć o Optional?

MAM WARTOŚĆ? → użyj jej

NIE MAM? → wybierz świadomą reakcję

Optional nie jest magicznym zamiennikiem każdego **null**. Używaj go tam, gdzie brak wyniku jest częścią modelu operacji.

reduce() – połącz wiele elementów w jeden wynik

1. Idea

WIELE ELEMENTÓW → ŁĄCZENIE → JEDEN WYNIK

Metoda reduce() łączy wszystkie elementy strumienia, zwracając jeden wynik.

2. Przykład – suma

```
import java.util.List;
List<Integer> liczby =
    List.of(1, 2, 3, 4);
int suma = liczby.stream()
    .reduce(0, (a, b) -> a + b);
```

Wynik: 10

3. Co oznacza 0?

Element neutralny dodawania: 0 + x = x.

```
0 + 1 = 1
1 + 2 = 3
3 + 3 = 6
6 + 4 = 10
```

4. Inny przykład – iloczyn

```
int iloczyn = liczby.stream()
    .reduce(1, (a, b) -> a * b);
```

Wynik: 24

1 to element neutralny mnożenia.

REDUCE

5. Przykład z tekstem

```
String wynik =
    List.of("Java", "jest", "OK")
        .stream()
        .reduce((a, b) -> a + " " + b)
        .orElse("");
```

Wynik: Java jest OK

Bez wartości początkowej reduce() zwraca Optional.

6. Kiedy używać?

- Sumowanie
- Obliczanie iloczynu
- Agregowanie danych
- Tworzenie jednego wyniku

7. map() kontra reduce()

map()	element → element
reduce()	wiele elementów → jeden wynik

REDUCE = ZREDUKUJ KOLEKCJĘ DO JEDNEJ WARTOŚCI

Method reference :: — krótszy zapis prostej lambdy

Wspólne dane źródłowe:

```
import java.util.List;
List<String> imiona = List.of("Anna", "Jan", "Ola");
```



LAMBDA

```
imiona.forEach(
    imie -> System.out.println(imie)
);
```

=

METHOD REFERENCE

```
imiona.forEach(
    System.out::println
);
```

1 Symbol

::

2 Co oznacza?

Użyj istniejącej metody jako operacji.

3 Inny przykład

Porównanie zapisu:

Lambda:

```
tekst -> tekst.toUpperCase()
```

Method reference:

```
String::toUpperCase
```

4 Przykład w streamie

```
imiona.stream()
    .map(String::toUpperCase)
    .forEach(System.out::println);
```

5 Kiedy używać?

Gdy lambda jedynie bezpośrednio wywołuje istniejącą metodę.

6 Kiedy nie używać?

Gdy taki zapis jest mniej czytelny niż prosta lambda.

7 Najważniejsza zasada

Czytelniej jako lambda? → użyj lambdy

Czytelniej jako ::? → użyj odwołania do metody

Odwołanie do metody to krótszy zapis użycia istniejącej metody.

Pipeline Stream API — filtruj, przekształcaj i sortuj w jednym przepływie

1. Dane

```
import java.util.List;

List<String> imiona = List.of(
    "anna", "jan", "aleksandra",
    "ola", "adam"
);
```

2. Zadanie

Chcemy:

1. Zostawić imiona na a.
2. Zmienić je na wielkie litery.
3. Posortować.
4. Utworzyć nową listę.

3. Pipeline

```
List<String> wynik =
    imiona.stream()
        .filter(imie -> imie.startsWith("a"))
        .map(String::toUpperCase)
        .sorted()
        .toList();
```

Wynik: `[ADAM, ALEKSANDRA, ANNA]`

`toList()`: Java 16+, lista niemodyfikowalna.

SOURCE → stream() → FILTER → MAP → SORT → toList() → RESULT

4. Każda linia ma jedno zadanie

`stream()`	utwórz strumień
`filter()`	wybierz
`map()`	przekształć
`sorted()`	uporządkuj
`toList()`	wykonaj i zbierz wynik

5. Dlaczego taki zapis pomaga?

- Widoczne kroki przetwarzania.
- Niewiele zmiennych pomocniczych.
- Łatwe dodawanie i usuwanie etapów.
- Czytanie od góry do dołu.

6. Dobra praktyka

Bardzo długi potok można podzielić na metody o czytelnych nazwach.

Stream jest najbardziej czytelny, gdy każda operacja reprezentuje jeden jasny krok przetwarzania.

