Klasy i Pliki

1. Plik i Klasa Publiczna

Zasada:

- W Javie każdy plik może zawierać wiele klas, ale tylko jedna z nich może być oznaczona jako `public`.
- Nazwa pliku **musi** być taka sama jak nazwa klasy publicznej, z rozszerzeniem `.java`.

Dlaczego?

- Dzięki temu kompilator i środowisko programistyczne łatwo identyfikują główną klasę w pliku.

Przykład:

- **Plik:** `Main.java`

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

Dodatkowy Przykład z Wieloma Klasami:

- **Plik:** `Main.java`

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

class Helper {
    public static void sayHello() {
        System.out.println("Hello from Helper");
    }
}
```

- **Uruchamianie:**

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

- **Wynik:**

```
Hello from Main
Hello from Helper
```

2. Funkcja `main` i `static`

Funkcja `main`:

- Jest to punkt wejścia programu Java.
- Musi być zdefiniowana jako:

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

- Chociaż można wykonywać kod bez `main` (np. w blokach statycznych), `main` jest niezbędny do uruchomienia aplikacji jako programu.

`static`:

- Słowo kluczowe `static` oznacza, że metoda lub zmienna należy do klasy, a nie do konkretnego obiektu.
- Pozwala na wywoływanie metod bez konieczności tworzenia obiektu klasy.

Dlaczego używamy `static`?

- Aby mieć dostęp do metod i zmiennych bez tworzenia instancji klasy, co jest przydatne np. w metodzie `main`.

Dodatkowy Przykład z `static`:

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

public class Main {
    public static void main(String[] args) {
        int sum = MathUtils.add(5, 3);
        System.out.println("Sum: " + sum); // Sum: 8
    }
}
```

Przykład z `static` Zmiennymi:

```
class Counter {
    public static int count = 0;
```

```
    public Counter() {
        count++;
    }
}

public class Main {
    public static void main(String[] args) {
        new Counter();
        new Counter();
        System.out.println("Count: " + Counter.count); // Count: 2
    }
}
```

Dodatkowy Przykład z Blokiem Statycznym: Bloki statyczne są wykonywane przy ładowaniu klasy, zanim jakiekolwiek obiekty zostaną utworzone.

```
class Initialization {
    static {
        System.out.println("Blok statyczny Initialization");
    }
}

public class Main {
    public static void main(String[] args) {
        System.out.println("Start programu");
        Initialization obj = new Initialization();
    }
}
```

Wynik:

```
Blok statyczny Initialization
Start programu
```

Kompilacja i Bytecode

1. Jak działa Java?

Kompilacja do Bytecode:

- Java jest językiem kompilowanym do bytecode, który jest niezależny od platformy.
- Bytecode jest zapisany w plikach `.class`.

Wykonanie przez JVM:

- JVM (Java Virtual Machine) interpretuje bytecode i wykonuje go na konkretnym systemie operacyjnym.

Dlaczego Bytecode?

- Umożliwia uruchamianie tego samego kodu na różnych platformach (Windows, Linux, macOS) bez konieczności rekompilacji.

Proces Kompilacji:

1. **Kod źródłowy:** `Main.java` (kod czytelny dla programisty)
2. **Kompilacja:** `javac Main.java` → **Bytecode:** `Main.class` (niezależny od platformy)
3. **Wykonanie:** `java Main` → JVM interpretuje `Main.class` i uruchamia program

Dodatkowy Przykład:

- **Plik:** `Hello.java`

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

- **Kompilacja:**

```
javac Hello.java
```

  - Tworzy `Hello.class`
- **Uruchomienie:**

```
java Hello
```

- **Wynik:**

```
Hello, Bytecode!
```

Uruchamianie Bez Funkcji `main`:

- Można użyć bloków statycznych lub bibliotek do uruchomienia kodu bez tradycyjnej funkcji `main`.
- **Przykład:**

```
public class NoMain {
    static {
        System.out.println("Uruchomiono kod bez funkcji main!");
        System.exit(0);
```

```
    }
}
```

- **Uruchomienie:**

```
java NoMain
```

- **Wynik:**

```
Uruchomiono kod bez funkcji main!
```

Zmienne i Typy Danych

1. Typy Prymitywne

Opis:

- Typy prymitywne to podstawowe typy danych w Javie, przechowujące proste wartości.

Lista Typów Prymitywnych:

1. **byte**
    - **Zakres:** -128 do 127
    - **Rozmiar:** 1 bajt
    - **Przykład:**

```
byte b = 100;
System.out.println(b); // 100
```

2. **short**
    - **Zakres:** -32,768 do 32,767
    - **Rozmiar:** 2 bajty
    - **Przykład:**

```
short s = 10000;
System.out.println(s); // 10000
```

3. **int**
    - **Zakres:** -2,147,483,648 do 2,147,483,647

- **Rozmiar:** 4 bajty
- **Przykład:**

```
int i = 100000;
System.out.println(i); // 100000
```

4. **long**

- **Zakres:** -9,223,372,036,854,775,808 do 9,223,372,036,854,775,807
- **Rozmiar:** 8 bajtów
- **Przykład:**

```
long l = 100000L; // Dodajemy 'L', aby oznaczyć jako long
System.out.println(l); // 100000
```

5. **float**

- **Zakres:** Liczby zmiennoprzecinkowe z precyzją do 6-7 cyfr
- **Rozmiar:** 4 bajty
- **Przykład:**

```
float f = 5.75f; // Dodajemy 'f', aby oznaczyć jako float
System.out.println(f); // 5.75
```

6. **double**

- **Zakres:** Liczby zmiennoprzecinkowe z precyzją do 15-16 cyfr
- **Rozmiar:** 8 bajtów
- **Przykład:**

```
double d = 19.99;
System.out.println(d); // 19.99
```

7. **char**

- **Opis:** Przechowuje pojedynczy znak Unicode
- **Rozmiar:** 2 bajty
- **Przykład:**

```
char c = 'A';
System.out.println(c); // A
```

8. **boolean**
    - **Opis:** Przechowuje wartość `true` lub `false`
    - **Rozmiar:** Niezdefiniowany (zależy od implementacji)
    - **Przykład:**

```
boolean isJavaFun = true;
System.out.println(isJavaFun); // true
```

Dodatkowe Przykłady z Typami Prymitywnymi:

- **byte:**

```
byte age = 25;
System.out.println("Age: " + age); // Age: 25
```

- **short:**

```
short temperature = -15;
System.out.println("Temperature: " + temperature); // Temperature: -15
```

- **int:**

```
int population = 7800000000;
System.out.println("Population: " + population); // Population: 7800000000
```

- **long:**

```
long distance = 10000000000L;
System.out.println("Distance: " + distance); // Distance: 10000000000
```

- **float:**

```
float pi = 3.14f;
System.out.println("Pi: " + pi); // Pi: 3.14
```

- **double:**

```
```

```
double gravity = 9.81;
System.out.println("Gravity: " + gravity); // Gravity: 9.81
```

- **char:**

```
char grade = 'A';
System.out.println("Grade: " + grade); // Grade: A
```

- **boolean:**

```
boolean isRaining = false;
System.out.println("Is it raining? " + isRaining); // Is it raining? false
```

2. Typy Referencyjne

Opis:

- Typy referencyjne przechowują odniesienia (adresy) do obiektów w pamięci, a nie same wartości.
- Są używane do przechowywania bardziej złożonych danych.

Przykłady Typów Referencyjnych:

- **String**
- **Arrays (Tablice)**
- **Klasy (np. `Integer`, `Scanner`, własne klasy)**

Cechy Typów Referencyjnych:

- **Niemutowalność `String`:** Obiekty typu `String` nie mogą być zmienione po utworzeniu. Każda operacja modyfikująca `String` tworzy nowy obiekt.

```
String text = "Hello";
text = text.toUpperCase();
System.out.println(text); // HELLO
```

- **Null:** Zmienna typu referencyjnego może mieć wartość `null`, co oznacza brak odwołania do obiektu.

```
String name = null;
System.out.println(name); // null
```

Dodatkowe Przykłady Typów Referencyjnych:

- **Klasa `Integer`:**

```
Integer num = 10;
System.out.println(num); // 10
```

- **Tablica:**

```
int[] numbers = {1, 2, 3, 4, 5};
System.out.println(numbers[2]); // 3
```

- **Tworzenie Obiektu Własnej Klasy:**

```
class Person {
    String name;
    int age;

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

public class Main {
    public static void main(String[] args) {
        Person person = new Person("Alice", 30);
        System.out.println(person.name + " is " + person.age + " years
old."); // Alice is 30 years old.
    }
}
```

3. Deklaracja Stałej

Opis:

- Słowo kluczowe `final` oznacza, że wartość zmiennej nie może być zmieniona po jej inicjalizacji.

Przykład:

```
public class Constants {
    public static final int MY_NUM = 15;

    public static void main(String[] args) {
        System.out.println("MY_NUM: " + MY_NUM); // MY_NUM: 15
        // MY_NUM = 20; // Błąd! Nie można zmienić wartości stałej
    }
}
```

Dodatkowy Przykład z Obiektami:

- Stała referencyjna oznacza, że referencja nie może być zmieniona, ale obiekt może być modyfikowany (jeśli jest mutowalny).

```
public class FinalObject {
    public static final StringBuilder sb = new StringBuilder("Hello");

    public static void main(String[] args) {
        sb.append(", World!");
        System.out.println(sb); // Hello, World!
        // sb = new StringBuilder(); // Błąd! Nie można zmienić referencji
    }
}
```

4. Rzutowanie (Casting)

Opis:

- Rzutowanie to proces konwersji jednej typu danych na inny.

Rodzaje Rzutowania:

1. **Rzutowanie Jawne (Explicit Casting):** Konwersja wymaga użycia nawiasów i jest konieczna, gdy konwertujemy typ o większym zakresie na mniejszy (np. `double` na `int`).

```
double myDouble = 9.78;
int myInt = (int) myDouble; // Rzutowanie z double na int
System.out.println(myInt); // 9
```

2. **Rzutowanie Niejawne (Implicit Casting):** Automatyczne konwertowanie typów o mniejszym zakresie na większy (np. `int` na `double`).

```
int myInt = 9;
double myDouble = myInt; // Automatyczne rzutowanie z int na double
System.out.println(myDouble); // 9.0
```

Przykłady Rzutowania:

- **Rzutowanie z `int` na `double`:**

```
int a = 5;
double b = a; // Automatyczne rzutowanie
System.out.println(b); // 5.0
```

- **Rzutowanie z `double` na `int`:**

```
double x = 9.99;
int y = (int) x; // Rzutowanie jawne
System.out.println(y); // 9
```

- **Rzutowanie Obiektów:**
  - **Upcasting (rzutowanie do typu bazowego):** Zawsze bezpieczne.

```
class Animal {}
class Dog extends Animal {}

public class Main {
    public static void main(String[] args) {
        Dog dog = new Dog();
        Animal animal = dog; // Upcasting
        System.out.println(animal instanceof Animal); // true
    }
}
```

  - **Downcasting (rzutowanie do typu pochodnego):** Wymaga sprawdzenia typu za pomocą `instanceof`.

```
public class Main {
    public static void main(String[] args) {
        Animal animal = new Dog(); // Upcasting
        if (animal instanceof Dog) {
            Dog dog = (Dog) animal; // Downcasting
            System.out.println("To jest pies.");
        }
    }
}
```

Uwaga:

- Niewłaściwe rzutowanie może prowadzić do `ClassCastException` podczas działania programu.

