Lesson 8

22.05.2023

```
public class Ex1 extends Foo {
    public static String sign() {
        return "fa";
    }

    public static void main(String[] args) {
        Ex1 ex = new Ex1();
        Foo foo = new Ex1();

        System.out.println(ex.sign() + " " + foo.sign());
    }
}

class Foo {
    public static String sign() {
        return "la";
}
```

```
public class RedWood extends Tree {
    public static void main(String[] args) {
        new RedWood().go();
    }

    void go() {
        run(new Tree(), new RedWood());
        run((Tree) new RedWood(), (RedWood) new Tree());
    }

    void run(Tree t1, RedWood r1) {
        RedWood r2 = (RedWood) t1;
        Tree t2 = (Tree) r1;

    }
}

class Tree {
}
```

```
public class Ex3 extends Electronic implements Gadget {
    public void doSomething() {
        System.out.println("serf internet ...");
    }

    public static void main(String[] args) {
        new Ex3().doSomething();
        new Ex3().getPower();
    }
}
abstract class Electronic{
    void getPower(){
        System.out.println("plug in ...");
    }
}
interface Gadget{
    void doSomething();
}
```

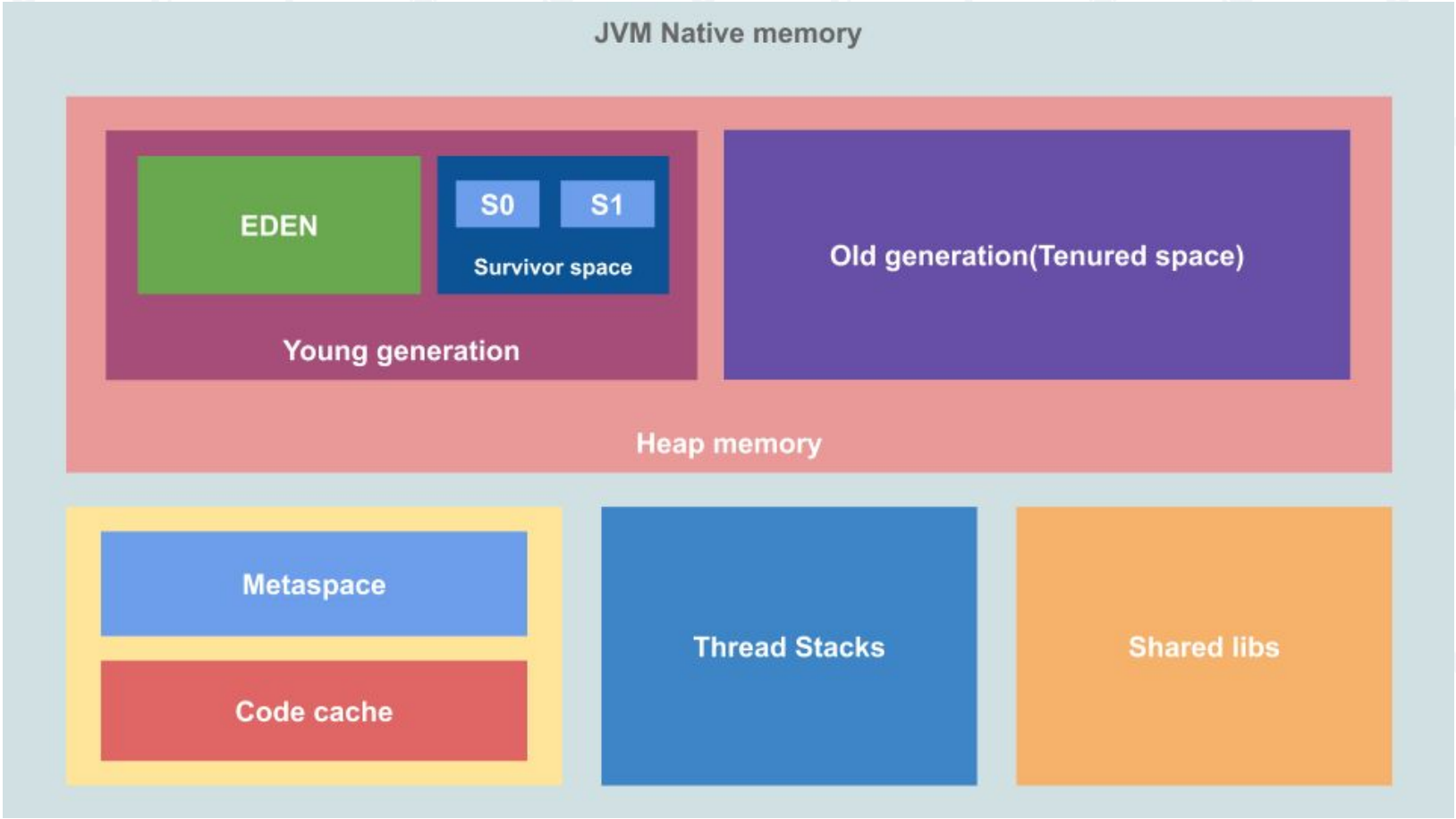
```
public class Ex4 {
    public static void main(String[] args) {
        int numFish = 4;
        String fishType = "tuna";
        String anotherFish = numFish + 1;
        System.out.println(anotherFish + " " + fishType);
        System.out.println(numFish + " " + 1);
    }
}
```

```
public class Ex5 {
    private final static String RESULT ="2cfalse";
    public static void main(String[] args) {
        String a = "";
        a += 2;
        a += 'c';
        a += false;
        if (a == RESULT) System.out.println("==");
        if (a.equals(RESULT)) System.out.println("equals");
    }
}
```

