Wstęp do programowania w języku Java

Poniższe notatki mają na celu wprowadzenie w podstawowe zagadnienia języka Java. Zostały one pogrupowane na kilka głównych tematów. Po omówieniu każdego zagadnienia znajdziesz propozycje pytań wraz z przykładowymi odpowiedziami, abyś mogła/mógł się sprawdzić.

---

1. Podstawowe typy danych oraz instrukcje warunkowe

1.1. Podstawowe typy danych w Java

W języku Java występują dwie główne kategorie typów danych:

1. **Typy proste (prymitywne)**
2. **Typy obiektowe (referencyjne)**

Typy proste (prymitywne)

- **byte** – 8-bitowy typ całkowitoliczbowy (zakres: -128 do 127).
- **short** – 16-bitowy typ całkowitoliczbowy (zakres: -32768 do 32767).
- **int** – 32-bitowy typ całkowitoliczbowy (zakres: -2^31 do 2^31 - 1).
- **long** – 64-bitowy typ całkowitoliczbowy (zakres: -2^63 do 2^63 - 1).
- **float** – 32-bitowy typ zmiennoprzecinkowy (pojedynczej precyzji).
- **double** – 64-bitowy typ zmiennoprzecinkowy (podwójnej precyzji).
- **char** – 16-bitowy typ znakowy (przechowuje znaki Unicode).
- **boolean** – typ logiczny (może przyjmować wartości `true` lub `false`).

Przykładowe deklaracje:

```
byte smallNumber = 10;
int age = 25;
long largeNumber = 123456789L;
float price = 19.99f;
double pi = 3.14159265359;
char initial = 'A';
boolean isStudent = true;
```

Typy obiektowe (referencyjne)

Do tej grupy należą m.in. **String**, klasy zdefiniowane przez użytkownika oraz inne typy, które dziedziczą po klasie bazowej `Object`. Przykład:

```
String name = "Ala";
```

1.2. Instrukcje warunkowe

W języku Java wyróżniamy głównie następujące instrukcje warunkowe:

- `if`, `else if`, `else`
- `switch`

`if`, `else if`, `else`

```
int number = 10;

if (number > 0) {
    System.out.println("Liczba dodatnia");
} else if (number < 0) {
    System.out.println("Liczba ujemna");
} else {
    System.out.println("Zero");
}
```

`switch`

Instrukcja `switch` pozwala na sprawdzenie wartości zmiennej w wielu przypadkach (tzw. case). Przykład:

```
int day = 3;
String dayName;

switch (day) {
    case 1:
        dayName = "Poniedziałek";
        break;
    case 2:
        dayName = "Wtorek";
        break;
    case 3:
        dayName = "Środa";
        break;
    default:
        dayName = "Inny dzień";
}

System.out.println("Dzień: " + dayName);
```

1.3. Pytania sprawdzające

1. **Jakie są podstawowe typy prymitywne w Java?**
    - **Odpowiedź**: byte, short, int, long, float, double, boolean, char.
2. **Do czego służy instrukcja `switch`?**

- **Odpowiedź**: Do sprawdzenia wartości zmiennej w wielu przypadkach (tzw. case) i wykonywania odpowiedniego bloku kodu zależnie od tej wartości.

3. **Czym różni się typ prymitywny od referencyjnego?**
   - **Odpowiedź**: Typy prymitywne przechowują bezpośrednio wartości w pamięci, natomiast typy referencyjne przechowują odwołanie (referencję) do obiektu w pamięci.

---

2. Działanie różnych typów pętli w Java

W języku Java wyróżniamy głównie następujące rodzaje pętli:

- `for`
- `while`
- `do-while`
- `for-each` (z użyciem konstrukcji `for(Type var : collection)`)

2.1. Pętla `for`

Najbardziej klasyczna forma pętli. Składa się z:

1. Inicjalizacji zmiennej sterującej (np. `int i = 0;`).
2. Warunku kontynuacji (np. `i < 10`).
3. Inkrementacji/dekrementacji (np. `i++`).

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

2.2. Pętla `while`

Wykonuje blok kodu dopóki warunek jest spełniony (true).