Operacje na Stringach

1. Przykłady Metod na `String`

Opis:

- `String` to klasa reprezentująca ciągi znaków. Jest niemutowalna, co oznacza, że operacje na `String` tworzą nowe obiekty.

Najczęściej Używane Metody:

1. `length()`
   - **Opis:** Zwraca długość (liczbę znaków) Stringa.
   - **Przykład:**

```
String text = "Ala ma kota";
System.out.println(text.length()); // 11
```

2. `toUpperCase()`
   - **Opis:** Zwraca nowy String z zamienionymi literami na wielkie.
   - **Przykład:**

```
String text = "Ala ma kota";
System.out.println(text.toUpperCase()); // ALA MA KOTA
```

3. `toLowerCase()`
   - **Opis:** Zwraca nowy String z zamienionymi literami na małe.
   - **Przykład:**

```
String text = "Ala ma kota";
System.out.println(text.toLowerCase()); // ala ma kota
```

4. `indexOf(String str)`
   - **Opis:** Zwraca indeks pierwszego wystąpienia podciągu `str` w Stringu. Jeśli nie znaleziono, zwraca -1.
   - **Przykład:**

```
String text = "Ala ma kota";
System.out.println(text.indexOf("ma")); // 4
System.out.println(text.indexOf("pies")); // -1
```

5. `substring(int beginIndex, int endIndex)`
   - **Opis:** Zwraca podciąg od `beginIndex` (włącznie) do `endIndex` (wyłącznie).
   - **Przykład:**

```
String text = "Hello World";
System.out.println(text.substring(0, 5)); // Hello
```

6. `replace(char oldChar, char newChar)`
    - **Opis:** Zwraca nowy String, w którym wszystkie wystąpienia `oldChar` są zastąpione przez `newChar`.
    - **Przykład:**

```
String text = "banana";
System.out.println(text.replace('a', 'o')); // bonono
```

7. `contains(CharSequence s)`
    - **Opis:** Sprawdza, czy String zawiera podciąg `s`.
    - **Przykład:**

```
String text = "Java Programming";
System.out.println(text.contains("Programming")); // true
System.out.println(text.contains("Python")); // false
```

8. `startsWith(String prefix)` **i** `endsWith(String suffix)`
    - **Opis:** Sprawdza, czy String zaczyna się od `prefix` lub kończy się na `suffix`.
    - **Przykład:**

```
String text = "Hello World";
System.out.println(text.startsWith("Hell")); // true
System.out.println(text.endsWith("World")); // true
```

Dodatkowe Przykłady:

- **Łączenie Stringów (`concat`):**

```
String a = "Hello";
String b = "World";
String c = a.concat(" ").concat(b);
System.out.println(c); // Hello World
```

- **Porównywanie Stringów (`equals` i `equalsIgnoreCase`):**

```
String str1 = "Java";
String str2 = "java";
System.out.println(str1.equals(str2)); // false
System.out.println(str1.equalsIgnoreCase(str2)); // true
```

- **Konwersja do Tablicy Znaków (`toCharArray`):**

```
String text = "Java";
char[] chars = text.toCharArray();
for (char c : chars) {
    System.out.println(c);
}
// Wynik:
// J
// a
// v
// a
```

2. Łączenie Stringów

Opis:

- Łączenie (konkatenacja) Stringów polega na łączeniu dwóch lub więcej Stringów w jeden.

Sposoby Łączenia Stringów:

1. **Operator `+`**
   - Najprostszy sposób na łączenie Stringów.
   - **Przykład:**

```
String firstName = "John";
String lastName = "Doe";
String fullName = firstName + " " + lastName;
System.out.println(fullName); // John Doe
```

2. **Metoda `concat`**
   - Metoda klasy `String` do łączenia Stringów.
   - **Przykład:**

```
String a = "Hello";
String b = "World";
String c = a.concat(" ").concat(b);
System.out.println(c); // Hello World
```

3. **StringBuilder i StringBuffer**
   - Używane do efektywnego budowania Stringów w pętlach.
   - **Przykład z `StringBuilder`:**

```
StringBuilder sb = new StringBuilder();
sb.append("Hello");
sb.append(" ");
sb.append("World");
String result = sb.toString();
System.out.println(result); // Hello World
```

- **Przykład z `StringBuffer` (wielowątkowy):**

```
StringBuffer sb = new StringBuffer();
sb.append("Hello");
sb.append(" ");
sb.append("World");
String result = sb.toString();
System.out.println(result); // Hello World
```

Dodatkowe Przykłady Łączenia Stringów:

- **Dynamiczne Łączenie w Pętli:**

```
String[] words = {"Java", "is", "awesome"};
StringBuilder sentence = new StringBuilder();
for (String word : words) {
    sentence.append(word).append(" ");
}
System.out.println(sentence.toString().trim()); // Java is awesome
```

- **Łączenie z Liczbami:**

```
String name = "Alice";
int age = 25;
String message = name + " ma " + age + " lat.";
System.out.println(message); // Alice ma 25 lat.
```

Tablice i Kolekcje

1. Tablice

Opis:

- Tablice przechowują wiele wartości tego samego typu w jednym miejscu.
- Mają stały rozmiar po utworzeniu.

Deklaracja i Inicjalizacja:

1. **Statyczna (z inicjalizacją):**

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

2. **Dynamiczna (bez inicjalizacji):**

```
int[] myNum = new int[4]; // Tworzy tablicę o rozmiarze 4
myNum[0] = 10;
myNum[1] = 20;
myNum[2] = 30;
myNum[3] = 40;
```

Dostęp do Elementów:

- Indeksowanie zaczyna się od 0.

```
int[] myNum = {10, 20, 30, 40};
System.out.println(myNum[0]); // 10
System.out.println(myNum[3]); // 40
```

Długość Tablicy:

- Każda tablica ma atrybut `length`, który zwraca jej rozmiar.

```
int[] myNum = {10, 20, 30, 40};
System.out.println(myNum.length); // 4
```

Tablice Wielowymiarowe:

- Tablice mogą mieć więcej niż jeden wymiar.
- Najczęściej używane są tablice dwuwymiarowe.

Przykład Tablicy Dwuwymiarowej:

```
int[][] myNumbers = {
    {1, 2, 3},
    {4, 5, 6}
};
System.out.println(myNumbers[0][0]); // 1
System.out.println(myNumbers[1][2]); // 6
```

Iteracja po Tablicy:

- **Za pomocą pętli `for`:**

```
int[] myNum = {10, 20, 30, 40};
for (int i = 0; i < myNum.length; i++) {
    System.out.println(myNum[i]);
}
```

- **Za pomocą pętli `for-each`:**

```
int[] myNum = {10, 20, 30, 40};
for (int num : myNum) {
    System.out.println(num);
}
```

Dodatkowe Przykłady Tablic:

- **Tablica Stringów:**

```
String[] fruits = {"Apple", "Banana", "Cherry"};
for (String fruit : fruits) {
    System.out.println(fruit);
}
// Wynik:
// Apple
// Banana
// Cherry
```

- **Tablica Wielowymiarowa z Większą Liczbą Wierszy:**

```
int[][] matrix = new int[3][3];
matrix[0][0] = 1;
matrix[1][1] = 5;
matrix[2][2] = 9;

for (int[] row : matrix) {
    for (int num : row) {
        System.out.print(num + " ");
    }
    System.out.println();
}
// Wynik:
// 1 0 0
// 0 5 0
// 0 0 9
```

2. ArrayList

Opis:

- `ArrayList` to dynamiczna struktura danych, która może zmieniać swój rozmiar w czasie działania programu.
- Przechowuje elementy w określonej kolejności i pozwala na duplikaty.

Importowanie `ArrayList`:

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

Deklaracja i Inicjalizacja:

```
ArrayList<String> cars = new ArrayList<String>();
```

Podstawowe Operacje:

1. **Dodawanie Elementów (`add`):**

   ```
   cars.add("Volvo");
   cars.add("BMW");
   cars.add("Ford");
   ```

2. **Dostęp do Elementów (`get`):**

   ```
   System.out.println(cars.get(0)); // Volvo
   ```

3. **Usuwanie Elementów (`remove`):**

   ```
   cars.remove(0); // Usuwa "Volvo"
   System.out.println(cars.get(0)); // BMW
   ```

4. **Sprawdzanie Rozmiaru (`size`):**

   ```
   System.out.println(cars.size()); // 2
   ```

5. **Iteracja po `ArrayList`:**
   - **Za pomocą pętli `for`:**

     ```
     ```

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

- **Za pomocą pętli `for-each`:**

```
for (String car : cars) {
    System.out.println(car);
}
```

Dodatkowe Przykłady z `ArrayList`:

- **Dodawanie na Konkretne Miejsce:**

```
cars.add(1, "Audi"); // Wstawia "Audi" na indeks 1
System.out.println(cars); // [BMW, Audi, Ford]
```

- **Sprawdzanie, czy `ArrayList` Zawiera Element:**

```
boolean hasBMW = cars.contains("BMW");
System.out.println(hasBMW); // true
```

- **Usuwanie Elementu po Wartości:**

```
cars.remove("Audi");
System.out.println(cars); // [BMW, Ford]
```

- **Konwersja `ArrayList` na Tablicę:**

```
String[] carArray = new String[cars.size()];
carArray = cars.toArray(carArray);
for (String car : carArray) {
    System.out.println(car);
}
// Wynik:
// BMW
// Ford
```

Pętle

1. Pętla For-each

Opis:

- Umożliwia iterację po elementach kolekcji lub tablicy bez potrzeby używania indeksów.
- Jest bardziej czytelna i mniej podatna na błędy niż tradycyjna pętla `for`.