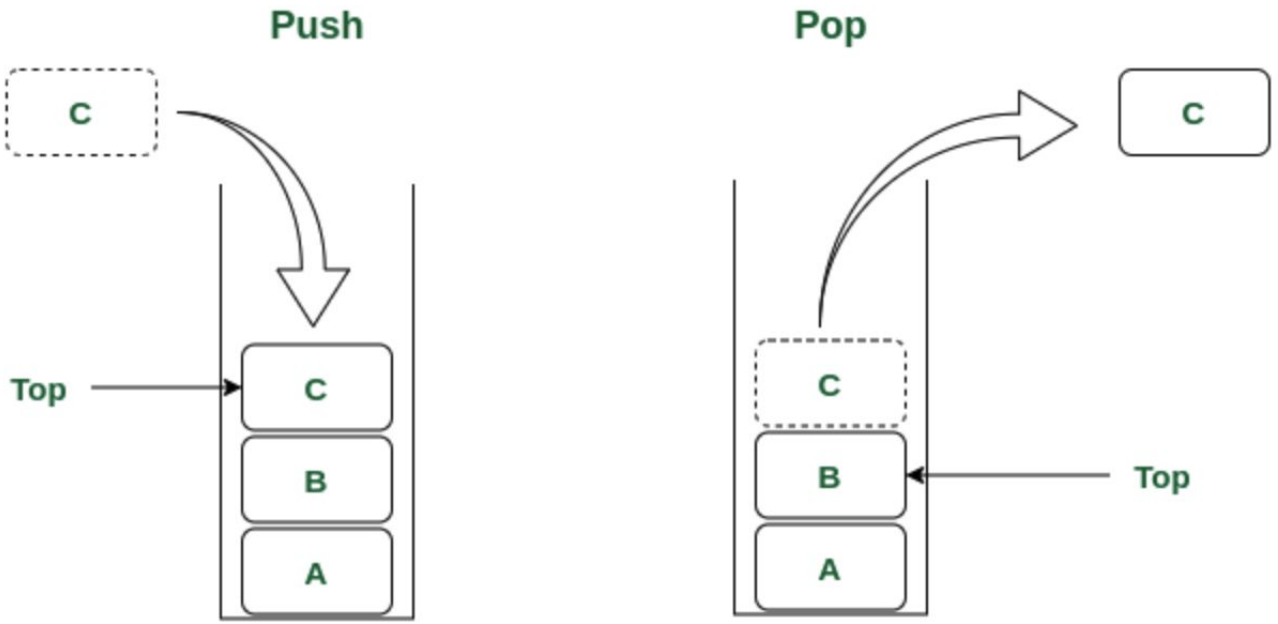
```
public class Ex6 {
    final static short x = 2;
    public static int y = 0;

    public static void main(String[] args) {
        for (int z =0; z < 3; z++){
            switch (z){
                case x: System.out.print("0 ");
                case x - 1: System.out.print("1 ");
                case x - 2: System.out.print("2 ");
            }
        }
    }
}
```

