Wprowadzenie do Javy - materiały na lab3

Lista tablicowa `ArrayList`

`ArrayList` to jedna z najczęściej używanych implementacji interfejsu `List`. Jest to dynamicznie rozszerzalna tablica, która może zmieniać swój rozmiar w miarę dodawania i usuwania elementów. Wewnętrznie `ArrayList` używa tablicy do przechowywania elementów.

Własności:

1. **Wewnętrzna struktura danych**: Jak wspomniano wcześniej, `ArrayList` wewnętrznie używa tablicy do przechowywania elementów. Gdy lista staje się pełna i potrzeba dodać kolejny element, Java tworzy większą tablicę i kopiuje elementy starej tablicy do nowej.
2. **Rozmiar vs. Pojemność**:
   - **Rozmiar** to liczba elementów w liście.
   - **Pojemność** to liczba elementów, które lista może aktualnie przechowywać bez potrzeby alokacji większej tablicy.
3. **Zalety**:
   - **Dynamiczne rozszerzanie**: `ArrayList` może zmieniać swój rozmiar w miarę potrzeb, co czyni ją bardziej elastyczną niż zwykłe tablice.
   - **Dostęp indeksowany**: Można łatwo uzyskać dostęp do elementu na podstawie jego indeksu.
   - Możliwość przechowywania wartości `null`.
4. **Wady**:
   - **Koszt wstawiania i usuwania**: Wstawianie i usuwanie elementów (zwłaszcza w środku listy) może być kosztowne, ponieważ wymaga przesuwania innych elementów.
   - Zużywa więcej pamięci w porównaniu z `LinkedList` z powodu dodatkowej pojemności rezerwowej.

5. **Bezpieczeństwo wątkowe**: Standardowa implementacja `ArrayList` nie jest synchronizowana, co oznacza, że nie jest bezpieczna do użytku w środowiskach wielowątkowych bez odpowiedniej synchronizacji. Jeśli potrzebujesz wersji synchronizowanej, możesz użyć `Collections.synchronizedList()`.

6. **Pamięć**: Jeśli znasz ostateczny rozmiar listy, warto zainicjalizować `ArrayList` z odpowiednią pojemnością początkową, aby uniknąć wielokrotnego rozszerzania wewnętrznej tablicy.

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

// Dodawanie elementów
list.add("A");
list.add("B");
list.add("C");

// Odczytywanie elementów
String element = list.get(1);  // B

// Ustalanie rozmiaru listy
int size = list.size();  // 3

// Usuwanie elementu
list.remove(1);  // Usuwa element "B"
```

Okej, przyjrzyjmy się niektórym z najczęściej używanych metod i właściwości klasy `ArrayList` w Javie:

Najczęściej używane metody:

1. **`add(E element)`**:

   - Dodaje element na końcu listy.

```java
ArrayList<String> list = new ArrayList<>();
list.add("Element");
```

2. **`add(int index, E element)`**:

   - Wstawia określony element w określonej pozycji na liście.

```java
list.add(1, "Nowy element");
```

3. **`get(int index)`**:
    - Zwraca element z określonej pozycji.

    ```java
    String el = list.get(1);
    ```

4. **`remove(int index)`**:
    - Usuwa element z określonej pozycji.

    ```java
    list.remove(1);
    ```

5. **`remove(Object o)`**:
    - Usuwa pierwsze wystąpienie określonego elementu z listy (jeśli istnieje).

    ```java
    list.remove("Element");
    ```

6. **`size()`**:
    - Zwraca liczbę elementów na liście.

    ```java
    int rozmiar = list.size();
    ```

7. **`isEmpty()`**:
    - Sprawdza, czy lista jest pusta.

    ```java
    boolean jestPusta = list.isEmpty();
    ```

8. **`clear()`**:
    - Usuwa wszystkie elementy z listy.

    ```java
    list.clear();
    ```

9. **`contains(Object o)`**:
    - Sprawdza, czy lista zawiera określony element.

    ```java
    boolean zawiera = list.contains("Element");
    ```

10. **`indexOf(Object o)`**:

- Zwraca indeks pierwszego wystąpienia określonego elementu na liście lub -1, jeśli elementu nie ma na liście.

