Dany jest fragment. które deklaracje są poprawne:

1. **class A {} – poprawne**
2. class B implements A {}
3. **interface I1 {} – poprawne**
4. **interface I2 {} – poprawne**
5. interface I3 {} implements I1 {}
6. interface I4 {} implements A {}
7. interface I5 {} extends A {}
8. **interface I6 {} extends I2 {} – poprawne**
9. class C extends A, B {}
10. **class D implements I2, I1 {} – poprawne**
11. **interface I7 extends I2, I1 {} – poprawne**
12. **class E extends A implements I2 {} – poprawne**
13. **class F extends A implements I2, I1 {} – poprawne**

Analiza deklaracji:

1. `class A {}`
   Poprawna deklaracja. Jest to podstawowa klasa w Javie, która nie rozszerza żadnej innej klasy ani nie implementuje interfejsów.
2. `class B implements A {}`
   Niepoprawna deklaracja. Klasa nie może implementować innej klasy, ponieważ słowo kluczowe `implements` jest używane tylko do implementacji interfejsów. Klasa powinna dziedziczyć po innej klasie za pomocą `extends`.
3. `interface I1 {}`
   Poprawna deklaracja. Jest to poprawny interfejs w Javie.
4. `interface I2 {}`
   Poprawna deklaracja. Jest to kolejny poprawny interfejs w Javie.
5. `interface I3 {} implements I1 {}`
   Niepoprawna deklaracja. Interfejsy w Javie mogą jedynie rozszerzać inne interfejsy za pomocą słowa kluczowego `extends`. Słowo kluczowe `implements` jest zarezerwowane dla klas implementujących interfejsy.
6. `interface I4 {} implements A {}`
   Niepoprawna deklaracja. Interfejsy nie mogą implementować klas, ponieważ klasy zawierają implementację metod, a interfejsy są abstrakcyjne.
7. `interface I5 {} extends A {}`
   Niepoprawna deklaracja. Interfejsy mogą rozszerzać tylko inne interfejsy, a nie klasy.
8. `interface I6 {} extends I2 {}`
   Poprawna deklaracja. Interfejs może rozszerzać inny interfejs.
9. `class C extends A, B {}`
   Niepoprawna deklaracja. W Javie klasa może dziedziczyć tylko po jednej klasie (brak wielokrotnego dziedziczenia klas). Można jednak implementować wiele interfejsów.
10. `class D implements I2, I1 {}`
    Poprawna deklaracja. Klasa może implementować wiele interfejsów.
11. `interface I7 extends I2, I1 {}`
    Poprawna deklaracja. Interfejs może rozszerzać wiele innych interfejsów.
12. `class E extends A implements I2 {}`
    Poprawna deklaracja. Klasa może dziedziczyć po jednej klasie i jednocześnie implementować dowolną liczbę interfejsów.
13. `class F extends A implements I2, I1 {}`
    Poprawna deklaracja. Klasa może dziedziczyć po jednej klasie i implementować wiele interfejsów.