Java memory model



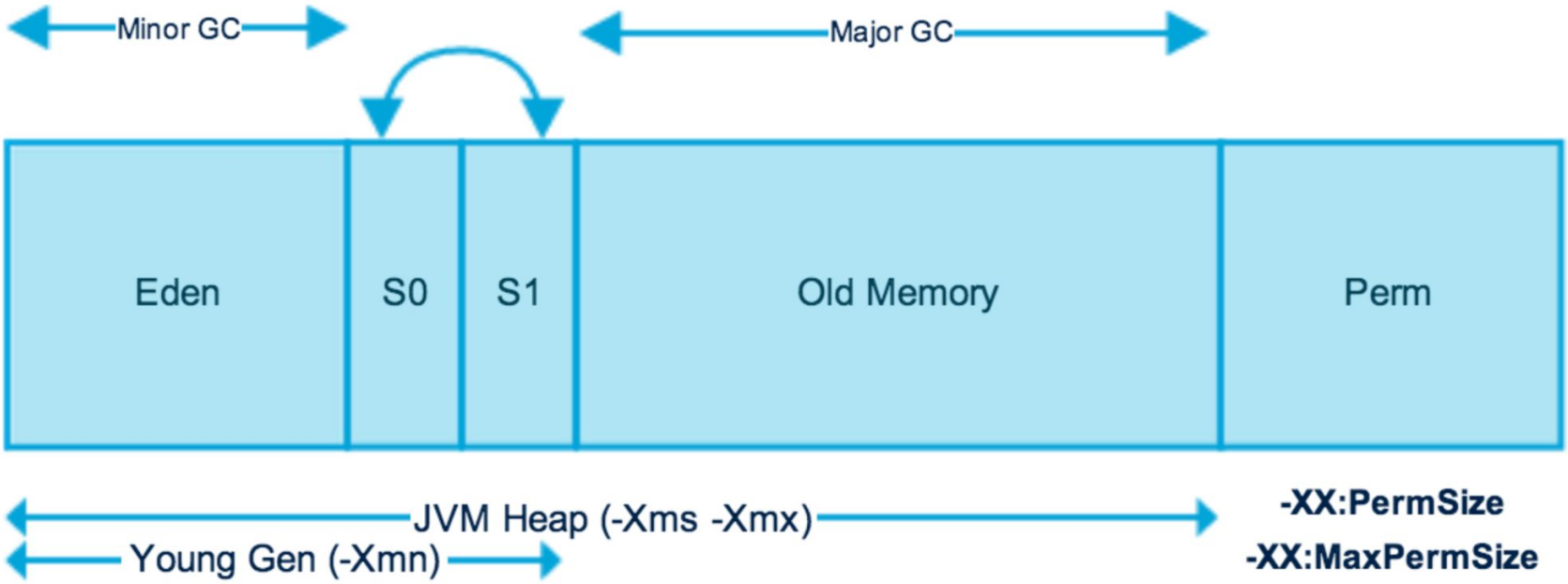
Stack



Основні особливості стеку

- Він заповнюється та звільняється в міру виклику та завершення нових методів
- Змінні в стеку існують до тих пір, поки виконується метод якому вони були створені
- Якщо пам'ять стека буде заповнена, Java кине виняток java.lang.StackOverFlowError
- Доступ до цієї області пам'яті здійснюється швидше, ніж до купи
- Є потокобезпечним, оскільки для кожного потоку створюється свій окремий стек

Heap

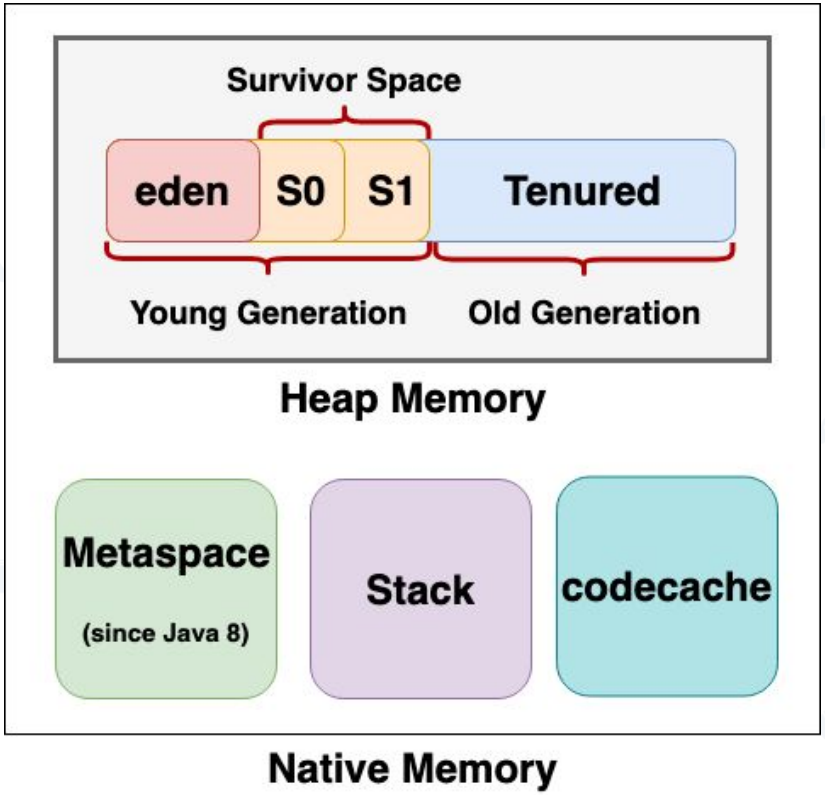


Ця область пам'яті розбита на кілька дрібніших частин, так званими поколіннями:

Young Generation — область, де розміщуються нещодавно створені об'єкти. Коли вона заповнюється, відбувається швидке складання сміття