Czytelny kod — nowoczesna Java bez przesady

1. Używaj lambd, gdy upraszczają kod

Krótki warunek:

```
x -> x > 10
```

Nie zamieniaj złożonej logiki w jedną ogromną lambdę.

2. Stream nie zawsze zastępuje pętlę

Jeśli prosta pętla jest czytelniejsza:

→ pętla jest w porządku.

3. Nadawaj dobre nazwy

W złożonym kontekście:

```
x -> uzytkownik
```

Czytelniejsza nazwa opisuje rolę wartości.

CZYTELNOŚĆ > SPRYT

4. Jeden krok = jedna odpowiedzialność

Potok powinien jasno pokazywać każde przekształcenie.

5. Unikaj niepotrzebnego null

Gdy brak wartości jest częścią wyniku:

→ rozważ Optional.

6. Nie komplikuj dla samej nowoczesności

Prosty, przewidywalny kod jest lepszy niż krótki, lecz niejasny.

7. Pisz dla człowieka

Komputer wykonuje kod. Ludzie go:

- ✓ czytają
- ✓ poprawiają
- ✓ rozwijają
- ✓ testują

Czy kod jest:

- ✓ czytelny?
- ✓ przewidywalny?
- ✓ łatwy do testowania?
- ✓ łatwy do zmiany?

DOBRY KOD NIE TYLKO DZIAŁA — DOBRY KOD DA SIĘ ZROZUMIEĆ

Podsumowanie — nowoczesna Java i przetwarzanie danych

1 Lambda

Krótki zapis zachowania.

Fragment: `(x) -> x * 2`

2 Predicate

Sprawdza warunek.

3 Function

Przekształca wartość.

NOWOCZESNA JAVA

4 Consumer

Wykonuje akcję na wartości.

5 Stream API

Opisuje przepływ przetwarzania danych.

6 filter()

Wybiera elementy.

7 map()

Przekształca elementy.

8 sorted() / distinct()

sorted() → porządkuje

distinct() → usuwa powtórzenia

9 reduce()

Wiele wartości → jeden wynik.

10 Optional

Modeluje wynik, który może istnieć lub nie.

Przykładowy potok przetwarzania



Najważniejsza zasada

Nowoczesna składnia ma pomagać w zrozumieniu kodu, a nie zamieniać go w zagadkę.

Co dalej?

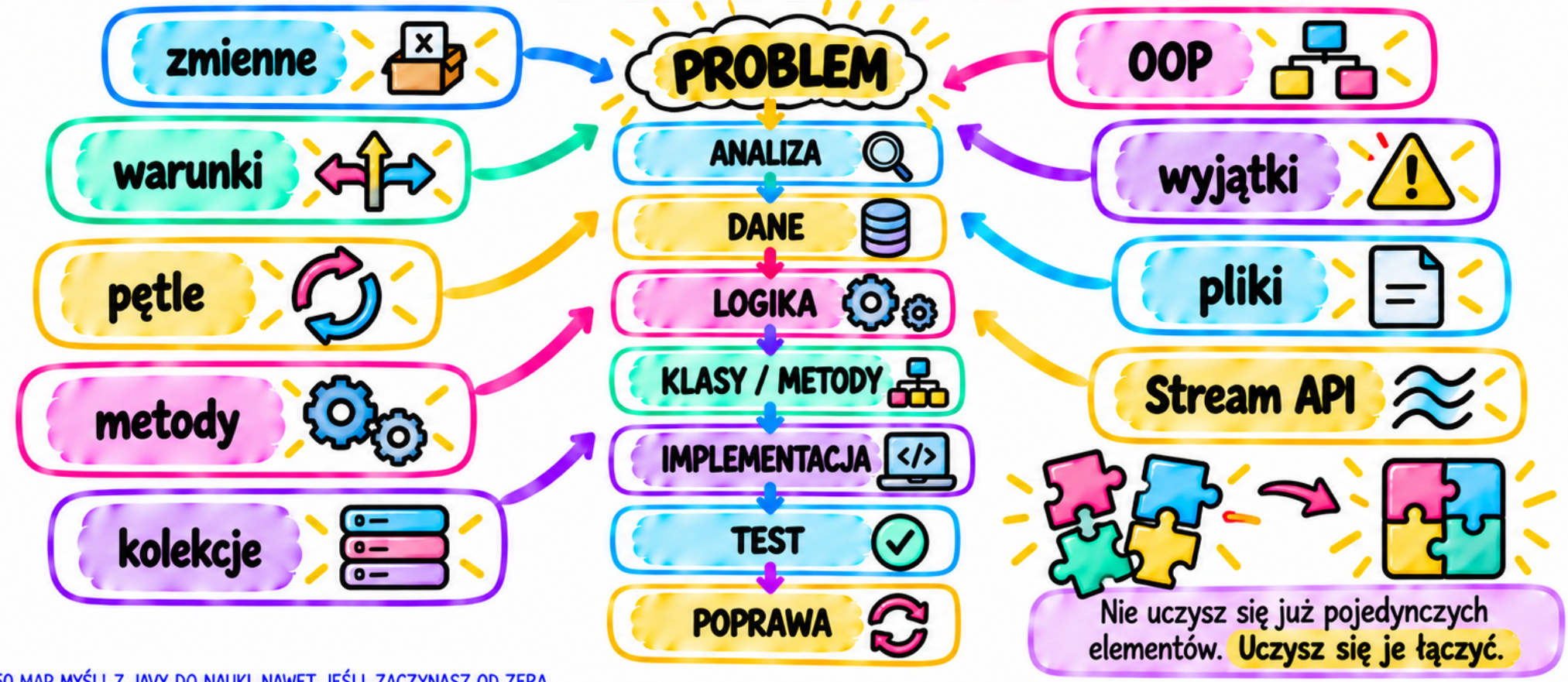
PROJEKTY, WZORCE MYŚLENIA I INTEGRACJA WIEDZY