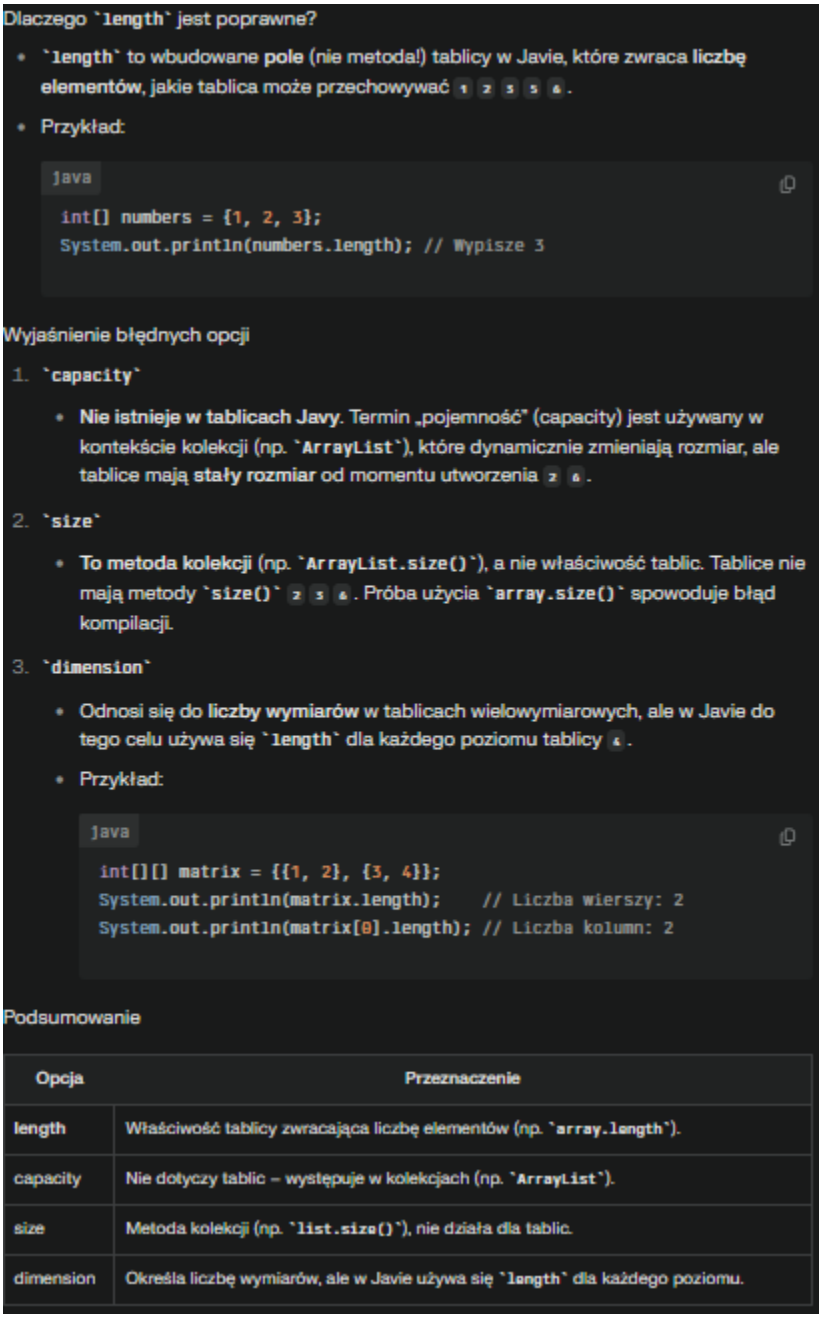
Co wypisze program?

```java
// Definicje wyjątków
class Wyjatek1 extends Exception {}  // Wyjątek typu 1
class Wyjatek2 extends Exception {}  // Wyjątek typu 2

class Program {
    // Metoda deklaruje, że może rzucić Wyjatek1 lub Wyjatek2
    void zlaMetoda() throws Wyjatek1, Wyjatek2 {
        throw new Wyjatek2(); // Rzucamy konkretnie Wyjatek2
    }
}

public class Main {
    public static void main(String[] args) {
        Program a = new Program();

        try {
            a.zlaMetoda();       // Wywołanie metody, która RZUCA
Wyjatek2
            System.out.println("a"); // Ta linia NIGDY nie zostanie
wykonana
        }
        catch (Wyjatek1 e) {     // Łapiemy Wyjatek1 (NIE PASUJE do
rzuconego Wyjatek2)
            System.out.println("b");
        }
        catch (Wyjatek2 e) {     // Łapiemy Wyjatek2 (PASUJE)
            System.out.println("c");
        }
        finally {                // Zawsze wykonany, nawet po wyjątku
            System.out.println("d");
        }
    }
}
```

```
C:\Users\Michal\.jdks\openjdk-20.0.1\bin\java.exe "
c
d
```

Jak nazywa się własność każdej tablicy języka java, która umożliwia odczytanie jej aktualnego rozmiaru?

- capacity
- **length – poprawna**
- size
- dimension



Dlaczego `length` jest poprawne?

