Lesson 8

20.01.2022

```
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();
}
```

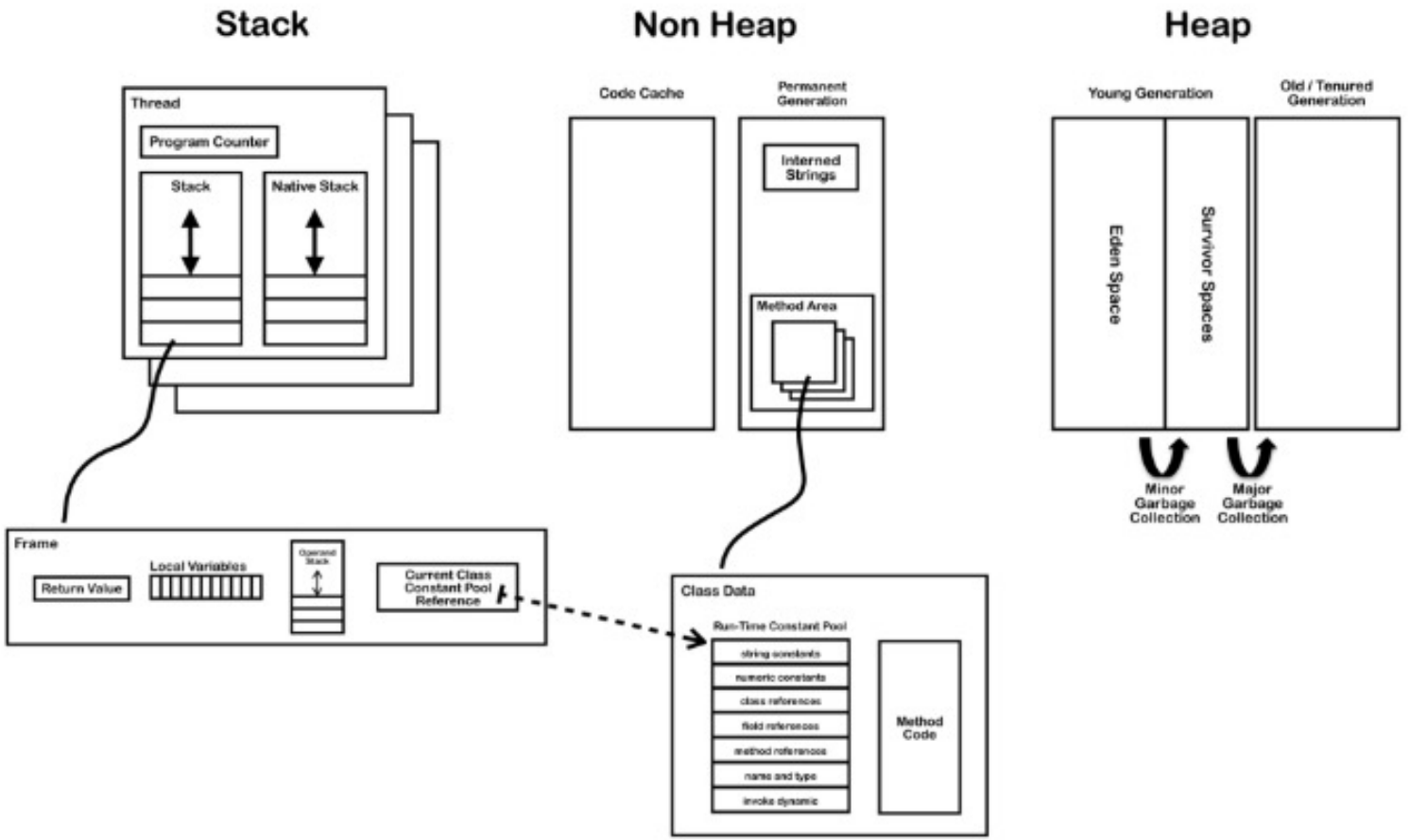
```
public class Ex4 {
    public static void main(String[] args) {
        int numFish = 4;
        String fishType = "tuna";
        String anotherFish = numFish + 1;
        System.out.println(anotherFish + " " + fishType);
        System.out.println(numFish + " " + 1);
    }
}
```

```
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");
    }
}
```

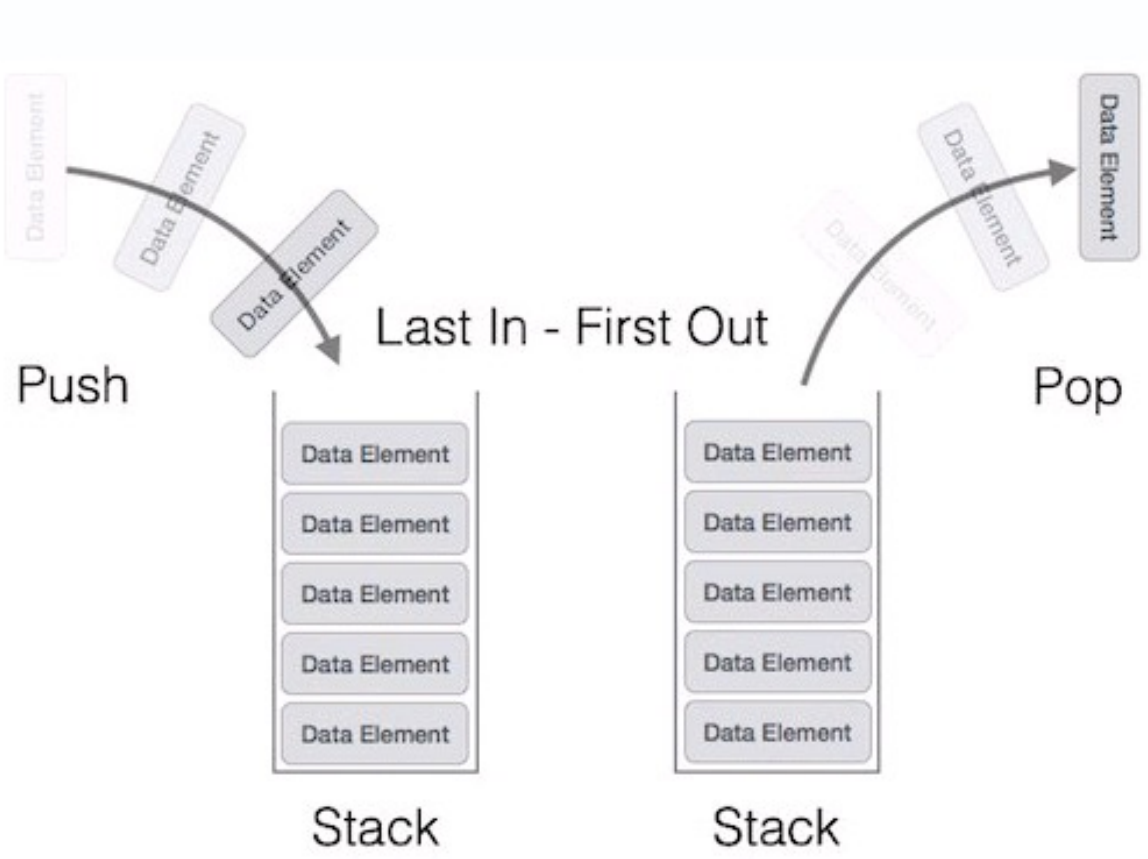
```
public class Ex6 {
    final static short x = 2;
    public static int y = 0;

    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 модель памяти



Stack


Data Element
Data Element