Czas połączyć wszystko, czego nauczyłeś się do tej pory.

MODUŁ 10

PROJEKTY, WZORCE MYŚLENIA I INTEGRACJA WIEDZY

Od pojedynczych instrukcji do kompletnego sposobu myślenia programisty.



Od problemu do kodu — 7 kroków myślenia programisty

1 KROK 1 — ZROZUM PROBLEM

Zapytaj:

1. Co ma zrobić program?
2. Jakie dane otrzymuje?
3. Jaki wynik ma zwrócić?



2 KROK 2 — OKREŚL DANE

Określ:

1. zmienne
2. typy danych
3. kolekcje
4. obiekty



3 KROK 3 — PODZIEL PROBLEM

Duże zadanie → mniejsze części

System ocen można podzielić na:

1. dodaj ocenę
2. policz średnią
3. znajdź najwyższą ocenę
4. pokaż raport



4 KROK 4 — WYBIERZ NARZĘDZIA

Decyzja?	if
Powtórzenie?	pętla
Wiele danych?	kolekcja
Wiele odpowiedzialności?	metody / klasy



5 KROK 5 — NAPISZ NAJPROSTSZE ROZWIĄZANIE

Nie szukaj perfekcji, zanim uzyskasz działające rozwiązanie.



6 KROK 6 — TESTUJ

Sprawdź:

1. przypadek normalny
2. wartość graniczną
3. błędne dane
4. pusty zbiór
5. nietypowe wejście



7 KROK 7 — UPROŚĆ I POPRAW

Zapytaj:

1. Czy nazwy są czytelne?
2. Czy kod się powtarza?
3. Czy metoda robi za dużo?
4. Czy istnieje prostsze rozwiązanie?



Skrót procesu

ZROZUM → PODZIEL → ZAPROJEKTUJ → NAPISZ → TESTUJ → POPRAW

Programowanie zaczyna się od myślenia o problemie — kod jest dopiero zapisem rozwiązania.

Projekt 1 — kalkulator krok po kroku

1. Co program ma robić?

1. Pobrać dwie liczby.
2. Pobrać rodzaj działania.
3. Wykonać obliczenie.
4. Pokazać wynik.



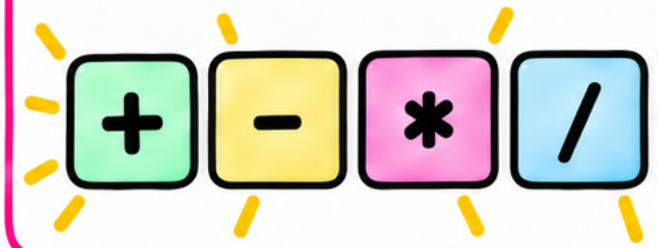
2. Dane wejściowe

Zmienne — wartości pobierz przed obliczeniem.