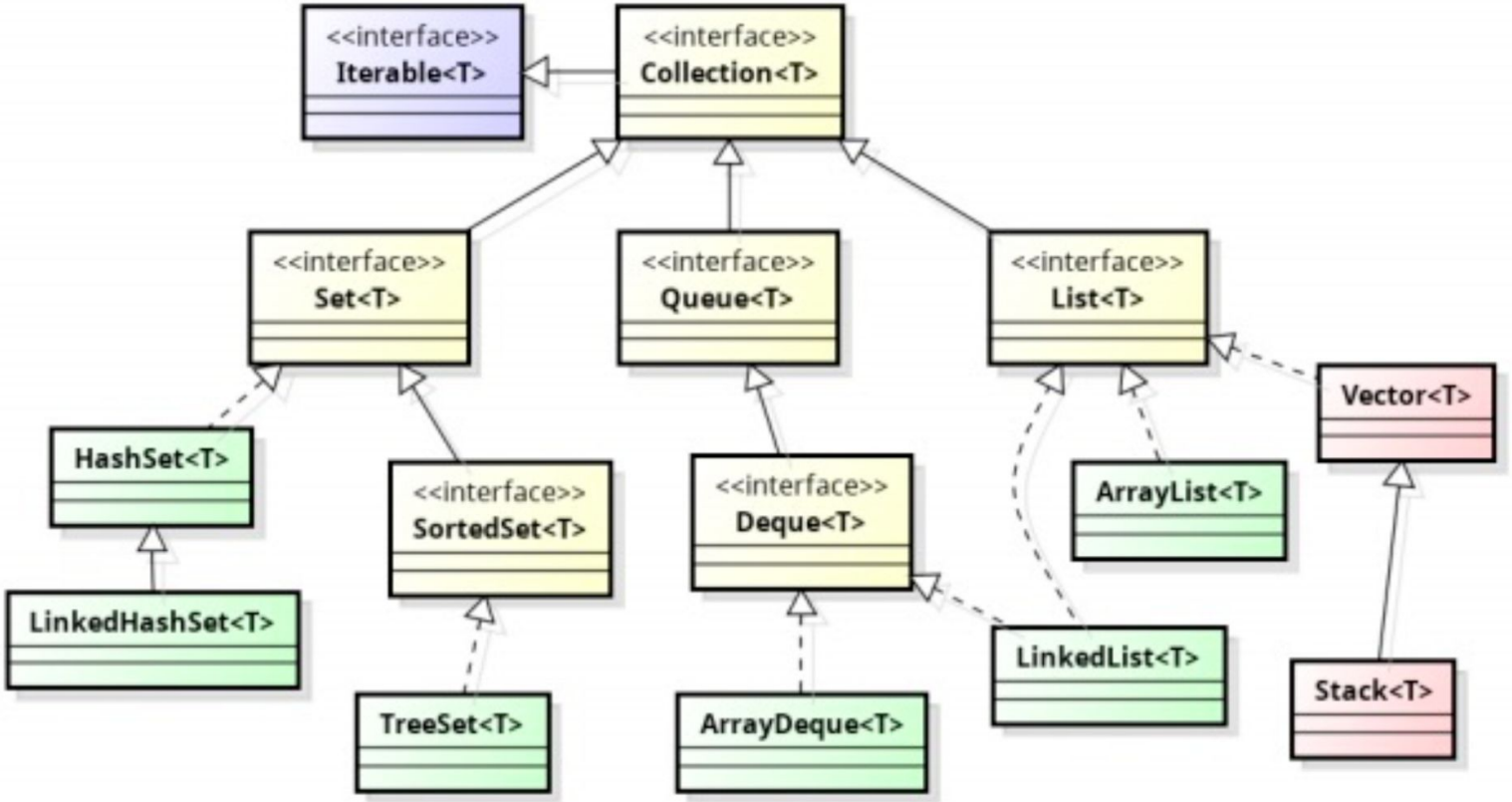
Old (Tenured) Generation - тут зберігаються довгоживучі об'єкти. Коли об'єкти з Young Generation досягають певного порога "віку", вони переміщуються до Old Generation