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

2.3. Pętla `do-while`

Podobna do `while`, ale instrukcje wewnątrz bloku są wykonywane przynajmniej raz, **zanim** sprawdzony zostanie warunek.

```
int i = 0;
do {
```

```
    System.out.println("i = " + i);
    i++;
} while (i < 5);
```

2.4. Pętla `for-each`

Służy do iteracji po kolekcjach i tablicach w prostszy sposób:

```
String[] names = {"Ala", "Ola", "Ela"};
for (String name : names) {
    System.out.println(name);
}
```

2.5. Pytania sprawdzające

1. **Czym różni się pętla `while` od `do-while`?**
    - **Odpowiedź**: Pętla `while` sprawdza warunek **przed** wykonaniem bloku kodu, natomiast `do-while` najpierw wykonuje blok kodu, a dopiero **potem** sprawdza warunek.
2. **Jak zdefiniować pętlę `for-each`?**
    - **Odpowiedź**: Używamy składni `for (Typ element : kolekcja) { ... }`, co pozwala na iterację po każdym elemencie kolekcji lub tablicy.
3. **Gdzie najczęściej stosuje się pętlę `for` z indeksem?**
    - **Odpowiedź**: Gdy chcemy mieć kontrolę nad indeksem, np. przy dostępie do elementów tablicy w sposób indeksowy lub gdy potrzebujemy licznika iteracji.

---

3. Zagadnienie tworzenia klas, obiektów oraz korzystanie z konstruktorów

3.1. Klasy i obiekty

- **Klasa** to podstawowa jednostka w Javie, która opisuje stan (pola) oraz zachowanie (metody) obiektu.
- **Obiekt** to konkretny egzemplarz klasy utworzony w pamięci podczas działania programu.

Przykład prostej klasy

```
public class Person {
    // Pola
    String name;
    int age;

    // Metody
    void sayHello() {
        System.out.println("Cześć! Mam na imię " + name + ".");
```

```
    }
}
```

3.2. Konstruktory

- Konstruktor to specjalna metoda wywoływana w momencie tworzenia obiektu.
- Ma taką samą nazwę jak klasa i nie posiada typu zwracanego (nawet `void`).

Przykład konstruktora

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

    // Konstruktor domyślny (bez parametrów)
    public Person() {
        this.name = "Nieznajomy";
        this.age = 0;
    }

    // Konstruktor z parametrami
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }

    void sayHello() {
        System.out.println("Cześć! Mam na imię " + name + ", mam " + age + "
lat.");
    }
}

// Tworzenie obiektów
Person p1 = new Person();                // wywołuje konstruktor domyślny
Person p2 = new Person("Ala", 25);       // wywołuje konstruktor z parametrami
```

3.3. Pytania sprawdzające

1. **Czym różni się klasa od obiektu?**
    - **Odpowiedź**: Klasa to definicja/plan, a obiekt to konkretny egzemplarz klasy, który powstaje podczas działania programu.
2. **Jaką nazwę nosi metoda wywoływana podczas tworzenia obiektu?**
    - **Odpowiedź**: Konstruktor.
3. **Czy konstruktor może zwracać wartość?**
    - **Odpowiedź**: Nie. Konstruktor nie ma typu zwracanego, nawet `void`.

4. Mechanizm dziedziczenia i kompozycji

4.1. Dziedziczenie

- Dziedziczenie pozwala jednej klasie (klasie pochodnej/podklasie) przejmować pola i metody innej klasy (klasy bazowej/nadklasy).
- Służy do ponownego wykorzystania kodu i zachowania tzw. zasady DRY (Don't Repeat Yourself).
- W Java używa się słowa kluczowego `extends` do zaznaczenia dziedziczenia.

```
class Animal {
    String name;

    public void eat() {
        System.out.println("Jem jedzenie.");
    }
}

class Dog extends Animal {
    public void bark() {
        System.out.println("Hau!");
    }
}

// Użycie:
Dog dog = new Dog();
dog.name = "Reksio";  // pole odziedziczone z Animal
dog.eat();            // metoda odziedziczona
dog.bark();           // metoda własna klasy Dog
```

4.2. Kompozycja

- Kompozycja polega na tym, że jedna klasa **zawiera** (ma w sobie) obiekt innej klasy jako pole.
- Zamiast dziedziczyć po klasie `Engine`, można stworzyć klasę `Car`, która zawiera pole `Engine engine;`.