- `length` to wbudowane **pole** (nie metoda!) tablicy w Javie, które zwraca **liczbę elementów**, jakie tablica może przechowywać [1] [2] [5] [5] [6].
- Przykład:

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

Wyjaśnienie błędnych opcji

1. `capacity`
   - **Nie istnieje w tablicach Javy.** Termin „pojemność" (capacity) jest używany w kontekście kolekcji (np. `ArrayList`), które dynamicznie zmieniają rozmiar, ale tablice mają **stały rozmiar** od momentu utworzenia [2] [6].
2. `size`
   - **To metoda kolekcji** (np. `ArrayList.size()`), a nie właściwość tablic. Tablice nie mają metody `size()` [2] [3] [6]. Próba użycia `array.size()` spowoduje błąd kompilacji.
3. `dimension`
   - Odnosi się do **liczby wymiarów** w tablicach wielowymiarowych, ale w Javie do tego celu używa się `length` dla każdego poziomu tablicy [6].
   - Przykład:

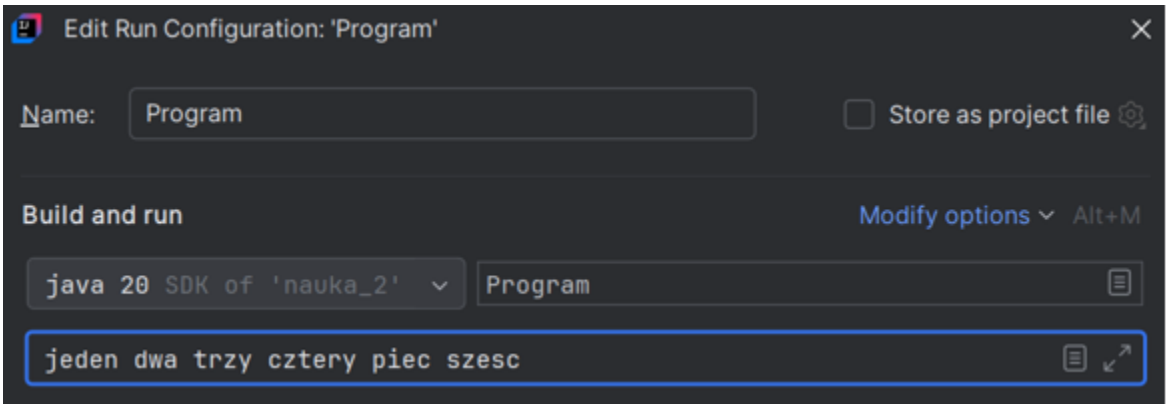
```java
int[][] matrix = {{1, 2}, {3, 4}};
System.out.println(matrix.length);    // Liczba wierszy: 2
System.out.println(matrix[0].length); // Liczba kolumn: 2
```

Podsumowanie

Opcja	Przeznaczenie
length	Właściwość tablicy zwracająca liczbę elementów (np. `array.length`).
capacity	Nie dotyczy tablic – występuje w kolekcjach (np. `ArrayList`).
size	Metoda kolekcji (np. `list.size()`), nie działa dla tablic.
dimension	Określa liczbę wymiarów, ale w Javie używa się `length` dla każdego poziomu.

Dana jest klasa przedstawiona na obrazku. Program uruchomiono w następujący sposób (bez cudzysłowiu): „Java Program jeden dwa trzy cztery piec szesc”. Co zostanie wypisane na ekranie

```java
class Program {
    public static void main(String[] args) {
        System.out.println(args[1] + "/" + args[2] + "/" + args[3]);
    }
}
```



1. **Struktura tablicy `args`:**
   - Argumenty przekazywane do programu są przechowywane w tablicy `args`.
   - **Indeksowanie zaczyna się od 0.**
   - Po uruchomieniu powyższą komendą, tablica `args` wygląda następująco:

```text
args[0] → "jeden"
args[1] → "dwa"
args[2] → "trzy"
args[3] → "cztery"
args[4] → "piec"
args[5] → "szesc"
```

