Java teoria:

1. **Co zyskujemy pisząc:**

List myList = new ArrayList(); zamiast **ArrayList myList = new ArrayList();**

Odpowiedź: Używając słowa List tworzymy listę w sposób ogólny, przez co kod jest bardziej elastyczny. Jeśli w przyszłości stwierdzę że LinkedList będzie lepszy wyborem to wtedy mogę zmienić definicje listy.

2. **ArrayList a LinkedList:**

ArrayList:

- jest oparta na tablicy dynamicznej, co oznacza, że przechowuje elementy w ciągłej strukturze pamięci.

- Dodawanie elementów na końcu listy jest również efektywne (średnio O(1)), ale dodawanie w dowolnym innym miejscu lub usuwanie może być kosztowne (O(n)), ponieważ wymaga przesunięcia elementów.

Kiedy używamy:

- Częste operacje dostępu do elementów

- Dodawanie elementów na końcu

- Stały rozmiar lub rzadkie zmiany rozmiaru

LinkedList:

- jest implementacją listy dwukierunkowej, co oznacza, że każdy element przechowuje referencje do następnego i poprzedniego elementu.

- Dostęp do elementów przez indeks jest wolniejszy (O(n)), ponieważ wymaga przeszukiwania listy od początku lub końca.

Kiedy używamy:

- Częste operacje dodawania/usuwania elementów

- Kiedy lista ma dużą liczbę elementów i wymaga częstych modyfikacji

3. **HashMap, TreeMap, LinkedHashMap:**

Mapy w Javie to struktury danych, które przechowują elementy w postaci par klucz-wartość. Klucz służy do identyfikacji wartości, a każda wartość jest przypisana do unikalnego klucza.

HashMap:

- opiera się na tablicy mieszającej (hash table)

- brak gwarancji porządku przechowywania elementów

- HashMap dopuszcza null jako klucze i wartości

Używamy gdy:

- Kiedy wymagana jest szybka operacja wyszukiwania, dodawania lub usuwania

- Gdy nie zależy Ci na porządku elementów

- Duże zbiory danych

TreeMap:

- jest oparta na strukturze drzewa czerwono-czarnego, co zapewnia posortowany zbiór kluczy

- Elementy są przechowywane w porządku naturalnym (według klucza) lub według dostarczonego komparatora

- Nie pozwala na null jako klucze, ale wartości mogą być null

Używamy gdy:

- Kiedy wymagana jest posortowana kolejność kluczy

LinkedHashMap:

- LinkedHashMap rozszerza HashMap o mechanizm połączeń dwukierunkowych, dzięki czemu zachowuje kolejność wstawiania elementów lub kolejność ostatnio używanych

Używamy gdy:

- Gdy wymagana jest kolejność wstawiania

4. List, Map, Set:

List:

Charakterystyka:

- **Kolekcja, która przechowuje elementy w określonej kolejności**, co oznacza, że każdy element ma swój indeks.
- **Pozwala na duplikaty**, więc możesz mieć wiele takich samych elementów.
- **Przykłady implementacji**: `ArrayList`, `LinkedList`, `Vector`.

Kiedy używać `List`?

- **Gdy kolejność elementów ma znaczenie**: Jeśli ważne jest, aby elementy były przechowywane i przetwarzane w określonej kolejności (np. lista zadań do wykonania).
- **Gdy musisz mieć dostęp do elementów przez indeks**: Na przykład, jeśli często będziesz używać metod takich jak `get(int index)` lub `set(int index, E element)`.
- **Gdy dozwolone są duplikaty**: Jeśli lista może zawierać powtarzające się elementy (np. lista zakupów, gdzie ten sam produkt może się pojawić wielokrotnie).

Map:

Charakterystyka:

- **Kolekcja, która przechowuje pary klucz-wartość**. Klucze muszą być unikalne, ale wartości mogą się powtarzać.
- **Przechowuje dane w taki sposób, że dostęp do wartości odbywa się przez klucz**.
- **Przykłady implementacji**: `HashMap`, `TreeMap`, `LinkedHashMap`.

Kiedy używać `Map`?