```
class Engine {
    void start() {
        System.out.println("Silnik uruchomiony");
    }
}

class Car {
    // Kompozycja: obiekt klasy Engine wewnątrz Car
    private Engine engine;

    public Car() {
        this.engine = new Engine();
    }
```

```
    public void startCar() {
        engine.start();
        System.out.println("Samochód rusza");
    }
}
```

4.3. Pytania sprawdzające

1. **Czym różni się dziedziczenie od kompozycji?**
    - **Odpowiedź**: Dziedziczenie to rozszerzanie funkcjonalności klasy bazowej (podklasa przejmuje pola i metody). Kompozycja to włączanie obiektów innych klas jako składowych klasy.
2. **Jak w Javie definiujemy dziedziczenie między klasami?**
    - **Odpowiedź**: Za pomocą słowa kluczowego `extends`.
3. **Dlaczego warto stosować kompozycję zamiast dziedziczenia w wielu przypadkach?**
    - **Odpowiedź**: Kompozycja zapewnia większą elastyczność, mniejsze powiązanie między klasami, a także ułatwia ponowne wykorzystanie kodu (można podmieniać składowe klasy bez naruszenia hierarchii dziedziczenia).

---

5. Interfejsy, wyrażenia lambda i klasy wewnętrzne

5.1. Interfejsy

- Interfejs w Javie definiuje **zestaw metod**, które klasa **musi** zaimplementować (o ile klasa deklaruje, że implementuje dany interfejs).
- Słowo kluczowe: `interface`.
- Klasy używają słowa kluczowego `implements`, aby zaimplementować interfejs.

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

class Car implements Drivable {
    @Override
    public void drive() {
        System.out.println("Jadę samochodem!");
    }
}
```

5.2. Wyrażenia lambda (od Java 8)

- Umożliwiają przekazywanie fragmentu kodu jako argumentu.
- Składnia: `(parametry) -> { ciało wyrażenia }`
- Przykład – użycie w interfejsie funkcyjnym (interfejs z jedną metodą abstrakcyjną):

```
@FunctionalInterface
interface Calculator {
    int operation(int a, int b);
}

public class Main {
    public static void main(String[] args) {
        Calculator add = (x, y) -> x + y;
        System.out.println("2 + 3 = " + add.operation(2, 3));
    }
}
```

5.3. Klasy wewnętrzne

- Klasa zdefiniowana w obrębie innej klasy.
- Pomocne w sytuacjach, gdy klasa jest ściśle związana logicznie z inną klasą i nie jest potrzebna samodzielnie.

```
class Outer {
    private String message = "Hello";

    class Inner {
        void printMessage() {
            System.out.println("Message: " + message);
        }
    }
}
```

5.4. Pytania sprawdzające

1. **Czym jest interfejs w Javie?**
    - **Odpowiedź**: To zbiór metod abstrakcyjnych (i ewentualnie pól statycznych), które klasa implementująca interfejs musi zaimplementować.
2. **Do czego służą wyrażenia lambda w Javie?**
    - **Odpowiedź**: Pozwalają na przekazywanie kodu (funkcji) jako argumentu, ułatwiają pracę z interfejsami funkcyjnymi i upraszczają zapis kodu.
3. **Kiedy warto stosować klasy wewnętrzne?**
    - **Odpowiedź**: Gdy ich istnienie jest ściśle powiązane z klasą zewnętrzną i nie chcemy, by były wykorzystywane poza nią.

---

6. Obsługa wyjątków, asercji

6.1. Obsługa wyjątków

- Wyjątki to zdarzenia zachodzące w czasie wykonywania programu, sygnalizujące błąd lub nieoczekiwaną sytuację.
- Struktura obsługi wyjątków opiera się na słowach kluczowych: `try`, `catch`, `finally`, `throw`, `throws`.

```
try {
    int result = 10 / 0; // Tutaj wystąpi wyjątek ArithmeticException
} catch (ArithmeticException e) {
    System.out.println("Nie dziel przez zero!");
} finally {
    System.out.println("Zawsze się wykona.");
}
```

6.2. Asercje

- Służą do sprawdzania założeń w czasie wykonywania programu (debugowanie, testowanie).
- Słowo kluczowe `assert`.

```
int x = 5;
assert x > 0 : "x nie jest większe od zera!";
```

6.3. Pytania sprawdzające

1. **Co to jest wyjątek w Javie?**
    - **Odpowiedź**: To sytuacja błędu lub nieoczekiwanego zdarzenia w czasie wykonywania programu, sygnalizowana przez obiekt klasy rozszerzającej `Throwable`.
2. **Jak wygląda podstawowa konstrukcja obsługi wyjątków?**
    - **Odpowiedź**: `try { ... } catch (Exception e) { ... } finally { ... }`.
3. **Do czego służą asercje?**
    - **Odpowiedź**: Do weryfikacji założeń w trakcie działania programu i wczesnego wychwytywania błędów (głównie używane podczas debugowania).