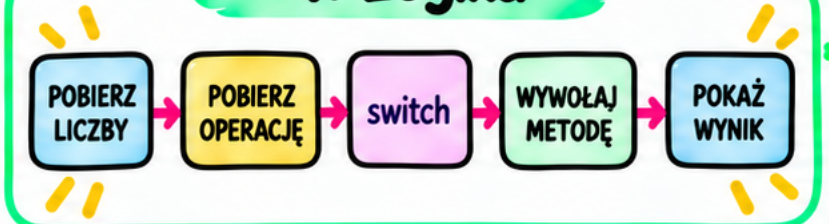
```
double a;
double b;
char operacja;
```



3. Dostępne operacje



4. Logika



KALKULATOR

5. Przykładowe metody

Wewnątrz klasy kalkulatora:

```
static double dodaj(double a, double b) {
    return a + b;
}

static double odejmij(double a, double b) {
    return a - b;
}
```



6. Uwaga przy dzieleniu

W gałęzi dzielenia:

```
if (b == 0) {
    System.out.println(
        "Nie można dzielić przez zero");
} else {
    System.out.println(a / b);
}
```



7. Co ćwiczysz?

- Zmienne
- Scanner
- switch
- Metody
- Warunki
- Walidację



ROZBUDUJ:

Dodaj procenty lub potęgowanie.



Projekt 2 — prosta lista zadań

1. Co przechowujemy?

```
import java.util.*;
List<String> zadania =
    new ArrayList<>();
```

2. Menu

```
1 - Dodaj zadanie
2 - Pokaż zadania
3 - Usuń zadanie
0 - Wyjście
```

3. Główna pętla

Szkielet pętli — obsługę opcji uzupełnij.

```
Scanner scanner =
    new Scanner(System.in);
while (true) {
    // pokaż menu
    int wybor = scanner.nextInt();
    if (wybor == 0) break;
    // wykonaj operację
}
```

Zakładamy wybór zapisany jako liczba całkowita.

4. Dodawanie

```
zadania.add(
    "Nauczyć się Stream API");
```

TO-DO LIST

5. Usuwanie

Przykład: usuń pierwsze zadanie.

```
if (!zadania.isEmpty()) {
    zadania.remove(0);
}
```

6. Wyświetlanie

```
for (String zadanie : zadania) {
    System.out.println(zadanie);
}
```

7. Podział na metody

- pokazMenu()
- dodajZadanie()
- pokazZadania()
- usunZadanie()

8. Co ćwiczysz?

- ArrayList
- Pętle
- Warunki
- Metody
- Dane użytkownika

Mały projekt uczy więcej, gdy każda funkcja ma własną odpowiedzialność.

Projekt 3 — system ocen ucznia

1. Klasa

```
import java.util.List;
import java.util.ArrayList;
class Uczen {
    private String imie;
    private List<Integer> oceny;
}
```

Bloki 2–4 należą do klasy Uczen.

2. Konstruktor

```
Uczen(String imie) {
    this.imie = imie;
    this.oceny = new ArrayList<>();
}
```

3. Dodawanie ocen

```
void dodajOcene(int ocena) {
    oceny.add(ocena);
}
```

UCZEŃ + OCENY

4. Obliczanie średniej

```
double obliczSrednia() {
    if (oceny.isEmpty()) {
        throw new IllegalStateException(
            "Brak ocen");
    }
    int suma = 0;
    for (int ocena : oceny) {
        suma += ocena;
    }
    return (double) suma / oceny.size();
}
```

5. Co powinien robić system?

1. Tworzyć ucznia
2. Dodawać oceny
3. Liczyć średnią
4. Pokazywać najwyższą ocenę
5. Wyświetlać raport

6. Co tutaj ćwiczysz?

- OOP
- private
- Konstruktor
- List
- Pętle
- Metody

ROZBUDUJ:

Dodaj metodę zwracającą „zdane" / „niezdane" na podstawie średniej.