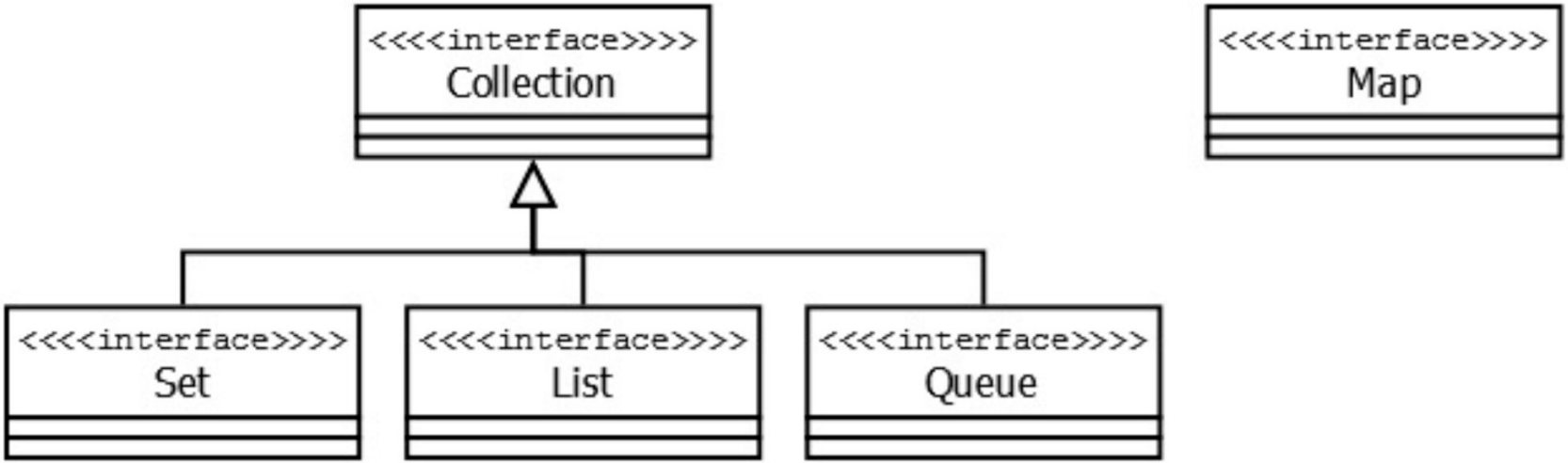
Metaspace



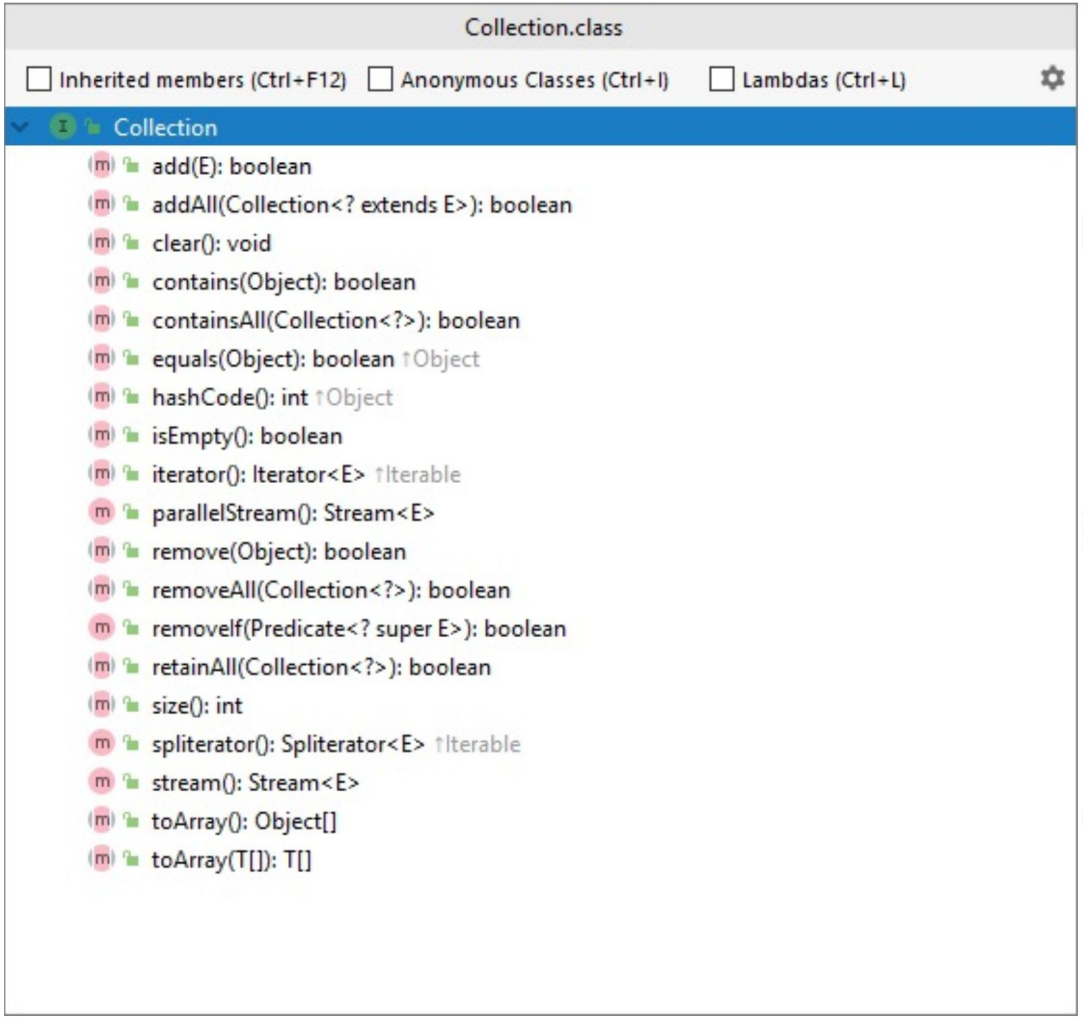
У MetaSpace віртуальна машина зберігає метадані завантажених класів. Також тут знаходяться весь статичний вміст програми, змінні примітивних типів та посилання на статичні об'єкти.