Składnia:

```
for (typ_elementu element : kolekcja) {
    // Kod do wykonania
}
```

Przykład:

```
String[] cars = {"Volvo", "BMW", "Ford"};
for (String car : cars) {
    System.out.println(car);
}
// Wynik:
Volvo
BMW
Ford
```

Dodatkowy Przykład z ArrayList:

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<String> fruits = new ArrayList<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Cherry");

        for (String fruit : fruits) {
            System.out.println(fruit);
        }
        // Wynik:
        // Apple
        // Banana
        // Cherry
    }
}
```

2. Tablica Wielowymiarowa

Opis:

- Tablice mogą mieć więcej niż jeden wymiar, co pozwala na przechowywanie danych w formie siatki (np. macierzy).

Przykład Tablicy Dwuwymiarowej:

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

for (int[] row : myNumbers) {
    for (int num : row) {
        System.out.println(num);
    }
}
// Wynik:
1
2
3
4
5
6
```

Dodatkowy Przykład z Tablicą 3-Wymiarową:

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

for (int[][] matrix : threeD) {
    for (int[] row : matrix) {
        for (int num : row) {
            System.out.println(num);
        }
    }
}
// Wynik:
1
2
3
4
```

```
5
6
7
8
```

Zastosowanie Tablic Wielowymiarowych:

- **Matematyka:** Macierze
- **Gry:** Plansze do gier (np. szachy, tic-tac-toe)
- **Przetwarzanie Obrazów:** Piksele obrazu

Przykład Zastosowania w Grze Tic-Tac-Toe:

```
public class TicTacToe {
    public static void main(String[] args) {
        char[][] board = {
            {'X', 'O', 'X'},
            {' ', 'X', ' '},
            {'O', ' ', 'O'}
        };

        for (char[] row : board) {
            for (char cell : row) {
                System.out.print(cell + " | ");
            }
            System.out.println();
        }
        // Wynik:
        // X | O | X |
        //   | X |   |
        // O |   | O |
    }
}
```

Uwagi:

- Indeksy tablic zaczynają się od 0.
- Przekroczenie zakresu tablicy (np. `myNumbers[2][0]` w powyższym przykładzie) spowoduje `ArrayIndexOutOfBoundsException`.

OOP w Javie (Programowanie Obiektowe)

1. Tworzenie Klasy

Opis:

- Klasa to szablon (plan) dla obiektów. Definiuje właściwości (pola) i zachowania (metody).

Przykład Tworzenia Klasy:

```java
public class MojaKlasa {
    int x = 5; // Pole
    final int STALA = 10; // Stała pole
    int z; // Pole

    // Konstruktor
    public MojaKlasa(int z) {
        this.z = z;
    }

    // Metoda
    public void display() {
        System.out.println("x: " + x + ", z: " + z);
    }
}
```

Tworzenie Obiektu Klasy:

```java
public class Main {
    public static void main(String[] args) {
        MojaKlasa obj = new MojaKlasa(15);
        obj.display(); // x: 5, z: 15
    }
}
```

Dodatkowy Przykład z Metodami i Polami:

```java
class Rectangle {
    double length;
    double width;

    // Konstruktor
    Rectangle(double length, double width) {
        this.length = length;
        this.width = width;
    }

    // Metoda do obliczania powierzchni
    double area() {
        return length * width;
    }

    // Metoda do obliczania obwodu
    double perimeter() {
        return 2 * (length + width);
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Rectangle rect = new Rectangle(5.0, 3.0);
        System.out.println("Area: " + rect.area()); // Area: 15.0
        System.out.println("Perimeter: " + rect.perimeter()); // Perimeter: 16.0
    }
}
```

2. Modyfikatory Dostępu

Opis:

- Modyfikatory dostępu kontrolują widoczność pól i metod klasy.

Rodzaje Modyfikatorów Dostępu:

1. `public`
    - **Opis:** Dostępny wszędzie.
    - **Przykład:**

```
public class PublicClass {
    public int number;

    public void display() {
        System.out.println("Number: " + number);
    }
}
```

2. `private`
    - **Opis:** Dostępny tylko w obrębie tej samej klasy.
    - **Przykład:**

```
public class PrivateClass {
    private int secret;

    private void revealSecret() {
        System.out.println("Secret: " + secret);
    }

    public void setSecret(int secret) {
        this.secret = secret;
    }
}
```

        - **Dostęp Zewnętrzny:**

```
public class Main {
    public static void main(String[] args) {
        PrivateClass obj = new PrivateClass();
        // obj.secret = 10; // Błąd! Pole jest prywatne
        obj.setSecret(10); // Poprawne
    }
}
```

3. `protected`
   - **Opis:** Dostępny w obrębie tej samej klasy, pakietu oraz klas dziedziczących.
   - **Przykład:**

```
public class ProtectedClass {
    protected int data;

    protected void showData() {
        System.out.println("Data: " + data);
    }
}

class SubClass extends ProtectedClass {
    public void display() {
        data = 20;
        showData(); // Data: 20
    }
}
```

     - **Dostęp Zewnętrzny (nie będący podklasą):**

```
public class Main {
    public static void main(String[] args) {
        ProtectedClass obj = new ProtectedClass();
        // obj.data = 10; // Błąd! Dostęp tylko w obrębie pakietu
lub podklas
    }
}
```

4. **Pakietowy Dostęp Domyślny (Bez Modyfikatora)**
   - **Opis:** Dostępny tylko w obrębie tego samego pakietu.
   - **Przykład:**

```
class PackageClass {
    int value; // Dostęp pakietowy
```

```
    void display() {
        System.out.println("Value: " + value);
    }
}
```

- **Dostęp Zewnętrzny (inny pakiet):**

```
// W innym pliku, innym pakiecie
public class Main {
    public static void main(String[] args) {
        PackageClass obj = new PackageClass();
        // obj.value = 10; // Błąd! Dostęp tylko w obrębie pakietu
    }
}
```

Dlaczego Używamy Modyfikatorów Dostępu?

- **Enkapsulacja:** Chroni dane przed nieautoryzowanym dostępem i modyfikacją.
- **Modularność:** Umożliwia tworzenie modułów, które są łatwiejsze do utrzymania.
- **Bezpieczeństwo:** Zapobiega przypadkowemu uszkodzeniu danych przez inne części programu.

3. Dziedziczenie

Opis:

- Dziedziczenie umożliwia tworzenie nowej klasy (podklasy), która dziedziczy właściwości i metody z innej klasy (klasy bazowej).

Zalety Dziedziczenia:

- **Reużywalność:** Umożliwia ponowne użycie kodu.
- **Hierarchia:** Tworzy naturalną hierarchię klas.

Przykład Dziedziczenia:

```
class Vehicle {
    protected String brand = "Ford";

    public void honk() {
        System.out.println("Tuut, tuut!");
    }
}

class Car extends Vehicle {
    private String model = "Mustang";

    public void display() {
        System.out.println(brand + " " + model);
    }
```

```
}

public class Main {
    public static void main(String[] args) {
        Car myCar = new Car();
        myCar.honk(); // Tuut, tuut!
        myCar.display(); // Ford Mustang
    }
}
```

Dodatkowy Przykład z Wielokrotnym Dziedziczeniem (Interfejsami):

- Java **nie** obsługuje wielokrotnego dziedziczenia klas, ale pozwala na implementację wielu interfejsów.

```
interface Flyable {
    void fly();
}

interface Drivable {
    void drive();
}

class FlyingCar implements Flyable, Drivable {
    public void fly() {
        System.out.println("Flying!");
    }

    public void drive() {
        System.out.println("Driving!");
    }
}

public class Main {
    public static void main(String[] args) {
        FlyingCar fc = new FlyingCar();
        fc.fly();   // Flying!
        fc.drive(); // Driving!
    }
}
```

Overriding Metod:

- Podklasa może nadpisać metodę z klasy bazowej, aby zmienić jej zachowanie.

```
class Animal {
    public void sound() {
        System.out.println("Animal makes a sound");
    }
}
```

```
class Dog extends Animal {
    @Override
    public void sound() {
        System.out.println("Dog barks");
    }
}

public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Animal();
        Animal myDog = new Dog();

        myAnimal.sound(); // Animal makes a sound
        myDog.sound();    // Dog barks
    }
}
```

Polimorfizm:

- Dzięki dziedziczeniu, obiekt może być traktowany jako instancja klasy bazowej lub podklasy.

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

class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Meow");
    }
}

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

public class Main {
    public static void main(String[] args) {
        Animal myAnimal = new Animal();
        Animal myCat = new Cat();
        Animal myDog = new Dog();

        myAnimal.makeSound(); // Some generic animal sound
        myCat.makeSound();    // Meow
        myDog.makeSound();    // Woof
    }
}
```

4. Interfejsy

Opis:

- Interfejsy definiują zestaw metod, które klasa musi zaimplementować, ale nie zawierają implementacji tych metod (z wyjątkiem metod domyślnych i statycznych od Javy 8).

Deklaracja Interfejsu:

```
interface Animal {
    void animalSound();
    void sleep();
}
```

Implementacja Interfejsu:

- Klasa implementująca interfejs musi zaimplementować wszystkie jego metody.

```
class Pig implements Animal {
    public void animalSound() {
        System.out.println("The pig says: wee wee");
    }

    public void sleep() {
        System.out.println("Zzz");
    }
}

public class Main {
    public static void main(String[] args) {
        Pig myPig = new Pig();
        myPig.animalSound(); // The pig says: wee wee
        myPig.sleep();       // Zzz
    }
}
```

Dodatkowy Przykład z Wieloma Interfejsami:

```
interface Drivable {
    void drive();
}

interface Flyable {
    void fly();
}
```

```
class FlyingCar implements Drivable, Flyable {
    public void drive() {
        System.out.println("Driving on the road!");
    }

    public void fly() {
        System.out.println("Flying in the sky!");
    }
}

public class Main {
    public static void main(String[] args) {
        FlyingCar fc = new FlyingCar();
        fc.drive(); // Driving on the road!
        fc.fly();   // Flying in the sky!
    }
}
```

Metody Domyślne i Statyczne (od Java 8):

- **Metody Domyślne (`default`):** Pozwalają na dodawanie metod z implementacją w interfejsie.

```
interface Animal {
    void animalSound();
    void sleep();

    default void breathe() {
        System.out.println("Breathing...");
    }
}

class Dog implements Animal {
    public void animalSound() {
        System.out.println("The dog says: woof woof");
    }

    public void sleep() {
        System.out.println("Zzz");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.animalSound(); // The dog says: woof woof
        myDog.sleep();       // Zzz
        myDog.breathe();     // Breathing...
    }
}
```

- **Metody Statyczne:**

```
interface MathOperations {
    static int add(int a, int b) {
        return a + b;
    }
}

public class Main {
    public static void main(String[] args) {
        int sum = MathOperations.add(5, 3);
        System.out.println("Sum: " + sum); // Sum: 8
    }
}
```

Różnice między Klasą a Interfejsem:

Klasa Abstrakcyjna	Interfejs
Może zawierać zarówno metody abstrakcyjne, jak i metody z implementacją.	W Java 8+ mogą mieć metody domyślne i statyczne, wcześniej tylko metody abstrakcyjne.
Może mieć pola (zmienne instancji).	Może mieć tylko stałe (`public static final`).
Może mieć konstruktory.	Nie może mieć konstruktorów.
Klasa może dziedziczyć tylko jedną klasę abstrakcyjną.	Klasa może implementować wiele interfejsów.
Może posiadać modyfikatory dostępu (public, private, etc.).	Wszystkie metody są domyślnie `public`.
Używana, gdy chcemy zdefiniować wspólne zachowania i właściwości dla grupy klas.	Używana, gdy chcemy zapewnić, że klasa implementuje określone metody.

5. Klasy Abstrakcyjne

Opis:

- Klasa abstrakcyjna to klasa, która nie może być instancjonowana bezpośrednio.
- Może zawierać zarówno metody abstrakcyjne (bez implementacji), jak i metody z implementacją.

