Notatka Jakub Wesoły

Języki programowania takie jak Java, C czy C# **zaliczają się do języków imperatywnych trzeciego poziomu**. Dla porównania, SQL czy Haskell reprezentują języki deklaratywne czwartego poziomu.

Języki imperatywne skupiają się na opisie kroków niezbędnych do wykonania zadania. Umożliwiają zmianę wartości zmiennych i przekazywanie typów. Często są one mniej bezpieczne przy pracy wielowątkowej.

Języki deklaratywne z kolei koncentrują się na tym, co ma być osiągnięte, bez szczegółowego wskazania jak to zrobić. Wykorzystują niemodyfikowalne typy i funkcje jako dane, co sprawia, że są bardziej odporne na błędy w środowiskach wielowątkowych.

W podejściu funkcyjnym **unika się mutowalności** – dane raz przypisane nie powinny być zmieniane. Java, mimo że nie jest językiem stricte funkcyjnym, umożliwia programowanie w tym stylu m.in. dzięki bibliotece Stream API.

Strumienie to narzędzie do przetwarzania zbiorów danych w sposób deklaratywny. Metoda stream() otwiera dostęp do zestawu operacji takich jak filter, map, czy forEach, pozwalających na manipulowanie danymi bez modyfikowania źródła. Typy danych są automatycznie rozpoznawane (typowanie wnioskowane), co upraszcza kod i zwiększa jego przejrzystość.

Jednak z racji na imperatywną naturę Javy, operacje na strumieniach bywają mniej wydajne niż w językach stricte funkcyjnych.

Wyrażenia lambda to forma funkcji anonimowych – nie posiadają nazw, deklarowane są tam, gdzie są potrzebne. Mają składnię:

(argumenty) -> { ciało funkcji }

(String s) -> System.out.println(s)

Kolejną istotną cechą programowania funkcyjnego jest **abstrakcja** – możemy zdefiniować ogólną strukturę funkcji (np. przez interfejs), nie podając jej konkretnej implementacji.

<u>Przykładowe interfejsy funkcyjne w Javie:</u>

1. **Function<T, R> – służy do przekształcania wartości typu T na R. Często wykorzystywana w map().**

```
Function<String, Integer> length = s -> s.length();

System.out.println(length.apply("Hello"));
```

2. **Predicate<T> – służy do sprawdzania warunków logicznych, zwraca true lub false. Wykorzystywany np. w filter().**

```
Predicate<String> startsWithA = s -> s.startsWith("A");

System.out.println(startsWithA.test("Ala"));
```

3. **Runnable – reprezentuje zadanie nieprzyjmujące parametrów i niezwracające wartości. Powszechnie stosowany przy tworzeniu wątków.**

```
Runnable task = () -> System.out.println("Running...");

new Thread(task).start();
```

4. **Iterable<T> – interfejs dla struktur, które można przeglądać (np. w pętli for-each).**

```
List<String> names = List.of("Jan", "Ola");

names.forEach(name -> System.out.println(name));
```

5. **Stream<T> – nowoczesna metoda pracy z kolekcjami w stylu funkcyjnym, pozwalająca na tworzenie tzw. potoku.**

```
List<String> names = List.of("Jan", "Anna", "Ola");

names.stream()

    .filter(s -> s.length() > 3)

    .map(String::toUpperCase)

    .forEach(System.out::println);
```

6. **Collection<E> – główny interfejs struktur danych (List, Set, Queue). Rozszerza Iterable.**

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

list.add("a");

list.add("b");

list.remove("a");

System.out.println(list.contains("b"));
```