Java Collections Framework



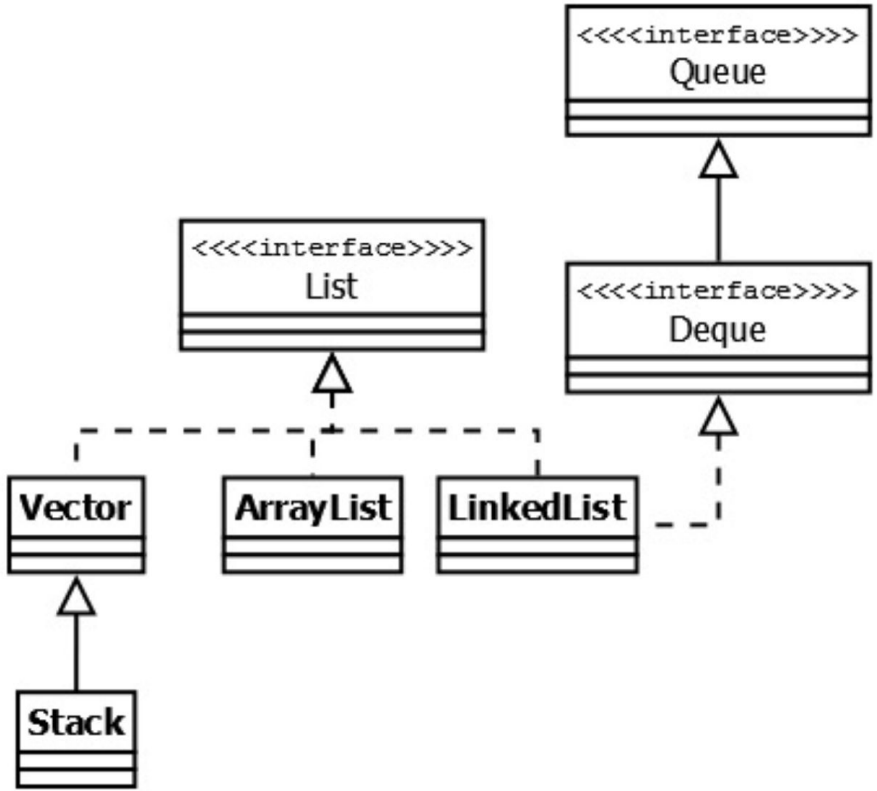


Collection - проста послідовність значень
Map – набір пар “ключ значення”


Collection.class
Inherited members (Ctrl+F12)
Anonymous Classes (Ctrl+I)
Lambdas (Ctrl+L)
Collection
add(E): boolean
addAll(Collection<? extends E>): boolean
clear(): void
contains(Object): boolean
containsAll(Collection<?>): boolean
equals(Object): boolean ↑Object
hashCode(): int ↑Object
isEmpty(): boolean
iterator(): Iterator<E> ↑Iterable
parallelStream(): Stream<E>
remove(Object): boolean
removeAll(Collection<?>): boolean
removeIf(Predicate<? super E>): boolean
retainAll(Collection<?>): boolean
size(): int
spliterator(): Spliterator<E> ↑Iterable
stream(): Stream<E>
toArray(): Object[]
toArray(T[]): T[]