Deklaracja Klasy Abstrakcyjnej:

```
abstract class Animal {
    public abstract void animalSound(); // Metoda abstrakcyjna

    public void sleep() { // Metoda z implementacją
        System.out.println("Zzz");
    }
}
```

Implementacja Klasy Abstrakcyjnej:

- Klasa dziedzicząca musi zaimplementować wszystkie metody abstrakcyjne lub sama być abstrakcyjna.

```
class Dog extends Animal {
    public void animalSound() {
        System.out.println("The dog says: woof woof");
    }
}

public class Main {
    public static void main(String[] args) {
        Dog myDog = new Dog();
        myDog.animalSound(); // The dog says: woof woof
        myDog.sleep();       // Zzz
    }
}
```

Dodatkowy Przykład z Większą Liczbą Metod:

```
abstract class Shape {
    String color;

    // Konstruktor
    Shape(String color) {
        this.color = color;
    }

    // Metoda abstrakcyjna
    abstract double area();

    // Metoda nieabstrakcyjna
    public void displayColor() {
        System.out.println("Color: " + color);
    }
}

class Circle extends Shape {
    double radius;

    Circle(String color, double radius) {
        super(color);
        this.radius = radius;
    }

    @Override
    double area() {
        return Math.PI * radius * radius;
    }
```

```
}

public class Main {
    public static void main(String[] args) {
        Circle circle = new Circle("Red", 5.0);
        circle.displayColor(); // Color: Red
        System.out.println("Area: " + circle.area()); // Area: 78.53981633974483
    }
}
```

Uwagi:

- **Abstrakcyjne Metody:** Nie mają implementacji i muszą być zaimplementowane przez podklasy.
- **Abstrakcyjne Klasy:** Mogą zawierać zarówno abstrakcyjne, jak i nieabstrakcyjne metody.

6. Różnice między Klasą Abstrakcyjną a Interfejsem

Tabela Porównawcza:

Cecha	Klasa Abstrakcyjna	Interfejs
Metody	Może zawierać zarówno metody abstrakcyjne, jak i z implementacją.	Od Java 8 mogą mieć metody domyślne i statyczne; wcześniej tylko abstrakcyjne.
Pola	Może mieć zmienne instancji.	Może mieć tylko stałe (`public static final`).
Konstruktory	Może mieć konstruktory.	Nie może mieć konstruktorów.
Dziedziczenie	Klasa może dziedziczyć tylko jedną klasę abstrakcyjną.	Klasa może implementować wiele interfejsów.
Dostęp	Może mieć różne modyfikatory dostępu (public, protected, etc.).	Wszystkie metody są domyślnie `public`.
Przeznaczenie	Używana do wspólnej implementacji dla grupy klas.	Używana do definiowania kontraktów, które klasy muszą spełniać.

Przykład Użycia:

- **Klasa Abstrakcyjna:**

```
abstract class Animal {
    public abstract void makeSound();

    public void sleep() {
        System.out.println("Zzz");
    }
}

class Cat extends Animal {
    @Override
```

```
    public void makeSound() {
        System.out.println("Meow");
    }
}

public class Main {
    public static void main(String[] args) {
        Cat myCat = new Cat();
        myCat.makeSound(); // Meow
        myCat.sleep();     // Zzz
    }
}
```

- **Interfejs:**

```
interface Drivable {
    void drive();
}

interface Flyable {
    void fly();
}

class FlyingCar implements Drivable, Flyable {
    @Override
    public void drive() {
        System.out.println("Driving on the road!");
    }

    @Override
    public void fly() {
        System.out.println("Flying in the sky!");
    }
}

public class Main {
    public static void main(String[] args) {
        FlyingCar fc = new FlyingCar();
        fc.drive(); // Driving on the road!
        fc.fly();   // Flying in the sky!
    }
}
```

Kiedy Używać Klas Abstrakcyjnych vs Interfejsów?

- **Klasy Abstrakcyjne:**
  - Gdy chcesz udostępnić wspólną implementację dla grupy klas.
  - Gdy istnieje silny związek między klasą abstrakcyjną a jej podklasami.
- **Interfejsy:**

- Gdy chcesz zdefiniować kontrakt, który mogą implementować różne, niezwiązane klasy.
- Gdy potrzebujesz wielokrotnego dziedziczenia zachowań.

7. Klasa Anonimowa

Opis:

- Klasa anonimowa to klasa bez nazwy, definiowana i instancjonowana jednocześnie.
- Używana głównie do tworzenia instancji interfejsów lub klas abstrakcyjnych w miejscu ich użycia.

Cechy:

- Nie ma nazwy klasy.
- Może dziedziczyć tylko jedną klasę lub implementować jeden interfejs.
- Może zawierać zmienne, metody i kod inicjalizacyjny.
- Nie może mieć konstruktorów, ale może korzystać z konstruktorów klasy nadrzędnej.

Typowy Przykład Użycia:

- **Implementacja Interfejsu `Runnable`:**

```
Runnable runnable = new Runnable() {
    @Override
    public void run() {
        System.out.println("Anonimowa klasa");
    }
};

runnable.run(); // Anonimowa klasa
```

Dodatkowy Przykład z Event Listener:

- **Przykład w Kontekście GUI:**

```
import javax.swing.*;
import java.awt.event.*;

public class ButtonExample {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Button Example");
        JButton button = new JButton("Click Me");

        button.addActionListener(new ActionListener() {
            @Override
            public void actionPerformed(ActionEvent e) {
                System.out.println("Button clicked!");
            }
        });

        frame.add(button);
```

```
            frame.setSize(300, 200);
            frame.setLayout(null);
            button.setBounds(100, 80, 100, 30);
            frame.setVisible(true);
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        }
    }
```

- **Opis:** Tutaj klasa anonimowa implementuje interfejs `ActionListener` i definiuje metodę `actionPerformed` bez tworzenia osobnej klasy.

Dodatkowy Przykład z Klasy Abstrakcyjnej:

```
abstract class Greeting {
    abstract void sayHello();
}

public class Main {
    public static void main(String[] args) {
        Greeting greet = new Greeting() {
            @Override
            void sayHello() {
                System.out.println("Hello from anonimowa klasa!");
            }
        };

        greet.sayHello(); // Hello from anonimowa klasa!
    }
}
```

Użycie z Lambda (od Java 8):

- Niektóre interfejsy (tzw. interfejsy funkcyjne) mogą być implementowane za pomocą wyrażeń lambda, co zastępuje klasę anonimową.

```
Runnable runnable = () -> System.out.println("Runnable z lambda");
runnable.run(); // Runnable z lambda
```

Uwagi:

- Klasy anonimowe są przydatne, gdy potrzebujemy szybkiej, jednorazowej implementacji interfejsu lub klasy abstrakcyjnej bez potrzeby tworzenia osobnej, nazwanej klasy.
- Z powodu braku nazwy, klasy anonimowe są trudniejsze do debugowania i ponownego użycia.

Wyjątki

1. Obsługa Wyjątków

Opis:

- Wyjątki są zdarzeniami, które zakłócają normalny przepływ programu. Mogą być spowodowane przez błędy w kodzie, problemy z zasobami, itp.
- Obsługa wyjątków pozwala na kontrolowane zarządzanie błędami, zapobiegając awariom programu.

Podstawowe Bloki Obsługi Wyjątków:

1. `try`
   - Blok kodu, w którym mogą wystąpić wyjątki.
2. `catch`
   - Blok kodu, który obsługuje wyjątek, gdy się pojawi.
3. `finally`
   - Blok kodu, który zawsze się wykonuje, niezależnie od tego, czy wyjątek wystąpił, czy nie.

Przykład Podstawowy:

```
public class Main {
    public static void main(String[] args) {
        try {
            int[] myNumbers = {1, 2, 3};
            System.out.println(myNumbers[10]); // Błąd: indeks poza zakresem
        } catch (Exception e) {
            System.out.println("Coś poszło nie tak.");
        } finally {
            System.out.println("Koniec");
        }
    }
}
```

Wynik:

```
Coś poszło nie tak.
Koniec
```

Wyjaśnienie:

- `try`**:** Próbujemy uzyskać dostęp do elementu tablicy o indeksie 10, co jest poza zakresem.
- `catch`**:** Przechwytujemy wszelkie wyjątki typu `Exception` i wyświetlamy komunikat.
- `finally`**:** Niezależnie od tego, czy wyjątek wystąpił, czy nie, wyświetlamy "Koniec".

Dodatkowe Przykłady:

1. **Różne Rodzaje Wyjątków:**
   - `ArithmeticException`**:** Błąd arytmetyczny, np. dzielenie przez zero.

```
public class Main {
    public static void main(String[] args) {
        try {
            int a = 10 / 0; // Błąd: dzielenie przez zero
        } catch (ArithmeticException e) {
            System.out.println("Nie można dzielić przez zero!");
        }
    }
}
```

Wynik:

```
Nie można dzielić przez zero!
```

- `NumberFormatException`: Błąd formatowania liczby.

```
public class Main {
    public static void main(String[] args) {
        try {
            int number = Integer.parseInt("ABC"); // Błąd: nie można
skonwertować
        } catch (NumberFormatException e) {
            System.out.println("Nieprawidłowy format liczby.");
        }
    }
}
```

Wynik:

```
Nieprawidłowy format liczby.
```

2. **Wielokrotne Bloki `catch`:**

```
public class Main {
    public static void main(String[] args) {
        try {
            String text = null;
            System.out.println(text.length()); // Błąd: NullPointerException
        } catch (ArithmeticException e) {
            System.out.println("Arithmetic error.");
        } catch (NullPointerException e) {
            System.out.println("Null reference!");
        } catch (Exception e) {
            System.out.println("Ogólny błąd.");
```

```
        }
    }
}
```

Wynik:

```
Null reference!
```

3. **Rzucanie Wyjątków (`throw`):**
    - Możemy ręcznie zgłaszać wyjątki za pomocą słowa kluczowego `throw`.

```
public class Main {
    public static void main(String[] args) {
        try {
            checkAge(15);
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }

    static void checkAge(int age) throws Exception {
        if (age < 18) {
            throw new Exception("Osoba niepełnoletnia!");
        }
        System.out.println("Osoba pełnoletnia.");
    }
}
```

Wynik:

```
Osoba niepełnoletnia!
```

4. **Rzucanie Wyjątków Złaszczańców:**
    - Tworzenie własnych klas wyjątków.

```
class MyException extends Exception {
    public MyException(String message) {
        super(message);
    }
}

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

```
        try {
            validate(13);
        } catch (MyException e) {
            System.out.println("Caught exception: " + e.getMessage());
        }
    }

    static void validate(int age) throws MyException {
        if (age < 18) {
            throw new MyException("Nie spełniasz wymagań wiekowych!");
        }
        System.out.println("Weryfikacja powiodła się.");
    }
}
```

Wynik:

```
Caught exception: Nie spełniasz wymagań wiekowych!
```

Uwagi:

- `try-catch-finally`**:** Umożliwia kontrolowane zarządzanie błędami.
- **Hierarchia Wyjątków:** `Exception` jest nadrzędnym typem dla większości wyjątków. `RuntimeException` to podtyp, który nie wymaga deklaracji w sygnaturze metod (np. `NullPointerException`).
- **Obsługa Specyficznych Wyjątków:** Najpierw umieszczaj bardziej specyficzne wyjątki przed bardziej ogólnymi, aby uniknąć ich przechwytywania przez ogólny `catch`.

Wątki (Threads)

1. Dziedziczenie po `Thread`

Opis:

- Wątek to jednostka wykonania w programie, która pozwala na równoczesne wykonywanie kodu.
- Można tworzyć wątki poprzez dziedziczenie po klasie `Thread` i nadpisanie metody `run`.

Przykład:

```
public class MyThread extends Thread {
    public void run() {
        System.out.println("Kod w wątku");
    }

    public static void main(String[] args) {
        MyThread t = new MyThread();
        t.start(); // Uruchamia wątek, wywołuje metodę run()
```

```
    }
}
```

Wynik:

```
Kod w wątku
```

Dodatkowy Przykład z Równoczesnym Wykonaniem:

```
class Task extends Thread {
    private String name;

    Task(String name) {
        this.name = name;
    }

    public void run() {
        for(int i = 1; i <= 5; i++) {
            System.out.println(name + " - Iteracja: " + i);
            try {
                Thread.sleep(500); // Pausa 500ms
            } catch (InterruptedException e) {
                System.out.println(name + " został przerwany.");
            }
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Task t1 = new Task("Wątek 1");
        Task t2 = new Task("Wątek 2");

        t1.start();
        t2.start();
    }
}
```

Przykładowy Wynik:

```
Wątek 1 - Iteracja: 1
Wątek 2 - Iteracja: 1
Wątek 1 - Iteracja: 2
Wątek 2 - Iteracja: 2
...
```

Uwagi:

- **Metoda `start()`:** Uruchamia nowy wątek i wywołuje metodę `run()`.
- **Metoda `run()`:** Zawiera kod, który ma być wykonany w nowym wątku.
- **`Thread.sleep(milliseconds)`:** Spowalnia działanie wątku na określoną liczbę milisekund.

2. Implementacja `Runnable`

Opis:

- Alternatywny sposób tworzenia wątków bez dziedziczenia po klasie `Thread`.
- Polega na implementacji interfejsu `Runnable` i przekazaniu instancji do obiektu `Thread`.

Przykład:

```
public class MyRunnable implements Runnable {
    public void run() {
        System.out.println("Kod w wątku Runnable");
    }

    public static void main(String[] args) {
        MyRunnable runnable = new MyRunnable();
        Thread t = new Thread(runnable);
        t.start(); // Uruchamia wątek, wywołuje metodę run()
    }
}
```

Wynik:

```
Kod w wątku Runnable
```

Dodatkowy Przykład z Wieloma Wątkami:

```
class Worker implements Runnable {
    private String name;

    Worker(String name) {
        this.name = name;
    }

    public void run() {
        for(int i = 1; i <= 5; i++) {
            System.out.println(name + " - Praca: " + i);
            try {
                Thread.sleep(300); // Pausa 300ms
            } catch (InterruptedException e) {
                System.out.println(name + " został przerwany.");
            }
```

```
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Thread t1 = new Thread(new Worker("Pracownik 1"));
        Thread t2 = new Thread(new Worker("Pracownik 2"));

        t1.start();
        t2.start();
    }
}
```

Przykładowy Wynik:

```
Pracownik 1 - Praca: 1
Pracownik 2 - Praca: 1
Pracownik 1 - Praca: 2
Pracownik 2 - Praca: 2
...
```

Porównanie `Thread` vs `Runnable`:

Cecha	Thread (Dziedziczenie)	Runnable (Interfejs)
Dziedziczenie	Musisz dziedziczyć po klasie `Thread`.	Możesz implementować interfejs `Runnable` bez dziedziczenia.
Wielokrotne Dziedziczenie	Java nie obsługuje wielokrotnego dziedziczenia klas.	Możesz implementować wiele interfejsów.
Reużywalność	Klasa dziedzicząca po `Thread` jest mniej elastyczna.	`Runnable` może być używany z różnymi obiektami `Thread`.
Separacja Zadań od Mechanizmu Wątków	Łączy zadanie z mechanizmem wątku.	Oddziela zadanie od mechanizmu wątku, co jest bardziej elastyczne.

Kiedy Używać Którego Sposobu?

- **`Runnable`:** Gdy klasa już dziedziczy po innej klasie lub gdy chcesz oddzielić logikę zadania od mechanizmu wątków.
- **`Thread`:** Gdy nie masz potrzeby dziedziczenia po innej klasie i chcesz szybko utworzyć prosty wątek.

Wyrażenia Lambda

1. Czym jest Lambda?

Opis:

- Wyrażenie lambda to sposób pisania krótkich, anonimowych funkcji (metod) w Javie.
- Ułatwia pisanie kodu, zwłaszcza przy pracy z interfejsami funkcyjnymi (interfejsy z jedną metodą abstrakcyjną).

Cechy Lambda:

- **Bez nazwy:** Nie muszą być nazwane jak tradycyjne metody.
- **Traktowane jako obiekty:** Można je przypisać do zmiennych lub przekazać jako argumenty.
- **Zwięzłość:** Pozwalają na pisanie mniej kodu i zwiększają czytelność.

Składnia:

```
(parameters) -> { // kod }
```

Przykład:

```
(parameter1, parameter2) -> {
    // Blok kodu
}
```

Przykład Użycia:

- **Interfejs `MathOperation`:**

  ```
  interface MathOperation {
      int operation(int a, int b);
  }
  ```

- **Implementacja bez Lambda:**

  ```
  public class Main {
      public static void main(String[] args) {
          MathOperation addition = new MathOperation() {
              @Override
              public int operation(int a, int b) {
                  return a + b;
              }
          };
          System.out.println(addition.operation(5, 3)); // 8
      }
  }
  ```

- **Implementacja z Lambda:**

```
public class Main {
    public static void main(String[] args) {
        MathOperation addition = (a, b) -> a + b;
        System.out.println(addition.operation(5, 3)); // 8
    }
}
```

Dodatkowe Przykłady:

1. **Sortowanie Listy Stringów z Lambda:**

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

public class Main {
    public static void main(String[] args) {
        ArrayList<String> names = new ArrayList<>();
        names.add("Alice");
        names.add("Bob");
        names.add("Charlie");

        // Sortowanie alfabetyczne
        Collections.sort(names, (a, b) -> a.compareTo(b));

        for(String name : names) {
            System.out.println(name);
        }
        // Wynik:
        // Alice
        // Bob
        // Charlie
    }
}
```

2. **Filtracja Elementów z Lambda:**

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

public class Main {
    public static void main(String[] args) {
        List<Integer> numbers = new ArrayList<>();
        for(int i = 1; i <= 10; i++) {
            numbers.add(i);
        }

        // Filtracja liczb parzystych
```

```
        List<Integer> evenNumbers = numbers.stream()
                                           .filter(n -> n % 2 == 0)
                                           .collect(Collectors.toList());

        System.out.println(evenNumbers); // [2, 4, 6, 8, 10]
    }
}
```

3. **Interfejs Funkcyjny `Comparator` z Lambda:**

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.Comparator;

public class Main {
    public static void main(String[] args) {
        ArrayList<String> fruits = new ArrayList<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Cherry");

        // Sortowanie malejąco
        Collections.sort(fruits, (a, b) -> b.compareTo(a));

        for(String fruit : fruits) {
            System.out.println(fruit);
        }
        // Wynik:
        // Cherry
        // Banana
        // Apple
    }
}
```

Uwagi:

- Lambda ułatwiają pracę z interfejsami funkcyjnymi, eliminując potrzebę tworzenia osobnych klas anonimowych.
- Są szczególnie przydatne w połączeniu z Stream API do przetwarzania kolekcji.

Programowanie Generyczne

1. Typ Wieloznaczny (`?`)

Opis:

- Symbol `?` w programowaniu generycznym oznacza dowolny typ.
- Używany, gdy nie znamy konkretnego typu, ale chcemy zachować ogólność kodu.

Przykład:

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

public class Main {
    public static void printList(List<?> list) {
        for(Object elem : list) {
            System.out.println(elem);
        }
    }

    public static void main(String[] args) {
        List<String> stringList = new ArrayList<>();
        stringList.add("Apple");
        stringList.add("Banana");
        printList(stringList); // Apple, Banana

        List<Integer> intList = new ArrayList<>();
        intList.add(1);
        intList.add(2);
        printList(intList); // 1, 2
    }
}
```

Dodatkowe Przykłady:

1. **Metoda Akceptująca Listę Dowolnego Typu:**

```
public static void displayList(List<?> list) {
    for(Object item : list) {
        System.out.println(item);
    }
}

public static void main(String[] args) {
    List<Double> doubleList = new ArrayList<>();
    doubleList.add(1.1);
    doubleList.add(2.2);
    displayList(doubleList); // 1.1, 2.2
}
```

2. **Metoda Zwracająca Dowolny Typ:**

```
public static <T> T getFirstElement(List<T> list) {
    if(list.isEmpty()) {
        return null;
    }
```

```
    return list.get(0);
}

public static void main(String[] args) {
    List<String> names = new ArrayList<>();
    names.add("Alice");
    names.add("Bob");
    String firstName = getFirstElement(names);
    System.out.println(firstName); // Alice
}
```

3. **Użycie Typów Wieloznacznych z Ograniczeniami (`extends` i `super`):**

   - **Górne Ograniczenia (`extends`):** Pozwalają na używanie typów, które są podtypami określonej klasy.

     ```
     public static <T extends Number> void processNumbers(List<T> numbers) {
         for(T num : numbers) {
             System.out.println(num.doubleValue());
         }
     }

     public static void main(String[] args) {
         List<Integer> ints = new ArrayList<>();
         ints.add(1);
         ints.add(2);
         processNumbers(ints); // 1.0, 2.0
     }
     ```

   - **Dolne Ograniczenia (`super`):** Pozwalają na używanie typów, które są nadtypami określonej klasy.

     ```
     public static void addNumbers(List<? super Integer> list) {
         list.add(10);
         list.add(20);
     }

     public static void main(String[] args) {
         List<Number> numbers = new ArrayList<>();
         addNumbers(numbers);
         System.out.println(numbers); // [10, 20]
     }
     ```

Uwagi:

- **Typy Generyczne:** Umożliwiają pisanie bardziej elastycznego i wielokrotnego użycia kodu.
- **Bezpieczeństwo Typów:** Generyki zapewniają, że typy są poprawnie używane podczas kompilacji, co zmniejsza ryzyko błędów w czasie wykonywania.

- **Unikanie Rzutowania:** Dzięki generykom, często nie trzeba ręcznie rzutować typów, co upraszcza kod.

Operatory

1. Porównywanie Obiektów

Opis:

- W Javie istnieją różne sposoby porównywania obiektów. Najczęściej używane to operator `==` oraz metoda `.equals()`.

Operator `==`:

- **Dla Typów Prymitywnych:** Sprawdza, czy wartości są takie same.

```
int a = 5;
int b = 5;
System.out.println(a == b); // true
```

- **Dla Typów Referencyjnych:** Sprawdza, czy dwie referencje wskazują na ten sam obiekt w pamięci.

```
String s1 = new String("Java");
String s2 = new String("Java");
System.out.println(s1 == s2); // false
```

Metoda `.equals()`:

- **Opis:** Sprawdza, czy zawartość obiektów jest taka sama. Może być nadpisana przez klasy, aby dostosować sposób porównywania.