- **Gdy dane muszą być przechowywane w powiązaniu klucz-wartość**: Na przykład, gdy chcesz zmapować nazwy na adresy, identyfikatory na dane użytkowników, itp.
- **Gdy musisz mieć szybki dostęp do wartości przez unikalne klucze**: Jeśli potrzebujesz szybko znajdować wartości na podstawie ich kluczy.
- **Gdy klucze są unikalne, ale wartości mogą się powtarzać**: Na przykład, w bazach danych, gdzie kluczem może być unikalny identyfikator, a wartością mogą być dane powiązane z tym identyfikatorem.

Set:

Charakterystyka:

- **Kolekcja, która przechowuje unikalne elementy**, tzn. nie pozwala na duplikaty.
- **Kolejność przechowywania nie jest gwarantowana** (w przypadku `HashSet`), ale istnieją implementacje (`LinkedHashSet`, `TreeSet`), które mogą zachować kolejność wstawiania lub sortowania.
- **Przykłady implementacji**: `HashSet`, `LinkedHashSet`, `TreeSet`.

Kiedy używać `Set`?

- **Gdy potrzebujesz unikalnych elementów**: Jeśli musisz upewnić się, że żaden element nie pojawi się więcej niż raz (np. zbiór identyfikatorów użytkowników).
- **Gdy kolejność nie ma znaczenia** lub gdy chcesz zachować unikalność przy jednoczesnym utrzymaniu porządku (używając `LinkedHashSet`) lub sortowania (używając `TreeSet`).
- **Gdy potrzebujesz szybkiego sprawdzania przynależności**: Sprawdzenie, czy element jest obecny w zbiorze (`contains()`) jest efektywne.

5. **Interfejs Comparable – jak go używać? jakie problemy rozwiązuje?**

Aby użyć Comparable, klasa musi go implementować.

Dzięki implementacji Comparable, nie ma potrzeby tworzenia zewnętrznych komparatorów dla podstawowego porównywania obiektów.

Dzięki temu można łatwo porównywać obiekty w kolekcjach.

6. **Sort, max**

Zastosowanie metody max(): Metoda Collections.max() znajduje obiekt Animal z największą ceną w kolekcji animalList

Zastosowanie metody sort():

a) W naturalnym porządku(np. alfabetycznie) przy użyciu interfajsu comparable
b) Przy użyciu comparatora, gdy chcemy dostosować proces porównywania
c) Przy użyciu funkcji Lambda

7. **Różnica między metodą equals a operatorem == (na przykładzie obiektu String)**

Operator == porównuje referencje do obiektów. Sprawdza, czy dwie zmienne odnoszą się do tego samego obiektu w pamięci.

Metoda equals() jest zdefiniowana w klasie Object i jest zwykle nadpisywana w klasach, aby porównywać zawartość obiektów. Dla obiektów String metoda equals() porównuje zawartość (wartości) tych obiektów.

8. **Po co używać @Override**

Adnotacja **@Override** w Javie jest używana do oznaczania metod, które nadpisują (implementują) metody z klas nadrzędnych (superclass) lub interfejsów (interfaces).

Plusy:

- poprawia czytelność kodu

- Zwiększa bezpieczeństwo kodu

- Adnotacja @Override pomaga kompilatorowi wykrywać błędy (np. metoda nie istnieje już w klasie bazowej)

9. **Klasy wewnętrzne, anonimowe:**

Klasa wewnętrzna to klasa, która jest zdefiniowana w obrębie innej klasy. Może mieć dostęp do wszystkich pól i metod klasy zewnętrznej, nawet tych prywatnych.

Klasa anonimowa to klasa, która nie ma nazwy i jest tworzona w miejscu, w którym jest używana. Zwykle służy do implementacji interfejsu lub nadpisania metody klasy.

10. **Czym są wyrażenia lambda, jak się je konstruuje, gdzie mogą być przydatne**

```
List<String> names = Arrays.asList("Anna", "John", "Alice", "Bob");

// Użycie wyrażenia lambda do filtrowania i sortowania
List<String> filteredNames = names.stream()
    .filter(name -> name.startsWith("A")) // filtruje imiona zaczynając
    .sorted() // sortuje
    .collect(Collectors.toList());

System.out.println(filteredNames); // Output: [Alice, Anna]
```