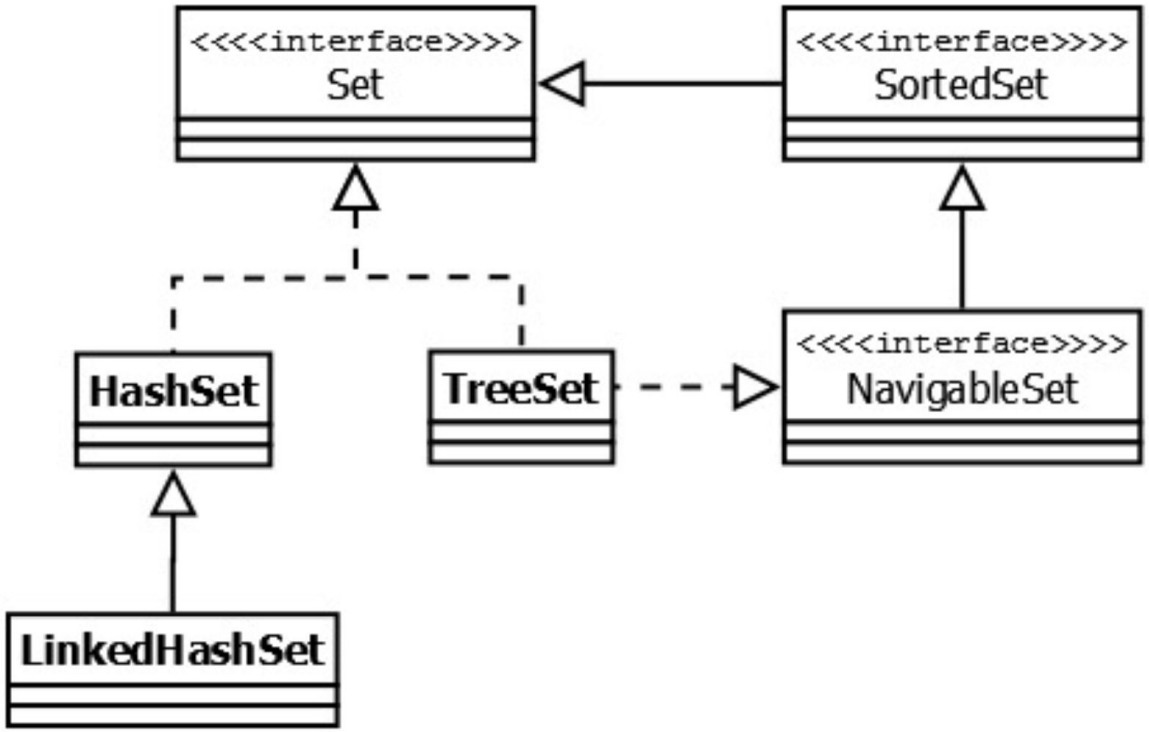
Інтерфейс List

Реалізації цього інтерфейсу є упорядковані колекції. Крім того, розробнику надається можливість доступу до елементів колекції за індексом та за значенням



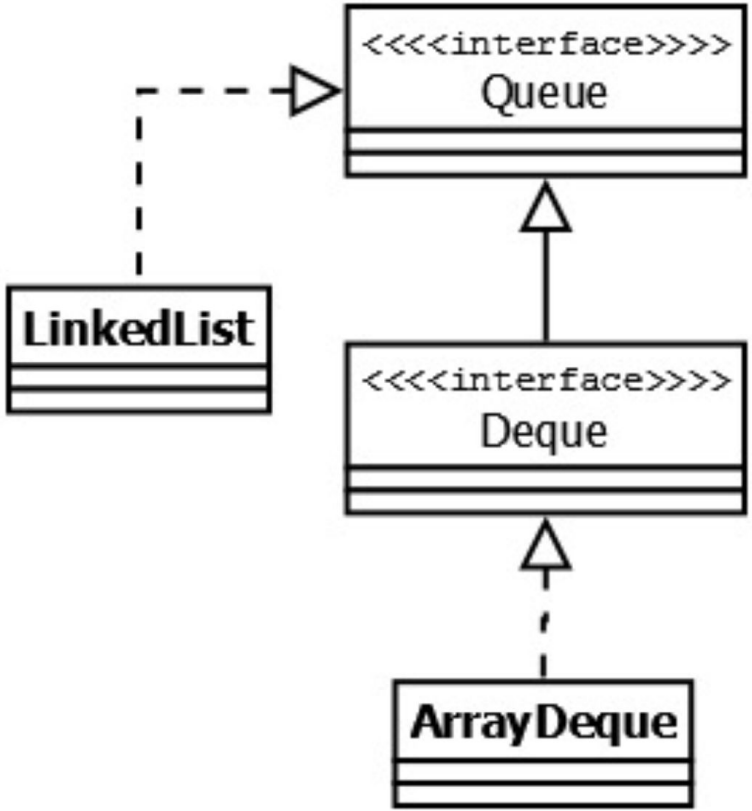
Інтерфейс Set

Являє собою невпорядковану колекцію, яка не може містити дані, що дублюються. Є програмною моделлю математичного поняття «множина».



Інтерфейс Queue

Цей інтерфейс описує колекції з певним способом вставки та вилучення елементів, а саме черги FIFO (first-in-first-out).



Складність алгоритмів

Складність алгоритму – це кількісна характеристика, що відображає споживані алгоритмом ресурси під час виконання.

1. **Логічна складність —** кількість людино-місяців, витрачених на створення алгоритму.
2. **Статична складність —** довжина опису алгоритмів (кількість операторів).
3. **Часова складність —** час виконання алгоритму.
4. **Ємнісна складність —** кількість умовних одиниць пам'яті, необхідних для роботи алгоритму.

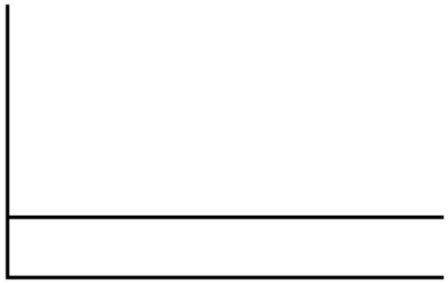
O(1)

means **constant complexity**

No matter the input size, complexity remains the same

e.g. accessing element at index from an array

```
let a = [1, 2, 3, 4, 5];
console.log(a[1]);
```