Data Element
Last In - First Out
Push
Pop
Data Element
Data Element
Data Element
Data Element
Data Element
Data Element
Data Element
Data Element
Data Element
Data Element
Stack
Stack

Основные особенности стека

✓ Он заполняется и освобождается по мере вызова и завершения новых методов

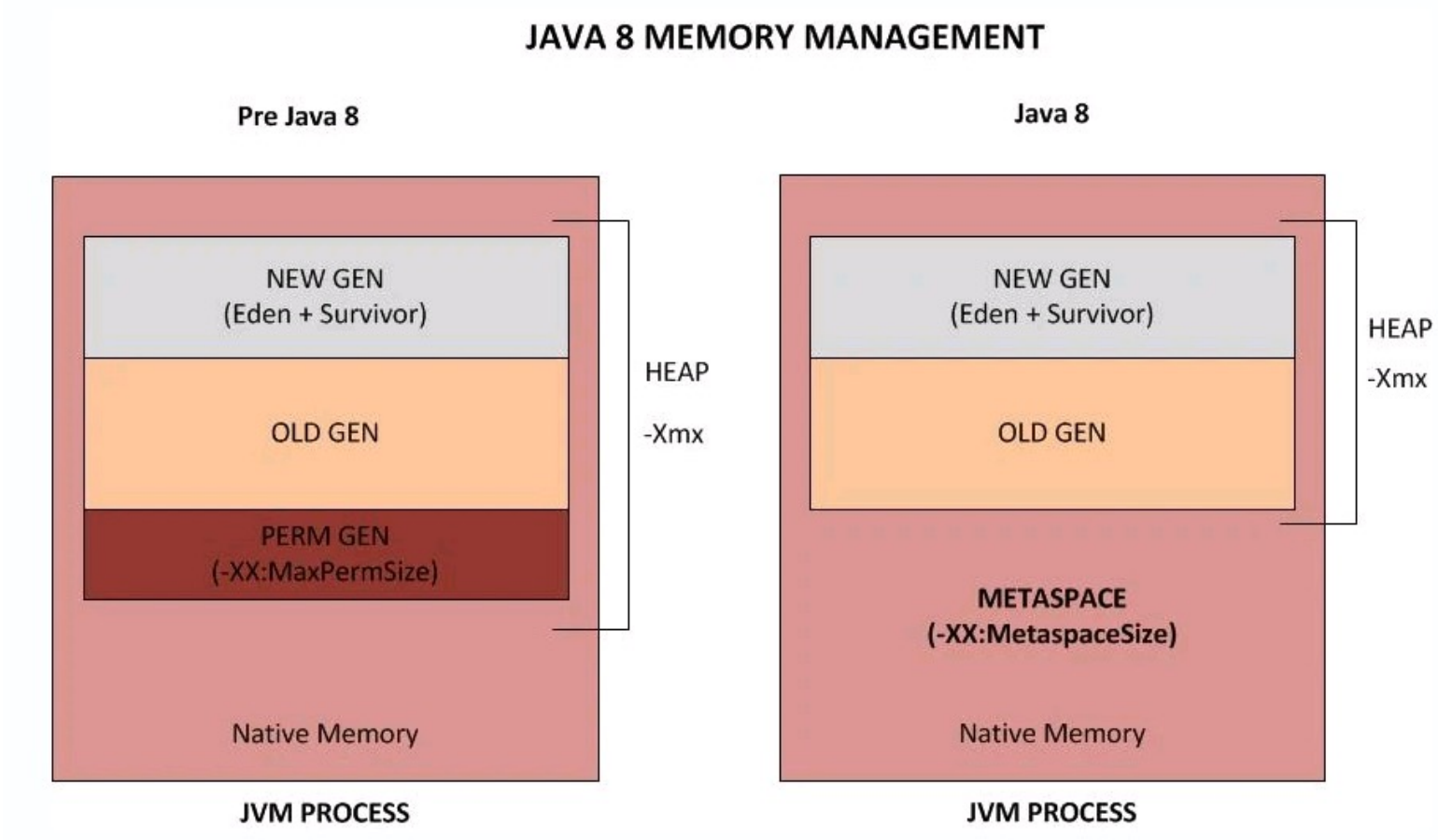
✓ Переменные в стеке существуют до тех пор, пока выполняется метод в котором они были созданы

✓ Если память стека будет заполнена, Java бросит исключение java.lang.StackOverFlowError

✓ Доступ к этой области памяти осуществляется быстрее, чем к куче

✓ Является потокобезопасным, поскольку для каждого потока создается свой отдельный стек

Heap


JAVA 8 MEMORY MANAGEMENT
Pre Java 8
NEW GEN
(Eden + Survivor)
OLD GEN
PERM GEN
(-XX:MaxPermSize)
Native Memory
HEAP
-Xmx
JVM PROCESS
Java 8
NEW GEN
(Eden + Survivor)
OLD GEN
METASPACE
(-XX:MetaspaceSize)
Native Memory
HEAP
-Xmx
JVM PROCESS

Эта область памяти разбита на несколько более мелких частей, называемых поколениями:

Young Generation — область где размещаются недавно созданные объекты. Когда она заполняется, происходит быстрая сборка мусора

Old (Tenured) Generation — здесь хранятся долгоживущие объекты. Когда объекты из Young Generation достигают определенного порога "возраста", они перемещаются в Old Generation

Permanent Generation — эта область содержит метаинформацию о классах и методах приложения, но начиная с Java 8 данная область памяти была упразднена.

