# QA Automation

**Олександр Подоляко**

PRESS START

robot_dreams

future
thinking
school

**Заняття 9.**
Множинні типи даних

# Масиви

Будь-який тип даних може бути обернутий в масив

```
int[]intsArray = new int[5];
String[] stringsArray = new String[5];
BigInteger[] bigIntegersArray = new BigInteger[10];
```

**Переваги**

+ Базова структура Java
+ Оптимальне використання CPU*
+ Швидкість звернення
+ Оболонка над примітивними типами

**Недоліки**

- Фіксована довжина
- Сувора типізація
- Неоптимальне використання пам'яті*
- Додавання/видалення не підтримується

```
int[] intArray = new int[] {10,15,20};  // неявно вказана довжина масиву 3
```

Для роботи з масивами є
`java.util.Arrays`

# Багатовимірні масиви

```
int[][] intsMatrix = new int[5][9];
int[][][] intsCube = new int[5][9][10];
```

Усі вкладені масиви мають однакову довжину

```
int[][]intsMatrix = new int[][] {{1,2,3}, {4,5}, {1}};
```

Усі вкладені масиви мають власну довжину

Вкладені масиви є масивами з усіма можливостями та обмеженнями масивів.

# Collections framework

- List
- Set
- Queue
  - Dequeue (double-ended-queue)

# ArrayList

Пряма репрезентація масиву

```
ArrayList<String> myList = new ArrayList<>();
```


Початкова місткість — 10.

Місткість збільшується за потребою за формулою:

```
newCapacity = oldCapacity * 1.5 + 1;
```

Можна вказати початкову місткість:

```
List<String> myList = new ArrayList<>(100);
```

Можна примусово збільшити місткість:

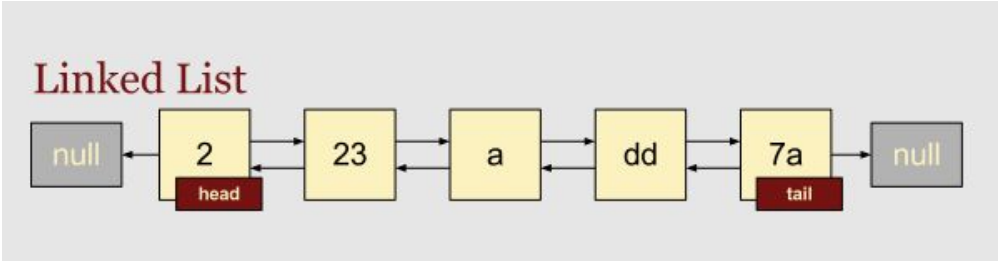
```
myList.ensureCapacity(100);
```

Можна примусово зменшити місткість до поточного розміру:

```
myList.trimToSize();
```

# LinkedList

Набір контейнерів, кожен з яких пов'язаний з попереднім та наступним.




LinkedList зберігає всередині тільки розмір та посилання на перший і останній елементи, подальше зв'язування відбувається методами елементів.

LinkedList має всі особливості ArrayList плюс набір додаткових методів add/remove/get … First/Last.

# Vector

Аналогічний ArrayList з деякими відмінностями:

- збільшення місткості відбувається за формулою

```
newCapacity = oldCapacity * 2
```

- можна встановити інкремент місткості

```
Vector<String> v = new Vector<>(x,y);
```

де **x** – початкова місткість, а **y** – інкремент місткості, тоді збільшення місткості відбувається за формулою `newCapacity = oldCapacity + y`

- всі методи об'єкта є синхронізованими

# Map

**Map (mapping)** — відношення одного значення до іншого (ключ-значення), одного ключа може відповідати тільки одне значення.

**Hash** — репрезентація чогось (об'єкта) у скороченому вигляді.

Властивості Hash:

- ➔ унікальність — для одного значення може існувати лише один hash
- ➔ незворотність — за hash неможливо відновити значення
- ➔ швидкість — hash повинен обчислюватися швидко
- ➔ два різні значення можуть мати однаковий hash

# HashMap

Структура даних, що зберігає відносини ключ-значення, використовує hash для прискорення пошуку елементів за ключем.

```
HashMap<Integer, String> myMap =  newHashMap<>();
myMap.put(10,"Ten");
myMap.get(10); → повертає 10
```

Потребує, щоб клас ключа реалізовував методи hashcode() та equals().