```java
int indeks = list.indexOf("Element");
```

11. **`set(int index, E element)`**:

- Zastępuje element na określonej pozycji.

```java
list.set(1, "Zastąpiony element");
```

12. **`toArray()`**:

- Zwraca tablicę zawierającą wszystkie elementy w liście w odpowiedniej kolejności.

```java
Object[] tablica = list.toArray();
```

Klasy takie jak `ArrayList` są generycznymi klasami kontenerowymi, które przechowują obiekty, a nie typy proste. Dlatego nie możemy bezpośrednio użyć `int` lub `double` jako typu elementu dla `ArrayList`.

Aby rozwiązać ten problem, Java dostarcza klasy opakowujące (wrapper classes) dla wszystkich typów prostych. Dla `int` mamy `Integer`, dla `double` mamy `Double` itd.

Dzięki autoboxingowi (automatyczne konwersje między typami prostymi a ich klasami opakowującymi) korzystanie z tych klas opakowujących jest stosunkowo proste i wygodne.

1. **Dla `int`**:

```java
ArrayList<Integer> listaInt = new ArrayList<>();
listaInt.add(1);  // Autoboxing konwertuje 'int' na 'Integer'
int liczba = listaInt.get(0);  // Autounboxing konwertuje 'Integer' na 'int'
```

2. **Dla `double`**:

```java
ArrayList<Double> listaDouble = new ArrayList<>();
listaDouble.add(1.5);  // Autoboxing konwertuje 'double' na 'Double'
double liczbaDouble = listaDouble.get(0);  // Autounboxing konwertuje 'Double' na 'do
```

Napisy, łańcuchy znaków

Inicjalizacja:

- Można inicjalizować napisy na różne sposoby:

```
String s1 = "Hello";
String s2 = new String("Hello");
```

Niezmienność (Immutability):

- Obiekty klasy `String` są niezmienne. Oznacza to, że raz utworzony łańcuch znaków nie może być zmieniony. Wszelkie operacje modyfikujące zawartość łańcucha (np. dodawanie, usuwanie znaków) skutkują stworzeniem nowego obiektu `String`.

Konkatenacja:

- Napisy można łączyć za pomocą operatora `+`:

```
String s1 = "Hello";
String s2 = "World";
String s3 = s1 + " " + s2;  // "Hello World"
String s4 = s1 + 4;  // "Hello4"
```

Porównywanie:

- Aby porównać zawartość dwóch napisów, powinno się używać metody `equals()` zamiast operatora `==`:

```
String s1 = "Hello";
String s2 = new String("Hello");
boolean isEqual = s1.equals(s2);  // true
```

Pamięć:

- Ze względu na optymalizację pamięci, Java posiada tzw. "pulę napisów" (String Pool). Dwa napisy o tej samej zawartości często będą wskazywać na ten sam obszar pamięci, jeśli zostały zainicjowane bez użycia słowa kluczowego `new`.

Zmienne łańcuchowe:

- Jeśli potrzebujesz modyfikowalnego łańcucha, Java oferuje klasy takie jak `StringBuilder` i `StringBuffer`. Są one szczególnie przydatne w sytuacjach, gdzie zachodzi wiele modyfikacji łańcucha, ponieważ operują one w miejscu (in-place) i są zwykle szybsze niż tworzenie wielu obiektów `String`.

Różnice między napisem pustym a `null`

Wartość:

- **Łańcuch pusty (`""`)**: To faktyczny obiekt klasy `String`, który ma wartość, ale ta wartość jest pusta. Inaczej mówiąc, jest to łańcuch, który nie zawiera żadnych znaków.
- **`null`**: To specjalna wartość, która oznacza, że zmienna nie wskazuje na żaden obiekt. Dla zmiennej typu `String`, jeśli jest ona ustawiona na `null`, nie wskazuje ona na żaden łańcuch (pusty czy inny).

Długość:

- **Łańcuch pusty (`""`)**: Jego długość wynosi 0. Można to sprawdzić używając metody `length()`: `"".length()` zwróci 0.
- **`null`**: Zmienna o wartości `null` nie posiada metod ani atrybutów. Próba wywołania metody, np. `null.length()`, spowoduje wyjątek `NullPointerException`.

Operacje:

- **Łańcuch pusty (`""`)**: Możesz wykonywać na nim różne operacje, takie jak konkatenacja czy wywoływanie innych metod klasy `String`.
- **`null`**: Nie możesz wykonywać na nim żadnych operacji. Każda próba dostępu do metody lub atrybutu na zmiennej o wartości `null` spowoduje `NullPointerException`.

Porównywanie:

- Możesz porównać zarówno łańcuch pusty, jak i `null` z innymi łańcuchami. Ale musisz być ostrożny z `null`, ponieważ:

```java
String s1 = "";
String s2 = null;

System.out.println(s1.equals(""));  // true
System.out.println(s2.equals(null));  // Wyjątek: NullPointerException
```

Zastosowania:

- **Łańcuch pusty (`""`)**: Często używany do inicjowania łańcuchów bez konkretnej wartości lub do wskazania, że łańcuch powinien być "czysty" lub "bez wartości", ale nadal istnieje.
- **`null`**: Wskazuje, że zmienna nie odnosi się do żadnego obiektu. Jest używane w wielu przypadkach, np. gdy wartość nie jest jeszcze znana lub nie została ustawiona.

Metody z API

Dokumentacja

`length()`:

- Zwraca liczbę znaków w łańcuchu.

```java
String s = "Hello";
int len = s.length();  // 5
```

`charAt(int index)`:

- Zwraca znak na określonej pozycji w łańcuchu.

```java
char ch = s.charAt(1);  // 'e'
```

`substring(int beginIndex, int endIndex)`:

- Zwraca nowy łańcuch zawierający znaki z oryginalnego łańcucha od `beginIndex` (włącznie) do `endIndex` (wyłącznie).

```java
String sub = s.substring(1, 4);  // "ell"
```

`indexOf(String str)` i **`lastIndexOf(String str)`**:

- Zwracają indeks pierwszego/ostatniego wystąpienia podciągu w łańcuchu. Jeśli podciąg nie jest znaleziony, zwraca -1.

```java
int first = s.indexOf("l");  // 2
int last = s.lastIndexOf("l");  // 3
```

`equals(Object obj)`:

- Porównuje zawartość tego łańcucha z zawartością innego obiektu. Zwraca `true`, jeśli są równe.

```java
boolean isEqual = s.equals("Hello");  // true
```

`equalsIgnoreCase(String anotherString)`:

- Porównuje łańcuchy bez uwzględniania wielkości liter.

```java
boolean isEqual = s.equalsIgnoreCase("HELLO");  // true
```

startsWith(String prefix) i **endsWith(String suffix)**:

- Sprawdzają, czy łańcuch zaczyna się lub kończy danym ciągiem znaków.

```
boolean starts = s.startsWith("He");  // true
boolean ends = s.endsWith("lo");  // true
```

replace(char oldChar, char newChar) lub **replace(CharSequence target, CharSequence replacement)**:

- Zwraca nowy łańcuch, w którym wszystkie wystąpienia `oldChar` lub `target` są zastąpione przez `newChar` lub `replacement`.

```
String replaced = s.replace("l", "w");  // "Hewwo"
```

trim():

- Zwraca kopię łańcucha z usuniętymi białymi znakami na początku i końcu.

```
String trimmed = "  Hello  ".trim();  // "Hello"
```

toLowerCase() i **toUpperCase()**:

- Zmieniają wielkość liter w łańcuchu.

```
String lower = s.toLowerCase();  // "hello"
String upper = s.toUpperCase();  // "HELLO"
```

split(String regex):

- Dzieli łańcuch według podanego wyrażenia regularnego.

```
String[] parts = "Hello-World".split("-");  // ["Hello", "World"]
```

isEmpty():

- Sprawdza, czy łańcuch jest pusty (długość wynosi 0).

```
boolean empty = "".isEmpty();  // true
```

valueOf() (statyczna metoda):

- Konwertuje różne typy danych (np. int, char) na łańcuchy.