Projekt 4 – prosty magazyn produktów

1. Klasa Produkt

Szkielet klasy – dodaj konstruktor inicjalizujący pola.



```
class Produkt {
    private String nazwa;
    private double cena;
    private int ilosc;
    public String getNazwa() {
        return nazwa;
    }
}
```

2. Lista produktów

```
import java.util.List;
import java.util.ArrayList;
List<Produkt> produkty =
    new ArrayList<>();
```



3. Operacje

- Dodaj produkt
- Usuń produkt
- Zmień ilość
- Znajdź produkt
- Policz wartość magazynu

MAGAZYN

4. Wyszukiwanie

```
String szukanaNazwa = "Długopis";
for (Produkt produkt : produkty) {
    if (szukanaNazwa.equals(
            produkt.getNazwa())) {
        // znaleziono
    }
}
```



5. Wartość produktu

Obliczenie wewnątrz klasy Produkt:

```
cena * ilosc
```



6. Wartość całego magazynu

PĘTLA → OBLICZ WARTOŚĆ KAŻDEGO PRODUKTU → DODAJ DO SUMY



7. Jakie zagadnienia łączysz?

- Klasy
- Obiekty
- Enkapsulację
- Listy
- Wyszukiwanie
- Agregację danych



WYZWANIE



DODAJ: Sortowanie produktów po cenie lub nazwie.

Projekt 5 — książka kontaktów z Map

1. Struktura danych

```
import java.util.Map;
import java.util.HashMap;
Map<String, String> kontakty =
    new HashMap<>();
```

Może reprezentować:

imię → numer telefonu

Przykłady operacji są niezależne.
Zakładamy dodany kontakt Anna.

2. Dodawanie

```
kontakty.put(
    "Anna",
    "123456789"
);
```

3. Pobieranie

```
String telefon =
    kontakty.get("Anna");
```

Wynik: **123456789**

KONTAKTY

4. Usuwanie

```
kontakty.remove("Anna");
```

5. Sprawdzanie

```
boolean istnieje =
    kontakty.containsKey("Anna");
```

Wynik: **true**

6. Wyświetlenie wszystkich

```
for (String imie : kontakty.keySet()) {
    System.out.println(
      imie + " -> " + kontakty.get(imie)
    );
}
```

7. Menu projektu

1. Dodaj kontakt
2. Znajdź kontakt
3. Usuń kontakt
4. Pokaż wszystkie
5. Wyjdź

8. Dlaczego Map pasuje?

Ponieważ odczytujemy wartość za pomocą unikalnego klucza.

Ponowne put z tym samym kluczem zastępuje numer.

Jeśli myślisz „znajdź wartość po identyfikatorze”, bardzo często warto rozważyć Map.

Projekt 6 — policz słowa w pliku tekstowym

1. Odczytaj plik

```
import java.io.IOException;
import java.nio.file.*;
import java.util.*;
```

Fragmenty jednej metody z throws IOException.

```
String tekst = Files.readString(
    Path.of("tekst.txt"));
```

2. Przygotuj tekst

```
tekst = tekst.toLowerCase(
    Locale.ROOT);
```

Zamień litery na małe.

3. Podziel na słowa

```
String[] slowa =
    tekst.split("\\s+");
```

Dzielimy według białych znaków.

ANALIZA TEKSTU

4. Mapa liczników

```
Map<String, Integer> licznik =
    new HashMap<>();
```

5. Liczenie

```
for (String slowo : slowa) {
    if (slowo.isEmpty()) continue;
    licznik.put(slowo,
        licznik.getOrDefault(
            slowo, 0) + 1);
}
```

6. Przykład wyniku

java	→	8
kod	→	5
program	→	3

7. Co tutaj ćwiczysz?

- Pliki
- Map
- String
- Pętle
- Tablice
- Agregację

8. Rozszerzenie

Możesz:

- Znaleźć najczęstsze słowo.
- Pomijać bardzo krótkie słowa.
- Sortować wyniki.

Uproszczenie: „java” i „java,” są liczone osobno.

To prosty przykład analizy danych: wczytaj → przetwórz → policz → pokaż wynik.

Projekt 7 — różne metody płatności przez interfejs

1. Interfejs

```
interface Platnosc {
    void zaplac(double kwota);
}
```

Interfejs określa kontrakt dla wszystkich metod płatności.

2. Implementacja karta

```
class Karta implements Platnosc {
    @Override
    public void zaplac(double kwota) {
        System.out.println(
          "Zapłacono kartą: " + kwota);
    }
}
```

3. Implementacja BLIK

```
class Blik implements Platnosc {
    @Override
    public void zaplac(double kwota) {
        System.out.println(
          "Zapłacono BLIK: " + kwota);
    }
}
```

BLIK

SYSTEM PŁATNOSCI

Symulacja — program tylko wypisuje komunikaty.