```
String s1 = new String("Java");
String s2 = new String("Java");
System.out.println(s1.equals(s2)); // true
```

Różnice między `==` a `.equals()`:

- `==`**:** Porównuje referencje (adresy w pamięci) obiektów.
- `.equals()`**:** Porównuje zawartość obiektów, jeśli jest nadpisana.

Dodatkowe Przykłady:

1. **Porównywanie Stringów:**

```
String a = "Hello";
String b = "Hello";
```

```
String c = new String("Hello");

System.out.println(a == b); // true (ze względu na internowanie Stringów)
System.out.println(a == c); // false
System.out.println(a.equals(c)); // true
```

2. **Porównywanie Obiektów Własnych Klas:**

```
class Person {
    String name;

    Person(String name) {
        this.name = name;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || getClass() != obj.getClass()) return false;
        Person person = (Person) obj;
        return name.equals(person.name);
    }
}

public class Main {
    public static void main(String[] args) {
        Person p1 = new Person("Alice");
        Person p2 = new Person("Alice");
        Person p3 = p1;

        System.out.println(p1 == p2); // false
        System.out.println(p1 == p3); // true
        System.out.println(p1.equals(p2)); // true
    }
}
```

3. **Porównywanie Liczb:**

```
Integer num1 = 100;
Integer num2 = 100;
Integer num3 = 200;
Integer num4 = 200;

System.out.println(num1 == num2); // true (dla wartości od -128 do 127 są
internowane)
System.out.println(num3 == num4); // false
System.out.println(num3.equals(num4)); // true
```

Uwagi:

- **Internowanie Stringów:** Java automatycznie internuje Stringi, co oznacza, że literały Stringów są przechowywane w puli Stringów, co pozwala na ich ponowne użycie i umożliwia porównanie za pomocą `==`.
- **Nadpisywanie `.equals()`:** Aby poprawnie porównać obiekty własnych klas, należy nadpisać metodę `.equals()` oraz `hashCode()`.

Kolekcje w Javie

1. Java Collections Framework (JCF)

Opis:

- JCF to zestaw interfejsów, klas i algorytmów, które ułatwiają przechowywanie, manipulowanie i dostęp do danych w formie kolekcji.

Podstawowe Interfejsy:

1. **Collection**
    - **Opis:** Podstawowy interfejs dla większości kolekcji.
    - **Podinterfejsy:**
        - **List:** Kolekcja z elementami w określonej kolejności, umożliwiająca duplikaty. Przykłady: `ArrayList`, `LinkedList`.
        - **Set:** Kolekcja bez duplikatów, niekoniecznie w określonej kolejności. Przykłady: `HashSet`, `TreeSet`.
        - **Queue:** Kolekcja oparta na zasadzie FIFO (First-In-First-Out). Przykład: `PriorityQueue`.
        - **Deque:** Kolekcja działająca jako kolejka dwukierunkowa, umożliwiająca operacje zarówno na początku, jak i na końcu. Przykład: `ArrayDeque`.
2. **Map**
    - **Opis:** Struktura danych przechowująca pary klucz-wartość.
    - **Podinterfejsy:**
        - **SortedMap:** Mapa z kluczami w uporządkowanej kolejności.
        - **NavigableMap:** Rozszerza `SortedMap`, umożliwiając nawigację po kluczach.
3. **Other Interfaces:**
    - **Iterator:** Obiekt do iterowania przez elementy kolekcji.
    - **ListIterator:** Rozszerzenie `Iterator` dla list, umożliwiające iterację w obu kierunkach.
    - **Comparable:** Interfejs do definiowania naturalnego porządku obiektów.
    - **Comparator:** Umożliwia dostosowanie porządku obiektów.

Dodatkowe Informacje:

- **Generics:** Wszystkie kolekcje mogą używać typów generycznych, co pozwala na przechowywanie określonych typów danych.

2. Przykłady Implementacji Kolekcji

List (Lista)

1. **ArrayList**
   - **Opis:** Lista oparta na dynamicznej tablicy. Szybki dostęp do elementów za pomocą indeksu, ale wolniejsze operacje dodawania/ usuwania elementów w środku listy.
   - **Przykład:**

```
import java.util.ArrayList;

public class Main {
    public static void main(String[] args) {
        ArrayList<String> fruits = new ArrayList<>();
        fruits.add("Apple");
        fruits.add("Banana");
        fruits.add("Cherry");
        System.out.println(fruits); // [Apple, Banana, Cherry]

        fruits.remove(1);
        System.out.println(fruits); // [Apple, Cherry]

        fruits.set(1, "Blueberry");
        System.out.println(fruits); // [Apple, Blueberry]
    }
}
```

2. **LinkedList**
   - **Opis:** Lista dwukierunkowa (elementy są połączone węzłami). Szybkie dodawanie/ usuwanie elementów w środku listy, ale wolniejszy dostęp do elementów za pomocą indeksu.
   - **Przykład:**

```
import java.util.LinkedList;

public class Main {
    public static void main(String[] args) {
        LinkedList<String> animals = new LinkedList<>();
        animals.add("Cat");
        animals.add("Dog");
        animals.add("Elephant");
        System.out.println(animals); // [Cat, Dog, Elephant]

        animals.addFirst("Lion");
        animals.addLast("Tiger");
        System.out.println(animals); // [Lion, Cat, Dog, Elephant,
Tiger]

        animals.removeFirst();
```

```
        animals.removeLast();
        System.out.println(animals); // [Cat, Dog, Elephant]
    }
}
```

3. **Vector**

   - **Opis:** Stara implementacja listy, która jest synchronizowana (bezpieczna dla wielowątkowości). Została w dużej mierze zastąpiona przez `ArrayList`.
   - **Przykład:**

```
import java.util.Vector;

public class Main {
    public static void main(String[] args) {
        Vector<String> stack = new Vector<>();
        stack.add("First");
        stack.add("Second");
        stack.add("Third");
        System.out.println(stack); // [First, Second, Third]

        stack.remove(1);
        System.out.println(stack); // [First, Third]
    }
}
```

4. **Stack**

   - **Opis:** Specjalizacja klasy `Vector`, działająca jako stos (LIFO - Last-In-First-Out).
   - **Przykład:**

```
import java.util.Stack;

public class Main {
    public static void main(String[] args) {
        Stack<String> stack = new Stack<>();
        stack.push("A");
        stack.push("B");
        stack.push("C");
        System.out.println(stack); // [A, B, C]

        String top = stack.pop();
        System.out.println(top); // C
        System.out.println(stack); // [A, B]

        String peek = stack.peek();
        System.out.println(peek); // B
    }
}
```

Set (Zbiór)

1. **HashSet**
   - **Opis:** Nieuporządkowany zbiór bez duplikatów. Umożliwia szybkie dodawanie, usuwanie i sprawdzanie zawartości.
   - **Przykład:**

```
import java.util.HashSet;

public class Main {
    public static void main(String[] args) {
        HashSet<String> uniqueNames = new HashSet<>();
        uniqueNames.add("Alice");
        uniqueNames.add("Bob");
        uniqueNames.add("Alice"); // Duplikat, nie zostanie dodany
        System.out.println(uniqueNames); // [Alice, Bob]
    }
}
```

2. **LinkedHashSet**
   - **Opis:** `HashSet` z zachowaniem kolejności dodawania elementów.
   - **Przykład:**

```
import java.util.LinkedHashSet;

public class Main {
    public static void main(String[] args) {
        LinkedHashSet<String> orderedSet = new LinkedHashSet<>();
        orderedSet.add("Red");
        orderedSet.add("Green");
        orderedSet.add("Blue");
        System.out.println(orderedSet); // [Red, Green, Blue]
    }
}
```

3. **TreeSet**
   - **Opis:** Zbiór oparty na drzewie, utrzymujący elementy w porządku naturalnym lub zdefiniowanym przez `Comparator`.
   - **Przykład:**

```
import java.util.TreeSet;

public class Main {
```

```
    public static void main(String[] args) {
        TreeSet<Integer> sortedNumbers = new TreeSet<>();
        sortedNumbers.add(50);
        sortedNumbers.add(20);
        sortedNumbers.add(40);
        sortedNumbers.add(10);
        System.out.println(sortedNumbers); // [10, 20, 40, 50]
    }
}
```

Queue (Kolejka)

1. **PriorityQueue**
   - **Opis:** Kolejka z priorytetami, gdzie elementy są sortowane według ich naturalnego porządku lub `Comparator`.
   - **Przykład:**

```
import java.util.PriorityQueue;

public class Main {
    public static void main(String[] args) {
        PriorityQueue<Integer> pq = new PriorityQueue<>();
        pq.add(40);
        pq.add(10);
        pq.add(30);
        pq.add(20);
        System.out.println(pq); // [10, 20, 30, 40]

        while(!pq.isEmpty()) {
            System.out.println(pq.poll()); // 10, 20, 30, 40
        }
    }
}
```

2. **LinkedList jako FIFO Queue**
   - **Opis:** `LinkedList` może działać jako kolejka FIFO.
   - **Przykład:**

```
import java.util.LinkedList;
import java.util.Queue;

public class Main {
    public static void main(String[] args) {
        Queue<String> queue = new LinkedList<>();
        queue.add("First");
        queue.add("Second");
        queue.add("Third");
```

```
        System.out.println(queue); // [First, Second, Third]

        String removed = queue.poll();
        System.out.println("Usunięto: " + removed); // Usunięto: First
        System.out.println(queue); // [Second, Third]
    }
}
```

Deque (Kolejka Dwukierunkowa)

1. **ArrayDeque**
   - **Opis:** Tablicowa implementacja `Deque`. Szybka i nie ma ograniczeń dotyczących rozmiaru.
   - **Przykład:**

```
import java.util.ArrayDeque;
import java.util.Deque;

public class Main {
    public static void main(String[] args) {
        Deque<String> deque = new ArrayDeque<>();
        deque.addFirst("Start");
        deque.addLast("End");
        deque.addFirst("Begin");

        System.out.println(deque); // [Begin, Start, End]

        deque.removeLast();
        System.out.println(deque); // [Begin, Start]
    }
}
```

2. **LinkedList jako Deque**
   - **Opis:** `LinkedList` może działać jako `Deque`.
   - **Przykład:**

```
import java.util.LinkedList;
import java.util.Deque;

public class Main {
    public static void main(String[] args) {
        Deque<Integer> deque = new LinkedList<>();
        deque.addLast(1);
        deque.addLast(2);
        deque.addFirst(0);

        System.out.println(deque); // [0, 1, 2]
```

```
        int first = deque.removeFirst();
        System.out.println("Usunięto pierwszy: " + first); // Usunięto
pierwszy: 0
        System.out.println(deque); // [1, 2]
    }
}
```

Map (Mapa)

1. **HashMap**
   - **Opis:** Mapa przechowująca pary klucz-wartość bez gwarancji kolejności.
   - **Przykład:**

```
import java.util.HashMap;

public class Main {
    public static void main(String[] args) {
        HashMap<String, Integer> map = new HashMap<>();
        map.put("One", 1);
        map.put("Two", 2);
        map.put("Three", 3);
        System.out.println(map); // {One=1, Two=2, Three=3}