2. **Instrukcja `System.out.println`:**
   - Program łączy trzy elementy tablicy `args` za pomocą znaku `/`:
     - `args` → **"dwa"**,
     - `args` → **"trzy"**,
     - `args` → **"cztery"**.
   - Wynikiem jest ciąg: **"dwa/trzy/cztery"**.

```
C:\Users\Michal\.jdks\openjdk-20.0.1\b
dwa/trzy/cztery

Process finished with exit code 0
```

Dane są klasy. Co się stanie po uruchomieniu programu

```java
class Auto {
    String marka;

    public Auto(String marka) {
        this.marka = marka;
    }

    public void jedzie() {
        System.out.println("Auto jedzie");
    }
}

class Toyota extends Auto {
    public Toyota() {
        super("Toyota"); // Wywołanie konstruktora klasy Auto z
argumentem "Toyota"
    }

    // Przesłonięcie metody jedzie() z klasy Auto
    @Override
    public void jedzie() {
        System.out.println("Jedzie Toyota");
    }

    // Przeciążenie metody jedzie() - dodatkowy parametr
    public void jedzie(String model) {
        System.out.println("Jedzie Toyota " + model);
    }
}

public class Program {
    public static void main(String[] args) {
        Auto a1 = new Toyota(); // Tworzenie obiektu Toyota, typ
referencji: Auto
        Toyota a2 = new Toyota(); // Tworzenie obiektu Toyota, typ
referencji: Toyota

        a1.jedzie();    // Wywołanie metody przesłoniętej
        a2.jedzie("Aigo"); // Wywołanie metody przeciążonej
    }
}
```

```
C:\Users\Michal\.jdks\openjd
Jedzie Toyota
Jedzie Toyota Aigo
```

Dany jest kod źródłowy. Co się stanie jeśli spróbujemy skompilować i uruchomić ten program.

```java
class Test {
    static int x;  // Zmienna statyczna (współdzielona przez wszystkie obiekty)
    int k;         // Zmienna instancji (unikalna dla każdego obiektu)

    public Test(int n, int m) {
        x = n;     // Modyfikacja zmiennej STATYCZNEJ (dotyczy całej klasy)
        k = m;     // Modyfikacja zmiennej instancji (dotyczy tylko tego obiektu)
    }

    public static void main(String[] args) {
        Test t1 = new Test(10, 20); // Tworzenie obiektu t1
        Test t2 = new Test(30, 40); // Tworzenie obiektu t2

        // Wypisywanie wartości
        System.out.print(t1.x + " "); // 30 (statyczne x po ostatniej modyfikacji)
        System.out.print(t1.k + " "); // 20 (k dla t1)
        System.out.print(t2.x + " "); // 30 (statyczne x wspólne dla wszystkich)
        System.out.print(t2.k);       // 40 (k dla t2)
    }
}
```

Kluczowe różnice między zmiennymi:

Właściwość	Zmienna statyczna (`x`)	Zmienna instancji (`k`)
Przynależność	Klasa (wspólna dla wszystkich obiektów)	Obiekt (unikalna dla każdego obiektu)
Modyfikacja	Zmiana dotyczy wszystkich obiektów	Zmiana dotyczy tylko danego obiektu
Dostęp	`NazwaKlasy.x` lub `obiekt.x`	Tylko przez obiekt (`obiekt.k`)

Dlaczego `t1.x` wynosi 30, a nie 10?

- Zmienna statyczna `x` jest **współdzielona** przez wszystkie obiekty klasy `Test`.
- Gdy tworzymy `t2`, nadpisujemy `x` na **30**, co zmienia wartość dla **całej klasy** (więc również dla `t1`).

```
C:\Users\Michal\.jdks\
30 20 30 40
```

W jakich nawiasach podajemy typ podczas definiowania klasy uogólnionej

Odpowiedź:

Podczas definiowania klasy uogólnionej (generycznej) w Javie typ podajemy w **nawiasach ostrych** `<>`. Przykład deklaracji klasy generycznej:

```java
public class KlasaGeneryczna<T> {
    // ciało klasy
}
```