4. Logika wspólna

Metoda w klasie obsługi zamówień:

```
static void zaplacZaZamowienie(
    Platnosc platnosc, double kwota) {
    platnosc.zapłac(kwota);
}
```

5. Co zyskujemy?

- Nowe metody płatności bez zmiany logiki głównej.
- Elastyczny kod.
- Mniejsze zależności między klasami.

6. Co ćwiczysz?

- Interfejs
- implements
- Polimorfizm
- Metody
- Projektowanie kontraktów

DODAJ:

Przelew jako trzeci sposób płatności.

Projekt 8 — raport danych ze Stream API

1. Klasa

```
class Uzytkownik {
    String imie;
    int wiek;
    Uzytkownik(String imie, int wiek) {
      this.imie = imie;
      this.wiek = wiek;
    }
}
```

2. Lista

```
import java.util.List;

List<Uzytkownik> osoby = List.of(
    new Uzytkownik("anna", 20),
    new Uzytkownik("jan", 16),
    new Uzytkownik("adam", 25)
);
```

3. Zadanie

1. Wybierz pełnoletnich użytkowników.
2. Zamień ich imiona na wielkie litery.
3. Posortuj imiona.

4. Pipeline

```
List<String> wynik = osoby.stream()
    .filter(osoba -> osoba.wiek >= 18)
    .map(osoba -> osoba.imie)
    .map(String::toUpperCase)
    .sorted()
    .toList();
```

Wynik: `[ADAM, ANNA]`

Java 16+; wynik to lista niemodyfikowalna.

RAPORT UŻYTKOWNIKÓW

5. Jak czytać?

DANE
↓
ZOSTAW PEŁNOLETNICH
↓
POBIERZ IMIĘ
↓
ZMIEŃ ZAPIS
↓
POSORTUJ
↓
UTWÓRZ LISTĘ

6. Co tutaj ćwiczysz?

- Obiekty
- Kolekcje
- Lambdy
- filter
- map
- sorted
- toList

7. Ważne

Każdy etap powinien wykonywać jedno jasno określone zadanie.

Dobry pipeline Stream API czyta się jak instrukcja przetwarzania danych.

Debugowanie — jak znaleźć błąd zamiast zgadywać

1. Nie zgaduj — obserwuj

Zapytaj:

- Gdzie wynik przestaje być poprawny?
- Jaka wartość znajduje się w zmiennej?
- Czy warunek jest prawdziwy?
- Ile razy wykonuje się pętla?



2. Czytaj komunikat błędu

Szukaj:

- Typ wyjątku
- Treść komunikatu
- Wskazana linia kodu



3. Dodaj tymczasowe informacje

```
int x = 7;
System.out.println(
    "wartosc x = " + x
);
```

Wynik: `wartosc x = 7`

4. Debugger

- Breakpoint
- Krok po kroku
- Podgląd zmiennych
- Wejście do metody
- Kontynuacja wykonania



DEBUGOWANIE

5. Zmniejsz problem

Gdy duży program zawodzi:

znajdź mały przykład, który odtwarza błąd.



6. Sprawdź założenia

Przykładowe pytania:

- Czy lista naprawdę nie jest pusta?
- Czy indeks jest poprawny?
- Czy wartość nie jest null?
- Czy warunek działa tak, jak myślisz?



7. Popraw przyczynę, nie tylko objaw

BŁĄD → ODTWÓRZ → ZLOKALIZUJ → ZROZUM → NAPRAW → SPRAWDŹ PONOWNIE

Debugowanie to proces zbierania dowodów o tym, co naprawdę robi program.

Testowanie — sprawdź więcej niż jeden „dobry” przypadek

1. Przypadek typowy

Wywołanie:

```
dodaj(2, 3)
```

Oczekiwany wynik: 5



2. Wartości graniczne

Zapytaj:

- 0?
- Wartość minimalna?
- Wartość maksymalna?
- Pusta lista?



3. Niepoprawne dane

Zależnie od reguł programu:

- Liczba ujemna
- null
- Zły format
- Nieistniejący plik



4. Różne ścieżki warunków

Schemat — pseudokod:

```
if (...) {
   ...
} else {
   ...
}
```



Przetestuj obie gałęzie.