```java
String str = String.valueOf(12345);  // "12345"
```

Różnice w podejściu do napisów

`String`:

- **Niemutowalność**: Obiekty `String` są niemutowalne, co oznacza, że po ich utworzeniu nie można ich zmienić. Każda operacja, która wydaje się modyfikować łańcuch (np. konkatenacja), faktycznie tworzy nowy obiekt `String`.
- **Wydajność**: Ze względu na niemutowalność operacje modyfikujące mogą być mniej wydajne (szczególnie w długich pętlach), ponieważ za każdym razem tworzony jest nowy obiekt.
- **Bezpieczeństwo wątków**: Jest bezpieczny w wielowątkowości ze względu na niemutowalność.

`StringBuilder`:

- **Mutowalność**: Obiekty `StringBuilder` są mutowalne. Można dodawać, usuwać i modyfikować zawartość obiektu bez tworzenia nowych obiektów.
- **Wydajność**: Jest zazwyczaj bardziej wydajny niż `String` w operacjach modyfikujących, szczególnie w intensywnych operacjach, takich jak budowanie łańcuchów w pętlach.
- **Bezpieczeństwo wątków**: Nie jest synchronizowany, co oznacza, że może nie być bezpieczny w środowiskach wielowątkowych. Jeśli bezpieczeństwo wątków nie jest wymagane, `StringBuilder` jest zwykle lepszym wyborem niż `StringBuffer`.

Często używane metody:

append():

- Dodaje wartość do końca obecnej zawartości `StringBuilder`.
- Jest przeciążona, by obsłużyć różne typy danych: `String`, `char`, `int`, `long`, `float`, `double` itp.
- Przykład:

  ```java
  StringBuilder sb = new StringBuilder("Hello");
  sb.append(" World");
  ```

insert():

- Wstawia wartość w określonej pozycji w `StringBuilder`.
- Przykład:

```java
StringBuilder sb = new StringBuilder("Hello World");
sb.insert(6, "Java ");
```

delete() i deleteCharAt():

- `delete(int start, int end)` usuwa podciąg znaków od indeksu `start` do indeksu `end - 1`.
- `deleteCharAt(int index)` usuwa znak w określonym indeksie.
- Przykład:

```java
StringBuilder sb = new StringBuilder("Hello Java World");
sb.delete(6, 11); // Usuwa słowo "Java "
```

replace():

- Zamienia podciąg znaków w określonym zakresie na inny ciąg znaków.
- Przykład:

```java
StringBuilder sb = new StringBuilder("Hello Java");
sb.replace(6, 10, "World");
```

toString():

- Konwertuje `StringBuilder` na `String`.
- Przykład:

```java
StringBuilder sb = new StringBuilder("Hello World");
String str = sb.toString();
```

length():

- Zwraca liczbę znaków w `StringBuilder`.
- Przykład:

```java
StringBuilder sb = new StringBuilder("Hello");
int len = sb.length();  // len = 5
```

setLength():

- Ustawia długość `StringBuilder`. Jeśli nowa długość jest krótsza niż obecna, ciąg zostanie obcięty. Jeśli jest dłuższy, dodane zostaną znaki o wartości `\u0000`.
- Przykład:

```
StringBuilder sb = new StringBuilder("Hello");
sb.setLength(3);  // sb = "Hel"
```

charAt(), setCharAt():

- `charAt(int index)` zwraca znak w określonym indeksie.
- `setCharAt(int index, char ch)` ustawia znak w określonym indeksie.
- Przykład:

```
StringBuilder sb = new StringBuilder("Hello");
char ch = sb.charAt(1);  // ch = 'e'
sb.setCharAt(1, 'a');  // sb = "Hallo"
```

reverse():

- Odwraca kolejność znaków w `StringBuilder`.
- Przykład:

```
StringBuilder sb = new StringBuilder("Hello");
sb.reverse();  // sb = "olleH"
```

`StringBuffer`:

- **Mutowalność**: Podobnie jak `StringBuilder`, obiekty `StringBuffer` są mutowalne.
- **Wydajność**: Generalnie podobna wydajność do `StringBuilder`, ale operacje są synchronizowane.
- **Bezpieczeństwo wątków**: Jest synchronizowany, co oznacza, że jest bezpieczny w środowiskach wielowątkowych. Jeśli potrzebujesz mutowalnego łańcucha w środowisku wielowątkowym, `StringBuffer` może być odpowiednim wyborem.

Metody dla `StringBuffer` są w większości analogiczne jak dla `StringBuilder`.

Funkcje znakowe

https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/Character.html