        System.out.println(map.get("Two")); // 2

        map.remove("One");
        System.out.println(map); // {Two=2, Three=3}
    }
}
```

2. **LinkedHashMap**
   - **Opis:** `HashMap` z zachowaniem kolejności wstawiania elementów.
   - **Przykład:**

```
import java.util.LinkedHashMap;

public class Main {
    public static void main(String[] args) {
        LinkedHashMap<String, String> capitals = new LinkedHashMap<>();
        capitals.put("Poland", "Warsaw");
        capitals.put("Germany", "Berlin");
        capitals.put("Italy", "Rome");
        System.out.println(capitals); // {Poland=Warsaw,
Germany=Berlin, Italy=Rome}
    }
}
```

3. **TreeMap**
   - **Opis:** Mapa z kluczami uporządkowanymi w naturalnym porządku lub przez `Comparator`.
   - **Przykład:**

```
import java.util.TreeMap;

public class Main {
    public static void main(String[] args) {
        TreeMap<Integer, String> sortedMap = new TreeMap<>();
        sortedMap.put(3, "Three");
        sortedMap.put(1, "One");
        sortedMap.put(2, "Two");
        System.out.println(sortedMap); // {1=One, 2=Two, 3=Three}
    }
}
```

4. **WeakHashMap**
   - **Opis:** Mapa, w której klucze są słabymi referencjami, co pozwala na ich usunięcie przez Garbage Collector, gdy nie są już używane.
   - **Przykład:**

```
import java.util.WeakHashMap;

public class Main {
    public static void main(String[] args) {
        WeakHashMap<String, String> weakMap = new WeakHashMap<>();
        String key = new String("Key");
        weakMap.put(key, "Value");

        System.out.println(weakMap); // {Key=Value}

        key = null; // Ustawiamy referencję na null

        System.gc(); // Ręcznie wywołujemy Garbage Collector

        // Po czasie (może się nie zdarzyć natychmiast)
        System.out.println(weakMap); // {} lub {Key=Value}
    }
}
```

5. **IdentityHashMap**
   - **Opis:** Mapa porównująca klucze na podstawie referencji (`==`), a nie metodą `.equals()`.
   - **Przykład:**

```
import java.util.IdentityHashMap;

public class Main {
    public static void main(String[] args) {
        IdentityHashMap<String, String> identityMap = new
IdentityHashMap<>();
        String a = new String("Test");
        String b = new String("Test");

        identityMap.put(a, "Value A");
        identityMap.put(b, "Value B");

        System.out.println(identityMap); // {Test=Value A, Test=Value
B}
        System.out.println(identityMap.size()); // 2
    }
}
```

6. **Hashtable**

   - **Opis:** Stara, synchronizowana implementacja `Map`, która została w dużej mierze zastąpiona przez `HashMap`.
   - **Przykład:**

```
import java.util.Hashtable;

public class Main {
    public static void main(String[] args) {
        Hashtable<String, Integer> table = new Hashtable<>();
        table.put("One", 1);
        table.put("Two", 2);
        table.put("Three", 3);
        System.out.println(table); // {One=1, Two=2, Three=3}

        System.out.println(table.get("Two")); // 2
    }
}
```

Specjalne Implementacje Kolekcji

1. **EnumSet**

   - **Opis:** Wysoko wydajny zbiór do zarządzania wartościami typu `enum`.
   - **Przykład:**

```
import java.util.EnumSet;

enum Days {
```

```
    MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY, SUNDAY
}

public class Main {
    public static void main(String[] args) {
        EnumSet<Days> weekend = EnumSet.of(Days.SATURDAY, Days.SUNDAY);
        System.out.println(weekend); // [SATURDAY, SUNDAY]
    }
}
```

2. **EnumMap**

   - **Opis:** Mapa z kluczami typu `enum`.
   - **Przykład:**

```
import java.util.EnumMap;

enum Size {
    SMALL, MEDIUM, LARGE
}

public class Main {
    public static void main(String[] args) {
        EnumMap<Size, String> sizeMap = new EnumMap<>(Size.class);
        sizeMap.put(Size.SMALL, "S");
        sizeMap.put(Size.MEDIUM, "M");
        sizeMap.put(Size.LARGE, "L");
        System.out.println(sizeMap); // {SMALL=S, MEDIUM=M, LARGE=L}
    }
}
```

3. **Properties**

   - **Opis:** Klasa dziedzicząca po `Hashtable`, przechowująca pary klucz-wartość w formie tekstu. Używana głównie do przechowywania ustawień konfiguracyjnych.
   - **Przykład:**

```
import java.util.Properties;
import java.io.FileInputStream;
import java.io.IOException;

public class Main {
    public static void main(String[] args) {
        Properties props = new Properties();
        try {
            FileInputStream fis = new
FileInputStream("config.properties");
            props.load(fis);
            fis.close();
```

```
        } catch (IOException e) {
            e.printStackTrace();
        }

        String dbUser = props.getProperty("db.user");
        String dbPass = props.getProperty("db.pass");
        System.out.println("DB User: " + dbUser);
        System.out.println("DB Pass: " + dbPass);
    }
}
```

- **Plik `config.properties`:**

```
db.user=root
db.pass=secret
```

- **Wynik:**

```
DB User: root
DB Pass: secret
```

Iteratory

1. **Iterator**
   - **Opis:** Podstawowy interfejs do iteracji po elementach kolekcji.
   - **Przykład:**

```
import java.util.ArrayList;
import java.util.Iterator;

public class Main {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("A");
        list.add("B");
        list.add("C");

        Iterator<String> itr = list.iterator();
        while (itr.hasNext()) {
            System.out.println(itr.next());
        }
        // Wynik:
        // A
        // B
        // C
```

```
    }
}
```

2. **ListIterator**

   - **Opis:** Rozszerzenie `Iterator` dla list, umożliwiające iterację w obu kierunkach oraz modyfikację listy podczas iteracji.
   - **Przykład:**

```
import java.util.ArrayList;
import java.util.ListIterator;

public class Main {
    public static void main(String[] args) {
        ArrayList<String> list = new ArrayList<>();
        list.add("A");
        list.add("B");
        list.add("C");

        ListIterator<String> litr = list.listIterator();
        while (litr.hasNext()) {
            String elem = litr.next();
            System.out.println(elem);
            if(elem.equals("B")) {
                litr.set("Bee");
            }
        }
        System.out.println(list); // [A, Bee, C]
    }
}
```

3. Klasy Narzędziowe

1. **Collections**

   - **Opis:** Klasa zawierająca metody statyczne do manipulacji kolekcjami, takie jak sortowanie, kopiowanie czy synchronizacja.

   - **Przykład Sortowania Listy:**

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

public class Main {
    public static void main(String[] args) {
        ArrayList<Integer> list = new ArrayList<>();
        list.add(3);
        list.add(1);
        list.add(4);
```

```
        list.add(2);

        Collections.sort(list);
        System.out.println(list); // [1, 2, 3, 4]
    }
}
```

- **Przykład Kopiowania Kolekcji:**

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

public class Main {
    public static void main(String[] args) {
        List<String> source = new ArrayList<>();
        source.add("A");
        source.add("B");
        source.add("C");

        List<String> destination = new ArrayList<>
(Collections.nCopies(source.size(), ""));
        Collections.copy(destination, source);

        System.out.println(destination); // [A, B, C]
    }
}
```

- **Przykład Synchronizacji Kolekcji:**

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

public class Main {
    public static void main(String[] args) {
        List<String> list = new ArrayList<>();
        list.add("A");
        list.add("B");

        List<String> synchronizedList =
Collections.synchronizedList(list);
        synchronizedList.add("C");
        System.out.println(synchronizedList); // [A, B, C]
    }
}
```

2. **Arrays**

- **Opis:** Klasa zawierająca metody do operacji na tablicach, takie jak sortowanie, wyszukiwanie, konwersja na listę itp.
- **Przykład Konwersji Tablicy na Listę:**

```
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        String[] array = {"A", "B", "C"};
        List<String> list = Arrays.asList(array);
        System.out.println(list); // [A, B, C]
    }
}
```

- **Przykład Sortowania Tablicy:**

```
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        int[] numbers = {5, 3, 8, 1, 2};
        Arrays.sort(numbers);
        System.out.println(Arrays.toString(numbers)); // [1, 2, 3, 5,
8]
    }
}
```

- **Przykład Wyszukiwania Elementu w Tablicy:**

```
import java.util.Arrays;

public class Main {
    public static void main(String[] args) {
        int[] numbers = {10, 20, 30, 40, 50};
        int index = Arrays.binarySearch(numbers, 30);
        System.out.println("Index of 30: " + index); // Index of 30: 2
    }
}
```

Uwagi:

- **Collections vs Arrays:** `Collections` oferują bardziej elastyczne i zaawansowane operacje na kolekcjach, podczas gdy `Arrays` są bardziej odpowiednie dla statycznych struktur danych.

4. Synchronizowane Kolekcje

Opis:

- Synchronizowane kolekcje są bezpieczne dla środowisk wielowątkowych. Oznacza to, że można je bezpiecznie modyfikować z wielu wątków jednocześnie.

Typowe Synchronizowane Kolekcje:

1. **CopyOnWriteArrayList**
   - **Opis:** Lista, która tworzy kopię na każdą modyfikację. Idealna dla środowisk, gdzie odczyty są częstsze niż modyfikacje.
   - **Przykład:**

```
import java.util.concurrent.CopyOnWriteArrayList;

public class Main {
    public static void main(String[] args) {
        CopyOnWriteArrayList<String> cowList = new
CopyOnWriteArrayList<>();
        cowList.add("A");
        cowList.add("B");
        cowList.add("C");
        System.out.println(cowList); // [A, B, C]

        cowList.remove("B");
        System.out.println(cowList); // [A, C]
    }
}
```

2. **CopyOnWriteArraySet**
   - **Opis:** Zbiór, który tworzy kopię na każdą modyfikację. Podobnie jak `CopyOnWriteArrayList`, jest idealny dla środowisk z częstymi odczytami.
   - **Przykład:**

```
import java.util.concurrent.CopyOnWriteArraySet;

public class Main {
    public static void main(String[] args) {
        CopyOnWriteArraySet<String> cowSet = new CopyOnWriteArraySet<>
();
        cowSet.add("X");
        cowSet.add("Y");
        cowSet.add("Z");
        System.out.println(cowSet); // [X, Y, Z]

        cowSet.remove("Y");
        System.out.println(cowSet); // [X, Z]
    }
}
```

3. **ConcurrentHashMap**
    - **Opis:** Wysoko wydajna mapa dla środowisk wielowątkowych. Pozwala na równoczesny odczyt i modyfikacje przez wiele wątków.
    - **Przykład:**

```
import java.util.concurrent.ConcurrentHashMap;

public class Main {
    public static void main(String[] args) {
        ConcurrentHashMap<String, Integer> concurrentMap = new
ConcurrentHashMap<>();
        concurrentMap.put("A", 1);
        concurrentMap.put("B", 2);
        concurrentMap.put("C", 3);
        System.out.println(concurrentMap); // {A=1, B=2, C=3}

        concurrentMap.remove("B");
        System.out.println(concurrentMap); // {A=1, C=3}
    }
}
```

Dodatkowy Przykład z Wieloma Wątkami:

```
import java.util.concurrent.ConcurrentHashMap;

class Worker implements Runnable {
    private ConcurrentHashMap<String, Integer> map;
    private String key;
    private int value;

    Worker(ConcurrentHashMap<String, Integer> map, String key, int value) {
        this.map = map;
        this.key = key;
        this.value = value;
    }

    public void run() {
        map.put(key, value);
        System.out.println(Thread.currentThread().getName() + " dodał: " + key +
"=" + value);
    }
}

public class Main {
    public static void main(String[] args) throws InterruptedException {
        ConcurrentHashMap<String, Integer> sharedMap = new ConcurrentHashMap<>();

        Thread t1 = new Thread(new Worker(sharedMap, "A", 1));
```

```
        Thread t2 = new Thread(new Worker(sharedMap, "B", 2));
        Thread t3 = new Thread(new Worker(sharedMap, "C", 3));

        t1.start();
        t2.start();
        t3.start();

        t1.join();
        t2.join();
        t3.join();

        System.out.println(sharedMap); // {A=1, B=2, C=3}
    }
}
```

Uwagi:

- **Synchronizowane Kolekcje vs. Kolekcje Współbieżne:** `Collections.synchronizedList` czy `Collections.synchronizedMap` są prostymi sposobami na synchronizację istniejących kolekcji, ale `CopyOnWriteArrayList`, `CopyOnWriteArraySet` i `ConcurrentHashMap` oferują lepszą wydajność i większą skalowalność w środowiskach wielowątkowych.