---

7. Zagadnienia związane z programowaniem generycznym

- **Generyki** (ang. Generics) pozwalają na pisanie kodu, który może działać na różnych typach, zapewniając jednocześnie bezpieczeństwo typów w czasie kompilacji.

7.1. Klasa generyczna

```
public class Box<T> {
    private T value;

    public Box(T value) {
        this.value = value;
    }

    public T getValue() {
        return value;
    }
}

Box<String> box1 = new Box<>("Hello");
Box<Integer> box2 = new Box<>(123);
```

7.2. Metody generyczne

```
public class Utils {
    public static <T> void printArray(T[] array) {
        for (T element : array) {
            System.out.println(element);
        }
    }
}

String[] names = {"Ala", "Ola", "Ela"};
Utils.printArray(names);
```

7.3. Ograniczenia i zalety generyków

- **Zalety**: Bezpieczeństwo typów, mniej rzutowań (castów).
- **Ograniczenia**: Nie można tworzyć tablic typów sparametryzowanych (`new T[]` jest niedozwolone), nie można używać typów prymitywnych jako parametrów generycznych.

7.4. Pytania sprawdzające

1. **Co to są generyki w Javie?**
    - **Odpowiedź**: Mechanizm pozwalający na parametryzowanie klas i metod typami, co zapewnia bezpieczeństwo typów podczas kompilacji.
2. **Jak deklaruje się klasę generyczną?**
    - **Odpowiedź**: Dodając parametr typu w nawiasach ostrych, np. `class Klasa<T> { ... }`.
3. **Dlaczego nie można używać typów prymitywnych jako parametrów generycznych?**
    - **Odpowiedź**: Generyki działają tylko z typami obiektowymi, a typy prymitywne nie dziedziczą po `Object`.

8. Kolekcje w Java

8.1. Podstawowe interfejsy i klasy w pakiecie `java.util`

- **Collection** – główny interfejs kolekcji.
  - **List** – lista (np. `ArrayList`, `LinkedList`) – pozwala na duplikaty i zachowuje kolejność.
  - **Set** – zbiór (np. `HashSet`, `TreeSet`) – nie pozwala na duplikaty.
  - **Queue** – kolejka (np. `LinkedList`).
- **Map** – mapa (np. `HashMap`, `TreeMap`) – przechowuje pary klucz-wartość.

8.2. Przykłady użycia kolekcji

List (ArrayList)

```
List<String> list = new ArrayList<>();
list.add("Ala");
list.add("Ola");
list.add("Ela");
System.out.println(list.get(0)); // "Ala"
```

Set (HashSet)

```
Set<String> set = new HashSet<>();
set.add("Kot");
set.add("Pies");
set.add("Kot"); // duplikat, nie zostanie dodany
System.out.println(set); // ["Kot", "Pies"]
```

Map (HashMap)

```
Map<Integer, String> map = new HashMap<>();
map.put(1, "Jeden");
map.put(2, "Dwa");
map.put(1, "JedenZmieniony"); // nadpisze poprzednią wartość
System.out.println(map.get(1)); // "JedenZmieniony"
```

8.3. Pytania sprawdzające

1. **Jaki jest główny interfejs dla wszystkich kolekcji w Javie (z wyjątkiem map)?**
   - **Odpowiedź**: `Collection`.
2. **Jaką strukturę danych reprezentuje `List`?**

- **Odpowiedź**: Uporządkowaną listę elementów, która może zawierać duplikaty.

3. **Czym różni się `Set` od `List`?**
   - **Odpowiedź**: `Set` nie pozwala na duplikaty, natomiast `List` dopuszcza duplikaty i zachowuje kolejność dodawanych elementów.