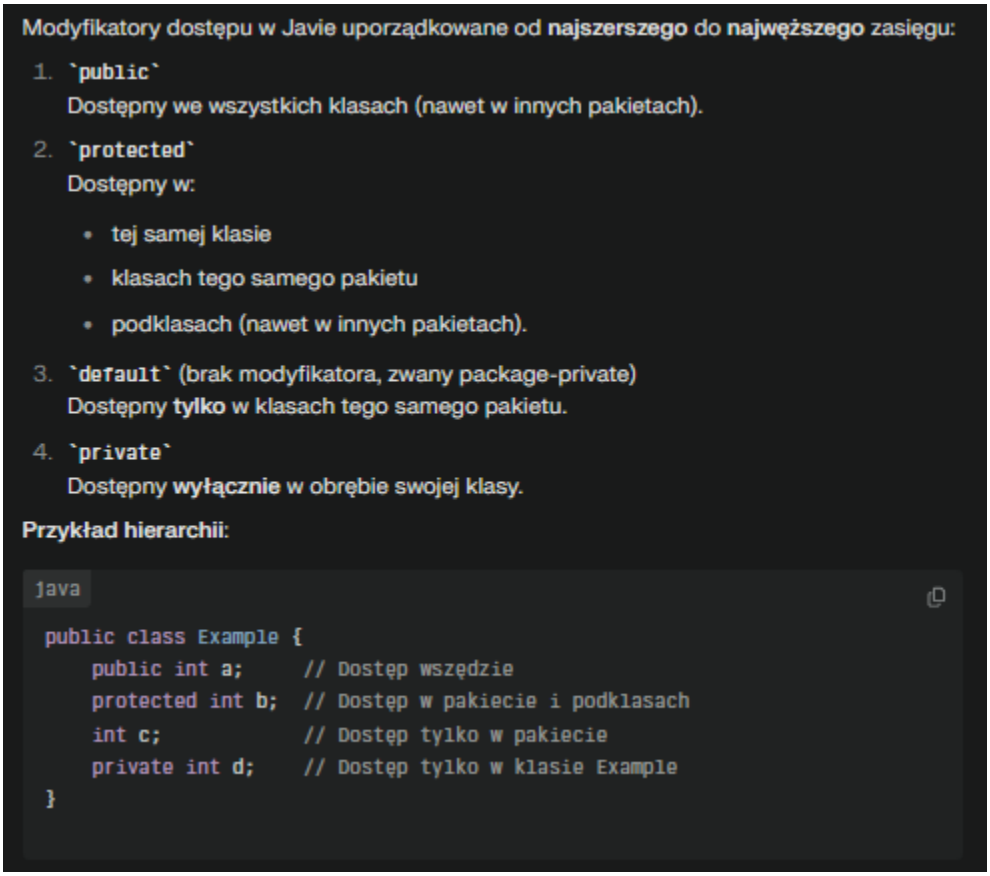
Gdzie `T` to parametr typu (może to być dowolna litera, np. `E`, `K`, `V`).

Wyjaśnienie pozostałych nawiasów w Javie:

1. `()` – **Nawiasy okrągłe (parentheses)**
   - Używane do:
     - Definiowania parametrów metod: `void metoda(int x) { ... }`.
     - Wywoływania metod: `metoda(5);`.
     - Grupowania wyrażeń logicznych: `if (a && (b || c))`.
2. `[]` – **Nawiasy kwadratowe (square brackets)**
   - Używane do:
     - Deklaracji tablic: `int[] tablica = new int[5];`.
     - Dostępu do elementów tablicy: `tablica = 10;`.
     - Typów generycznych w niektórych językach (ale **nie w Javie!**).
3. `{}` – **Nawiasy klamrowe (curly braces)**
   - Używane do:
     - Określania zakresu (bloków kodu): `if (x > 0) { ... }`.
     - Inicjalizacji tablic: `int[] liczby = {1, 2, 3};`.
     - Definiowania klas/metod: `class A { ... }`.
4. `<>` – **Nawiasy ostre (angle brackets)**
   - Używane **tylko dla generyków**:
     - Deklaracja klasy generycznej: `class Lista<T> { ... }`.
     - Określanie konkretnego typu: `Lista<String> lista = new Lista<>();`.
     - Metody generyczne: `<T> void metoda(T arg) { ... }`.