Struktury dla Kolejek i Kolekcji Zablokowanych

1. BlockingQueue

Opis:

- `BlockingQueue` to interfejs dla kolejek z mechanizmami blokującymi.
- Operacje wstawiania i usuwania mogą być blokowane, gdy kolejka jest pełna lub pusta.

Podinterfejsy i Implementacje:

1. **ArrayBlockingQueue**
    - **Opis:** Kolejka z ustaloną pojemnością.
    - **Przykład:**

```
import java.util.concurrent.ArrayBlockingQueue;
import java.util.concurrent.BlockingQueue;

public class Main {
    public static void main(String[] args) throws InterruptedException
{
        BlockingQueue<String> queue = new ArrayBlockingQueue<>(2);
        queue.put("Element1");
        queue.put("Element2");
        // queue.put("Element3"); // Zablokuje się, ponieważ kolejka
jest pełna
```

```
        System.out.println(queue.take()); // Element1
        System.out.println(queue.take()); // Element2
        // System.out.println(queue.take()); // Zablokuje się, ponieważ
kolejka jest pusta
    }
}
```

2. **LinkedBlockingQueue**

   - **Opis:** Kolejka o nieograniczonej pojemności (ograniczenie zależy od pamięci).
   - **Przykład:**

```
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.BlockingQueue;

public class Main {
    public static void main(String[] args) throws InterruptedException
{
        BlockingQueue<String> queue = new LinkedBlockingQueue<>();
        queue.put("Item1");
        queue.put("Item2");
        queue.put("Item3");
        System.out.println(queue.take()); // Item1
        System.out.println(queue.take()); // Item2
        System.out.println(queue.take()); // Item3
    }
}
```

3. **PriorityBlockingQueue**

   - **Opis:** Kolejka z priorytetami, gdzie elementy są sortowane według ich naturalnego porządku lub `Comparator`.
   - **Przykład:**

```
import java.util.concurrent.PriorityBlockingQueue;
import java.util.concurrent.BlockingQueue;

public class Main {
    public static void main(String[] args) throws InterruptedException
{
        BlockingQueue<Integer> pq = new PriorityBlockingQueue<>();
        pq.put(40);
        pq.put(10);
        pq.put(30);
        pq.put(20);

        while(!pq.isEmpty()) {
            System.out.println(pq.take()); // 10, 20, 30, 40
        }
```

```
        }
    }
```

2. BlockingDeque

Opis:

- `BlockingDeque` to interfejs dla kolejek dwukierunkowych z mechanizmami blokującymi.
- Pozwala na operacje zarówno na początku, jak i na końcu kolejki.

Implementacje:

1. **LinkedBlockingDeque**
    - **Opis:** Dwukierunkowa kolejka z możliwością blokowania operacji.
    - **Przykład:**

```
import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.BlockingDeque;

public class Main {
    public static void main(String[] args) throws InterruptedException
{
        BlockingDeque<String> deque = new LinkedBlockingDeque<>();
        deque.putFirst("Start");
        deque.putLast("End");
        deque.putFirst("Begin");

        System.out.println(deque.takeFirst()); // Begin
        System.out.println(deque.takeLast());  // End
        System.out.println(deque.takeFirst()); // Start
    }
}
```

Dodatkowy Przykład z Wieloma Wątkami:

```
import java.util.concurrent.LinkedBlockingDeque;
import java.util.concurrent.BlockingDeque;

class Producer implements Runnable {
    private BlockingDeque<String> deque;

    Producer(BlockingDeque<String> deque) {
        this.deque = deque;
    }

    public void run() {
        try {
            deque.putLast("Element from Producer");
            System.out.println("Producer dodał element");
```

```
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }

    class Consumer implements Runnable {
        private BlockingDeque<String> deque;

        Consumer(BlockingDeque<String> deque) {
            this.deque = deque;
        }

        public void run() {
            try {
                String item = deque.takeFirst();
                System.out.println("Consumer pobrał: " + item);
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }

    public class Main {
        public static void main(String[] args) {
            BlockingDeque<String> deque = new LinkedBlockingDeque<>();

            Thread producer = new Thread(new Producer(deque));
            Thread consumer = new Thread(new Consumer(deque));

            producer.start();
            consumer.start();
        }
    }
```

Przykładowy Wynik:

```
Producer dodał element
Consumer pobrał: Element from Producer
```

Uwagi:

- **Mechanizmy Blokujące:** Operacje takie jak `put`, `take`, `offer`, `poll` mogą blokować wątki, jeśli kolejka jest pełna lub pusta.
- **Bezpieczeństwo Wielowątkowe:** `BlockingQueue` i `BlockingDeque` są zaprojektowane do bezpiecznej współpracy w środowiskach wielowątkowych bez konieczności ręcznej synchronizacji.

Strumienie Danych (Streams)

1. Co to jest Stream API?

Opis:

- Stream API to zestaw narzędzi w Javie (od wersji 8), które umożliwiają przetwarzanie kolekcji danych w sposób deklaratywny i równoległy.
- Umożliwia operacje takie jak filtrowanie, mapowanie, sortowanie, redukcja itp., bez konieczności pisania zagnieżdżonych pętli.

Cechy Stream API:

- **Deklaratywne:** Skupia się na "co" robić, a nie "jak" to robić.
- **Łańcuchowe Operacje:** Pozwala na łączenie wielu operacji w jeden potok.
- **Lazy Evaluation:** Operacje są wykonywane dopiero, gdy jest to potrzebne (np. terminalne operacje).
- **Równoległość:** Umożliwia łatwe przetwarzanie danych w wielu wątkach.

2. Przykład Użycia Stream API

Filtracja i Wyświetlanie Elementów:

```
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<String> names = Arrays.asList("Alice", "Bob", "Charlie", "Amanda");

        names.stream()
             .filter(name -> name.startsWith("A"))
             .forEach(System.out::println);
        // Wynik:
        // Alice
        // Amanda
    }
}
```

Wyjaśnienie:

1. `names.stream()`
    - Tworzy strumień danych z listy `names`.
2. `.filter(name -> name.startsWith("A"))`
    - Filtruje elementy, pozostawiając tylko te, które zaczynają się na literę "A".
3. `.forEach(System.out::println)`
    - Iteruje po przefiltrowanych elementach i wyświetla je.

Dodatkowe Przykłady:

1. **Mapowanie Elementów (Przekształcanie):**

```
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<String> words = Arrays.asList("java", "stream", "api");

        words.stream()
             .map(String::toUpperCase)
             .forEach(System.out::println);
        // Wynik:
        // JAVA
        // STREAM
        // API
    }
}
```

2. **Sortowanie Elementów:**

```
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(5, 3, 8, 1, 2);

        numbers.stream()
               .sorted()
               .forEach(System.out::println);
        // Wynik:
        // 1
        // 2
        // 3
        // 5
        // 8
    }
}
```

3. **Redukcja Elementów (`reduce`):**

```
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);
```

```
        int sum = numbers.stream()
                         .reduce(0, (a, b) -> a + b);
        System.out.println("Sum: " + sum); // Sum: 15
    }
}
```

4. **Praca z Obiektami:**

```
import java.util.Arrays;
import java.util.List;

class Person {
    String name;
    int age;

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

    public String getName() {
        return name;
    }

    public int getAge() {
        return age;
    }
}

public class Main {
    public static void main(String[] args) {
        List<Person> people = Arrays.asList(
            new Person("Alice", 30),
            new Person("Bob", 25),
            new Person("Charlie", 35)
        );

        people.stream()
              .filter(p -> p.getAge() > 28)
              .map(Person::getName)
              .forEach(System.out::println);
        // Wynik:
        // Alice
        // Charlie
    }
}
```

5. **Zliczanie Elementów:**

```
import java.util.Arrays;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        List<String> items = Arrays.asList("apple", "banana", "apple",
"cherry");

        long count = items.stream()
                          .filter(item -> item.equals("apple"))
                          .count();
        System.out.println("Liczba jabłek: " + count); // Liczba jabłek: 2
    }
}
```

6. **Praca z Kolekcjami Współbieżnymi:**

```
import java.util.Arrays;
import java.util.List;
import java.util.concurrent.CopyOnWriteArrayList;

public class Main {
    public static void main(String[] args) {
        List<String> list = new CopyOnWriteArrayList<>(Arrays.asList("A",
"B", "C", "D"));

        list.stream()
            .filter(s -> !s.equals("B"))
            .forEach(s -> list.remove(s));

        System.out.println(list); // [A, B, C, D]
    }
}
```

Uwagi: `CopyOnWriteArrayList` pozwala na bezpieczne modyfikacje podczas iteracji.

Uwagi:

- **Terminalne Operacje:** Operacje, które kończą strumień, np. `forEach`, `collect`, `reduce`.
- **Intermediate Operations:** Operacje, które zwracają strumień, np. `filter`, `map`, `sorted`.
- **Lazy Evaluation:** Operacje pośrednie są wykonywane dopiero, gdy następuje operacja terminalna.
- **Równoległość:** Możemy używać `parallelStream()` do równoległego przetwarzania danych, co może zwiększyć wydajność na dużych zbiorach danych.

Przykład z `parallelStream()`:

```
import java.util.Arrays;
import java.util.List;
```

```
public class Main {
    public static void main(String[] args) {
        List<Integer> numbers = Arrays.asList(1, 2, 3, 4, 5);

        int sum = numbers.parallelStream()
                         .filter(n -> n % 2 == 0)
                         .mapToInt(n -> n)
                         .sum();
        System.out.println("Suma parzystych liczb: " + sum); // Suma parzystych
liczb: 6
    }
}
```

Uwaga:

- **Strumienie Jednorazowe:** Strumień może być używany tylko raz. Próba ponownego użycia spowoduje `IllegalStateException`.