TESTUJ ŚWIADOMIE

5. Przykład testowego myślenia

Nagłówek metody do przetestowania:

```
boolean pelnoletni(int wiek)
```

Sprawdź:

- 17 → false
- 18 → true
- 19 → true
- Wartość ujemna → co powinno się stać?



6. Test jednostkowy — idea



7. Dobra zasada

Nie pytaj tylko: „Czy działa?”

Zapytaj też: „W jakich sytuacjach może przestać działać?”



TESTOWANIE TO SPRAWDZANIE ZAŁOŻEŃ

Refaktoryzacja — popraw kod bez zmiany jego działania

1. Co to znaczy?

Refaktoryzacja = poprawianie struktury kodu bez zmiany tego, co program robi.

Fragmenty tej samej klasy. Zmienne wejściowe mają już wartości.

2. Przed refaktoryzacją

```
double x = cena * ilosc;
double y = x * 0.23;
double z = x + y;
System.out.println(z);
```

Problemy:
- niejasne nazwy;
- kilka odpowiedzialności razem;
- trudniej zrozumieć intencję.

3. Po refaktoryzacji

Obliczenia wewnątrz metody:

```
double wartoscNetto = cena * ilosc;
double podatek =
    obliczPodatek(wartoscNetto);
double wartoscBrutto = wartoscNetto + podatek;
System.out.println(wartoscBrutto);
```

Metoda pomocnicza w klasie:

```
static double obliczPodatek(double netto) {
    return netto * 0.23;
}
```

TEN SAM WYNIK

LEPSZA CZYTELNOŚĆ

REFAKTORYZACJA

4. Typowe refaktoryzacje

- zmień nazwę zmiennej;
- wydziel metodę;
- usuń duplikację;
- uprość warunek;
- podziel dużą klasę;
- usuń niepotrzebny kod.

5. Powtarzający się kod

PRZED:

```
System.out.println("Imię: " + imie1);
System.out.println("Imię: " + imie2);
System.out.println("Imię: " + imie3);
```

Wywołania wewnątrz metody:

```
pokazImie(imie1);
pokazImie(imie2);
pokazImie(imie3);
```

PO — metoda w klasie:

```
static void pokazImie(String imie) {
    System.out.println("Imię: " + imie);
}
```

6. Kiedy refaktoryzować?

- gdy kod działa;
- gdy rozumiesz jego zachowanie;
- małymi krokami;
- najlepiej z testami.

Nie zmieniaj jednocześnie funkcjonalności i całej struktury, jeśli nie musisz — trudniej wtedy znaleźć źródło problemu.

NAJPIERW NIECH DZIAŁA → POTEM NIECH BĘDZIE CZYTELNY

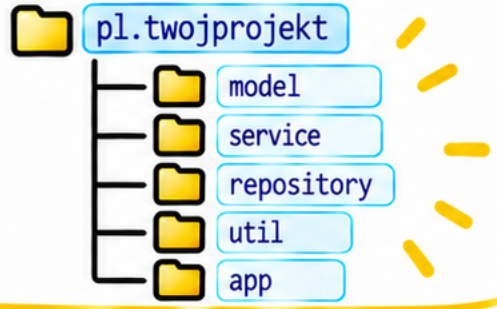
Pakiety — uporządkuj większy projekt

1. Problem

Mały projekt:
- 3 klasy;
- łatwo znaleźć pliki.

Większy projekt:
20, 50, 100+ klas.
Bez organizacji → chaos.

2. Pakiet grupuje powiązane klasy



3. Przykład deklaracji

Plik Produkt.java:

```
package pl.twojprojekt.model;

public class Produkt {}
```

PACKAGE

4. Przykładowy podział

model
- Produkt
- Uzytkownik
- Zamowienie

service
- ProduktService
- PlatnoscService

util
- pomocnicze narzędzia

app
- uruchomienie programu

5. Import

Publiczną klasę z innego pakietu można zaimportować:

```
import pl.twojprojekt.model.Produkt;
```

Import nie zmienia zasad dostępu.

6. Po co pakiety?

- porządek;
- łatwiejsza nawigacja;
- uniknięcie konfliktów nazw;
- logiczny podział projektu;
- lepsza współpraca zespołu.

7. Dobra zasada

Nie twórz pakietów tylko po to, by wyglądały „profesjonalnie”. Powinny odzwierciedlać rzeczywistą organizację projektu.