- `()`: Są zarezerwowane dla metod i wyrażeń.
- `[]`: Są związane z tablicami.
- `{}`: Określają bloki kodu, nie typy.

Modyfikatory dostępu to: 1 default; 2 public; 3 private; 4 protected. Uszereguj modyfikatory dostępu od najobszerniejszego do najwęższego.



Modyfikatory dostępu w Javie uporządkowane od **najszerszego** do **najwęższego** zasięgu:

1. `public`
   Dostępny we wszystkich klasach (nawet w innych pakietach).
2. `protected`
   Dostępny w:
   - tej samej klasie
   - klasach tego samego pakietu
   - podklasach (nawet w innych pakietach).
3. `default` (brak modyfikatora, zwany package-private)
   Dostępny **tylko** w klasach tego samego pakietu.
4. `private`
   Dostępny **wyłącznie** w obrębie swojej klasy.

Przykład hierarchii:

```java
public class Example {
    public int a;     // Dostęp wszędzie
    protected int b;  // Dostęp w pakiecie i podklasach
    int c;            // Dostęp tylko w pakiecie
    private int d;    // Dostęp tylko w klasie Example
}
```

2 4 1 3

```java
class A {
    int n; // Pole przechowujące wartość liczbową

    // Konstruktor klasy A
    public A(int n) {
        this.n = n; // Przypisanie wartości do pola 'n'
    }

    // Metoda suma: oblicza sumę liczb od n1 do n2 (włącznie)
    public int suma(int n1, int n2) {
        int s = 0;
        for (int i = n1; i <= n2; i++) {
            s += i; // Dodawanie kolejnych liczb do zmiennej 's'
        }
        return s; // Zwrócenie sumy
    }
}
```

```
class B extends A {
    // Konstruktor klasy B: wywołuje konstruktor klasy A
    public B(int n) {
        super(n); // super(n) przekazuje wartość 'n' do konstruktora
klasy A
    }

    // Metoda suma: przeciążenie metody z klasy A (trzy parametry)
    public int suma(int k1, int k2, int k3) {
        int s = 0;
        int k = k1;
        while (k <= k2) {
            s += k; // Dodawanie liczb od k1 do k2
            k++;
        }
        return s + k3; // Suma liczb od k1 do k2 + wartość k3
    }
}
```

Tutaj jest przeciążenie metod (overloading)

- **Przeciążanie**: Nowa metoda o **tej samej nazwie**, ale **innych parametrach**.
- **Przesłanianie**: Nadpisanie istniejącej metody z **identyczną sygnaturą**.
- W podanym kodzie mamy do czynienia **tylko z przeciążaniem**.

Jaka jest minimalne ilość danych byte?

Odp: - 128

1. byte (8-bitowy, ze znakiem)

- **Minimalna wartość:** `-128`
- **Maksymalna wartość:** `127`

```java
byte min = Byte.MIN_VALUE; // -128
byte max = Byte.MAX_VALUE; // 127
```

2. short (16-bitowy, ze znakiem)

- **Minimalna wartość:** `-32,768`
- **Maksymalna wartość:** `32,767`

```java
short min = Short.MIN_VALUE; // -32768
short max = Short.MAX_VALUE; // 32767
```

3. int (32-bitowy, ze znakiem)

- **Minimalna wartość:** `-2,147,483,648`
- **Maksymalna wartość:** `2,147,483,647`

```java
int min = Integer.MIN_VALUE; // -2147483648
int max = Integer.MAX_VALUE; // 2147483647
```

4. long (64-bitowy, ze znakiem)

- **Minimalna wartość:** `-9,223,372,036,854,775,808`
- **Maksymalna wartość:** `9,223,372,036,854,775,807`

```java
long min = Long.MIN_VALUE; // -9223372036854775808L
long max = Long.MAX_VALUE; // 9223372036854775807L
```