O(1) = 1
O(2) = 1
O(3) = 1
O(4) = 1
.....
O(n) = 1

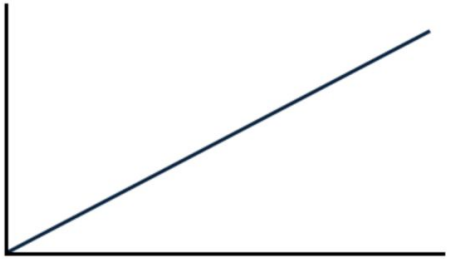
O(n)

means **linear complexity**

Complexity grows linearly over time - higher the number of inputs, higher the complexity.

e.g. looping over all the items of an array

```
for (let i = 0; i <= n; i++) {
   // do something
}
```



O(1) = 1
O(2) = 2
O(3) = 3
O(4) = 4
.....
O(n) = n

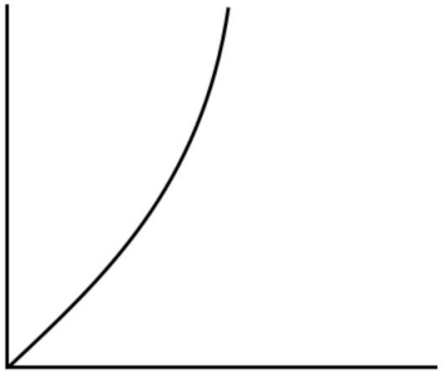
$O(n^2)$

means **quadratic complexity**

Complexity squares the number of inputs

e.g. loop within a loop

```
for (let i = 0; i <= n; i++) {
   for (let y = 0; y <= n; y++) {
      // do something
   }
}
```



$O(1) = 1$
$O(2) = 4$
$O(3) = 9$
$O(4) = 16$
.....
$O(n) = n^2$

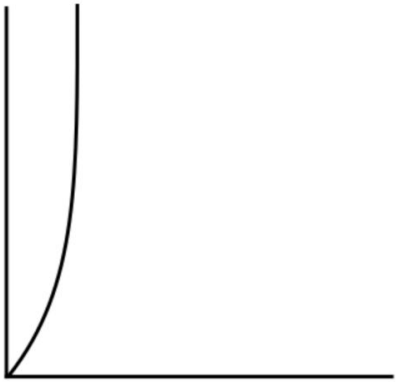
$O(2^n)$

means **exponential complexity**

complexity doubles with each addition to the input dataset

e.g. looping over all possible combinations of an array

```
function fibonacci(n) {
    if (n <= 1)  return n;
    return fibonacci(n - 2) + fibonacci(n-1);
}
```



$O(1) = 1$
$O(2) = 4$
$O(3) = 8$
$O(4) = 16$
$O(5) = 32$
$O(6) = 64$
.....
$O(n) = 2^n$

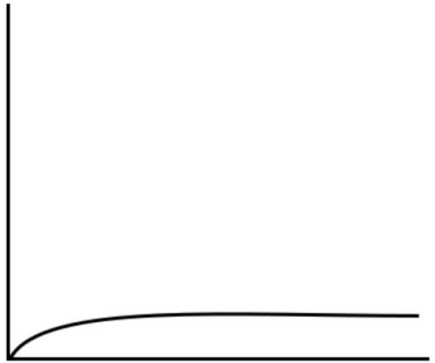
O(log n)

means **logarithmic complexity**

complexity goes up linearly while the input goes up exponentially

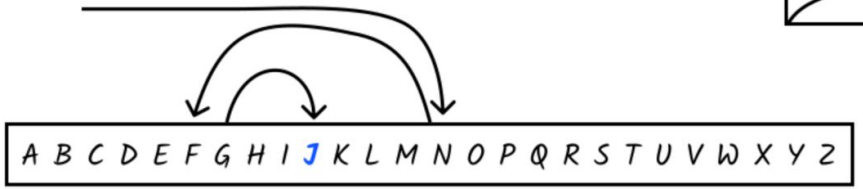
e.g. here is the example log 2

```
for (let i = 1; i <= n; i = i * 2) {
  console.log("hello");
}
```



O(10) = 1
O(200) = 2
O(300) = 3
O(400) = 4
O(500) = 5
O(600) = 6
.....
O(n) = log n

another famous example is binary search



	Временная сложность							
	Среднее				*Худшее*			
	Индекс	Поиск	Вставка	Удаление	Индекс	Поиск	Вставка	Удаление
ArrayList	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
Vector	O(1)	O(n)	O(n)	O(n)	O(1)	O(n)	O(n)	O(n)
LinkedList	O(n)	O(n)	O(1)	O(1)	O(n)	O(n)	O(1)	O(1)
Hashtable	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
HashMap	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
LinkedHashMap	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
TreeMap	n/a	O(log(n))	O(log(n))	O(log(n))	n/a	O(log(n))	O(log(n))	O(log(n))
HashSet	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
LinkedHashSet	n/a	O(1)	O(1)	O(1)	n/a	O(n)	O(n)	O(n)
TreeSet	n/a	O(log(n))	O(log(n))	O(log(n))	n/a	O(log(n))	O(log(n))	O(log(n))