DUŻY PROJEKT → podziel według odpowiedzialności → PAKIETY → zawierają → POWIĄZANE KLASY

Im większy projekt, tym ważniejsza staje się dobra organizacja plików i odpowiedzialności.

Java w jednej mapie — połącz wszystkie elementy

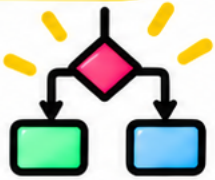
1. FUNDAMENTY

- JDK / JVM
- składnia
- zmienne
- typy
- operatory



2. LOGIKA

- if
- switch
- warunki
- for
- while



3. METODY

- parametry
- argumenty
- return
- zakres
- przeciążanie



4. DANE

- String
- tablice
- indeksy
- przetwarzanie



PROGRAM JAVA

5. KOLEKCJE

- List
- Set
- Map
- generyki



6. OOP

- klasy
- obiekty
- konstruktor
- enkapsulacja



7. ELASTYCZNE OOP

- dziedziczenie
- interfejsy
- polimorfizm
- kompozycja



8. BEZPIECZEŃSTWO PROGRAMU

- wyjątki
- try-catch
- walidacja
- pliki

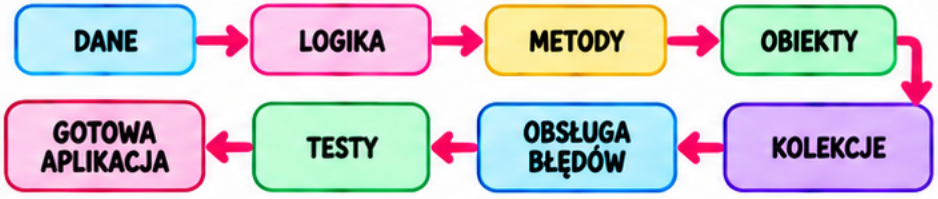


9. NOWOCZESNA JAVA

- lambdy
- Stream API
- Optional
- operacje na danych



JAK POWSTAJE PROGRAM?





Schemat poglądowy — elementy dobierasz do problemu.

Nie musisz pamiętać wszystkiego.

Programista regularnie:

- sprawdza dokumentację;
- wraca do przykładów;
- testuje rozwiązania;
- poprawia kod.



NAJWAŻNIEJSZE JEST ROZUMIENIE, JAK ELEMENTY ŁĄCZĄ SIĘ ZE SOBĄ

Mapa 150 — co dalej? Twoja droga z Javą dopiero się zaczyna

MASZ JUŻ FUNDAMENTY. TERAZ BUDUJ.

1 ETAP 1 — POWTÓRZ PODSTAWY

Upewnij się, że rozumiesz:

- zmienne;
- warunki;
- pętle;
- metody;
- kolekcje.



2 ETAP 2 — ĆWICZ OOP

Buduj własne klasy:

- Produkt;
- Uzytkownik;
- Konto;
- Zamowienie.



Zawsze pytaj:

- Jakie dane ma obiekt?
- Co powinien potrafić zrobić?

3 ETAP 3 — BUDUJ MAŁE PROJEKTY

Przykłady:

- kalkulator;
- lista zadań;
- magazyn;
- system ocen;
- książka kontaktów.



Nie czekaj na „idealny" projekt. Małe projekty też wiele uczą.

4 ETAP 4 — CZYTAJ CUDZY KOD

Zwracaj uwagę na:

- nazwy;
- podział kodu na metody;
- klasy;
- błędy;
- rozwiązania innych programistów.



5 ETAP 5 — IDŹ DALEJ

Po opanowaniu podstaw stopniowo poznawaj:

- testy jednostkowe;
- Maven / Gradle;
- Git;
- SQL;
- REST API;
- Spring;
- projekty backendowe.



JAK SIĘ UCZYĆ?

MAPA → zrozumienie
↓
KOD → praktyka
↓
BŁĄD → analiza
↓
POPRAWA → doświadczenie
↓
PROJEKT → rozwój



NIE MUSISZ ZNAĆ 150 MAP NA PAMIĘĆ

Mogą służyć jako:

- przewodnik;
- szybka powtórka;
- punkt odniesienia;
- pomoc podczas pisania kodu.





OTWÓRZ IDE. NAPISZ KOD. POPEŁNIJ BŁĄD. ZNAJDŹ ROZWIĄZANIE. NAPISZ KOLEJNY PROGRAM.

Tak naprawdę uczy się programowania.