5. float (32-bitowy, zmiennoprzecinkowy)

- **Minimalna dodatnia wartość:** `1.4E-45`
- **Maksymalna wartość:** `3.4028235E38`

```java
float min = Float.MIN_VALUE; // 1.4E-45
float max = Float.MAX_VALUE; // 3.4028235E38
```

6. double (64-bitowy, zmiennoprzecinkowy)

- **Minimalna dodatnia wartość:** `4.9E-324`
- **Maksymalna wartość:** `1.7976931348623157E308`

```java
double min = Double.MIN_VALUE; // 4.9E-324
double max = Double.MAX_VALUE; // 1.7976931348623157E308
```

7. char (16-bitowy, bez znaku – kod Unicode)

- **Minimalna wartość:** `\u0000` (0)
- **Maksymalna wartość:** `\uffff` (65,535)

```java
char min = Character.MIN_VALUE; // '\u0000'
char max = Character.MAX_VALUE; // '\uffff'
```

8. boolean (wartość logiczna)

- **Dozwolone wartości:** `true` lub `false`

```java
boolean flag = true; // lub false
```

Czy klasa może mieć więcej niż jeden konstruktor bezparametrowy?

- Odp: NIE. Jedna klasa może zawierać **co najwyżej jeden konstruktor bezparametrowy**.

Elementami kodu źródłowego mogą być: 1 komentarze; 2 deklaracje import; 3 deklaracje package; 4 metody; 5 deklaracje klasy; 6 zmienne. Która z poniższych kolejności występowania jest dopuszczalna?

Poprawna kolejność elementów w kodzie źródłowym Javy to **3 → 2 → 1 → 5 → 6 → 4**, odpowiadająca sekwencji **321564**.

Objaśnienie:

1. **Deklaracja pakietu (`package`)** – zawsze na początku pliku (jeśli istnieje).
2. **Importy (`import`)** – po deklaracji pakietu, przed klasą.
3. **Komentarze** – mogą pojawić się przed klasą (np. opis klasy).
4. **Deklaracja klasy** – obowiązkowa, otacza zmienne i metody.
5. **Zmienne** – zadeklarowane wewnątrz klasy (przed metodami zgodnie z konwencją).
6. **Metody** – umieszczane po zmiennych wewnątrz klasy.

Przykład struktury:

```java
// 3. Deklaracja pakietu
package pl.example;

// 2. Importy
import java.util.List;

// 1. Komentarz przed klasą
/**
 * Klasa demonstrująca poprawną kolejność.
 */
// 5. Deklaracja klasy
public class Przyklad {
    // 6. Zmienne
    private int liczba;

    // 4. Metody
    public void metoda() {}
}
```

Dany jest fragment (obrazek). Jaki będzie wynik działania programu?

```java
// Abstrakcyjna klasa bazowa Figura
abstract class Figura {
    // Dwie metody abstrakcyjne WYMAGAJĄCE implementacji w podklasach
    abstract void rysuj();       // Brak implementacji!
    abstract double obliczPole();// Brak implementacji!
}

// Klasa Kwadrat dziedziczy po Figura
class Kwadrat extends Figura {
    double a; // Długość boku kwadratu

    public Kwadrat(double a) {
        this.a = a;
    }

    // Implementacja JEDNEJ metody abstrakcyjnej (obliczPole)
    @Override
    double obliczPole() {
        return a * a; // Poprawna implementacja pola kwadratu
    }

    // BRAKUJE implementacji metody rysuj() → BŁĄD KOMPILACJI!
}

public class Program {
    public static void main(String[] args) {
        Kwadrat k = new Kwadrat(2);
        double P = k.obliczPole();
        System.out.print("Pole kwadratu = " + P); // Gdyby kod działał:
"Pole kwadratu = 4.0"
    }
}
```

Odpowiedź:
Program **nie skompiluje się** i zgłosi błąd. Powodem jest brak implementacji metody abstrakcyjnej `rysuj()` w klasie `Kwadrat`, która dziedziczy po abstrakcyjnej klasie `Figura`.