Зберігає дані у структурі, яка подібна на масив, кожним елементом якого є контейнер (bucket), в якому може зберігатися безліч пар ключ-значення.

# HashMap

При додаванні пари використовується hashcode метод ключа, щоб визначити в який контейнер покласти пару (у контейнері вже може бути інша пара).

При пошуку за ключем використовується hashcode метод ключа, щоб визначити, в якому контейнері шукати пару, якщо в контейнері кілька пар, використовується метод **equals** ключа для визначення шуканої пари.

Не гарантує впорядкованість зберігання даних.

# LinkedHashMap

Те ж саме, що і HashMap, але кожна пара ключ-значення обгорнута у внутрішній об'єкт Entry, який додатково зберігає посилання на попередній та наступний елементи, так забезпечується збереження порядку додавання елементів.

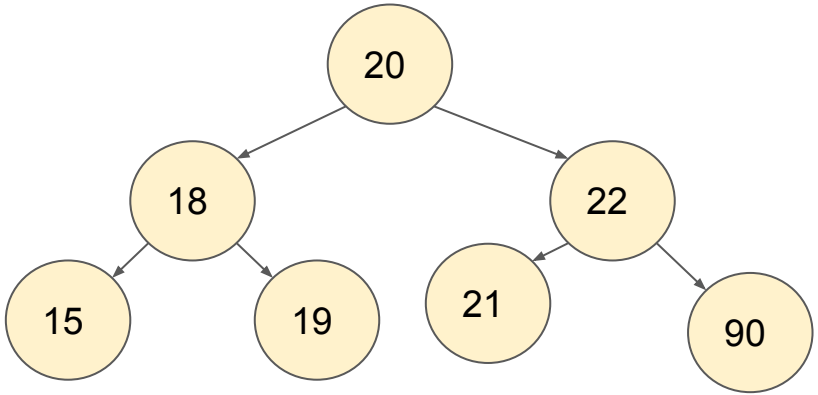
# TreeMap

Структура даних, що зберігає відносини ключ-значення, що використовує деревоподібну структуру для прискорення пошуку елементів ключа.

Не використовує hashcode, використовує equals, вимагає, щоб ключ був порівнюваним (або примітивний тип, або реалізує інтерфейс Comparable). Можна також використовувати кастомний клас Comparator.

Має механізм балансування, якщо одна з гілок стає більшою за іншу.

На відміну від HashMap, гарантує упорядкованість ключів.



# HashSet

Несортований набір унікальних елементів:

- не надає жодної впорядкованості елементів
- як внутрішнє сховище використовує HashMap
- 
```
HashSet<String> mySet = newHashSet<>();
mySet.add("Foo");
mySet.contains("Foo");
mySet.remove("Foo");
mySet.addAll(myList) ->
```
найпростіший спосіб видалити дублікати з List

# LinkedHashSet

Упорядкований у порядку додавання набір унікальних елементів:

- ➔ є гібридом HashSet та LinkedList
- ➔ як внутрішнє сховище використовує LinkedHashMap

# TreeSet

Сортований за значенням набір унікальних елементів:

- ➔ як внутрішнє сховище використовує TreeMap
- ➔ як і TreeMap, не використовує hashcode, використовує equals; вимагає, щоб ключ був порівнюваним

Попри те, що Set і Map не пов'язані прямо, всі основні Set використовують Map для внутрішнього зберігання даних, оскільки Map забезпечує унікальність ключів.

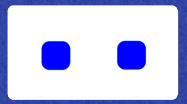
Оскільки прямий зв'язок між Set і Map немає, то немає ніякої гарантії, що абсолютно всі Set будуть використовувати Map.

# Питання

???

???

# robot_dreams



future
thinking
school

robot_dreams