Основные особенности кучи:

✓ Когда эта область памяти полностью заполняется, Java бросает java.lang.OutOfMemoryError

✓Доступ к ней медленнее, чем к стеку

✓Эта память, в отличие от стека, автоматически не освобождается. Для сбора неиспользуемых объектов используется сборщик мусора

✓В отличие от стека, куча не является потокобезопасной и ее необходимо контролировать, правильно синхронизируя код

Свойства	Стек	Куча
Использование приложением	Для каждого потока используется свой стек	Пространство кучи является общим для всего приложения
Размер	Предел размера стека определен операционной системой	Размер кучи не ограничен
Хранение	Хранит примитивы и ссылки на объекты	Все созданные объекты хранятся в куче
Порядок	Работает по схеме последним вошел, первым вышел (LIFO)	Доступ к этой памяти осуществляется с помощью сложных методов управления памятью, включая Young Generation, Old и Permanent Generation
Существование	Память стека существует пока выполняется текущий метод	Пространство кучи существует пока работает приложение
Скорость	Обращение к памяти стека происходит значительно быстрее, чем к памяти кучи	Медленнее, чем стек
Выделение и освобождение памяти	Эта память автоматически выделяется и освобождается, когда метод вызывается и завершается соответственно	Память в куче выделяется, когда создается новый объект и освобождается сборщиком мусора, когда в приложении не остается ни одной ссылки на его

1. java.lang.OutOfMemoryError: Java heap space

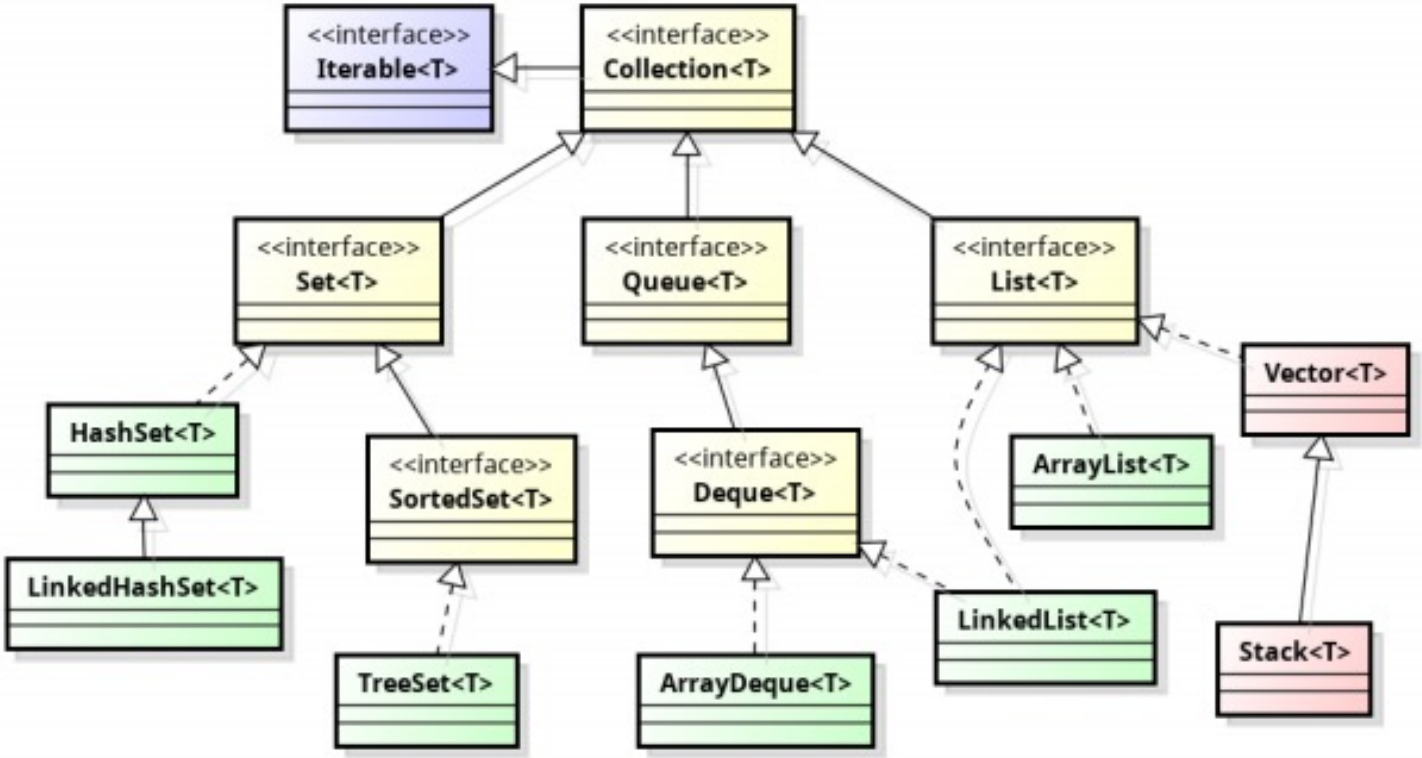
2. java.lang.OutOfMemoryError: PermGen space

3. java.lang.OutOfMemoryError: GC overhead limit exceeded

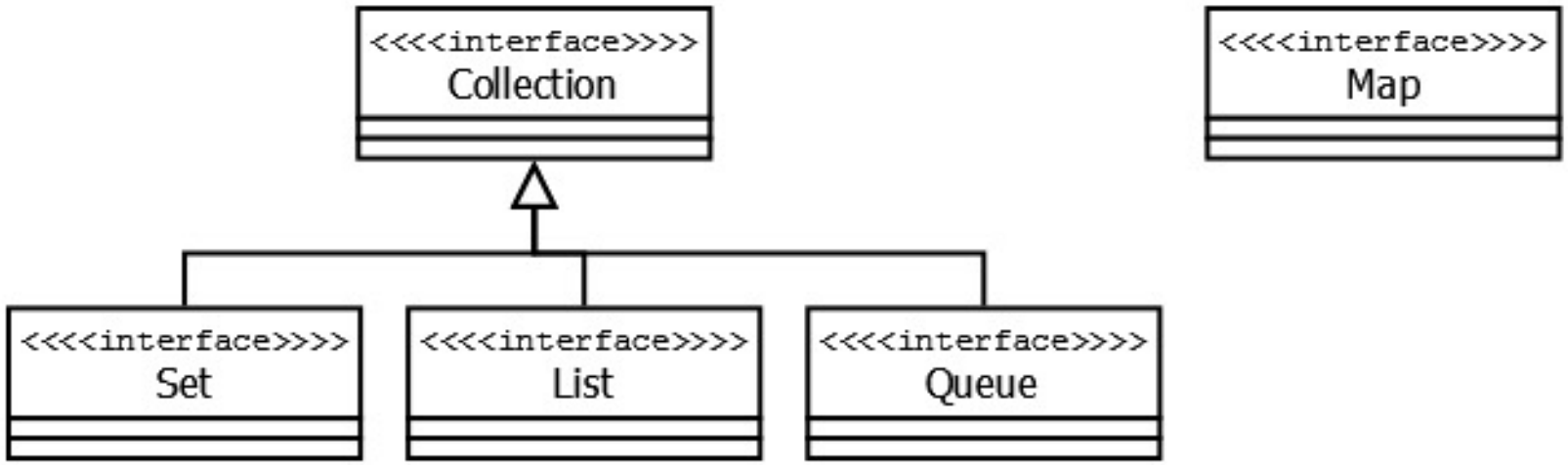
4. java.lang.OutOfMemoryError: unable to create new native thread

5. java.lang.OutOfMemoryError: Metaspace

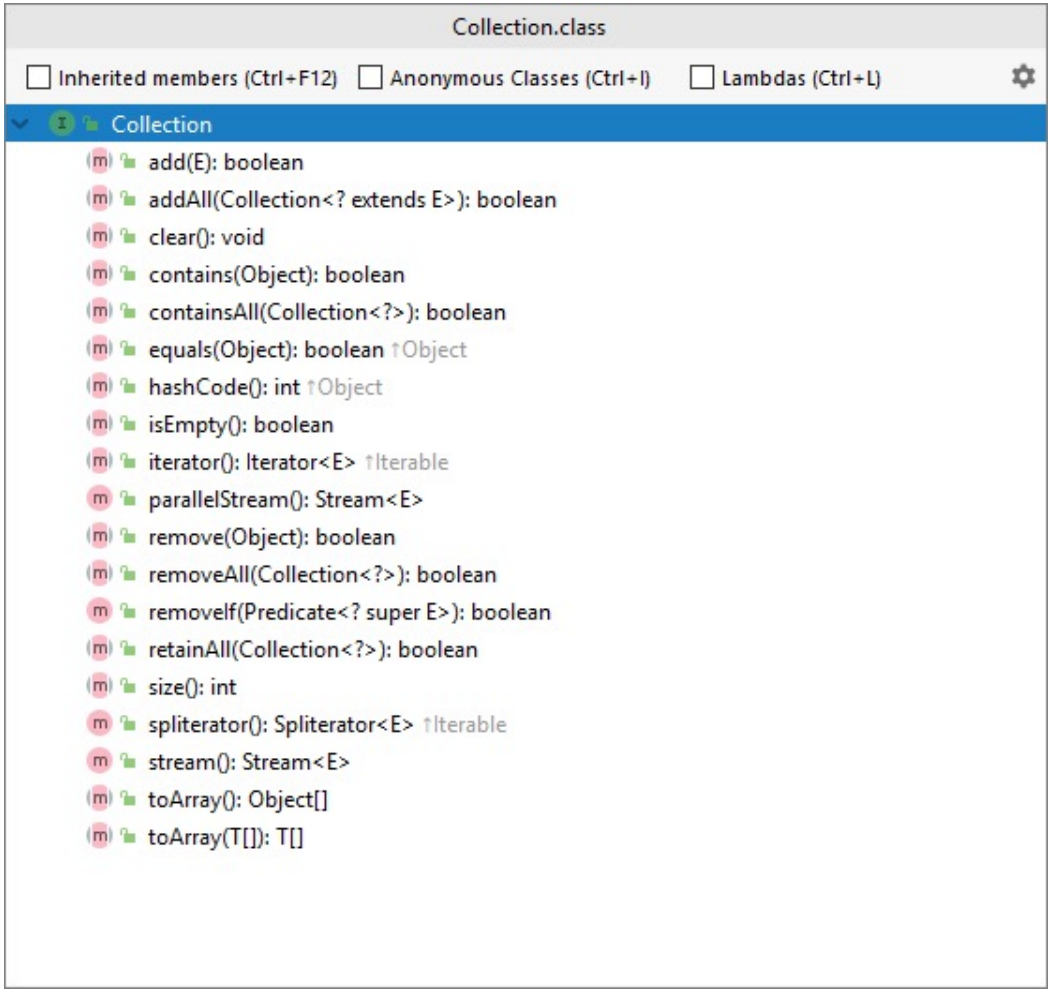
Java Collections Framework



Java Collections Framework

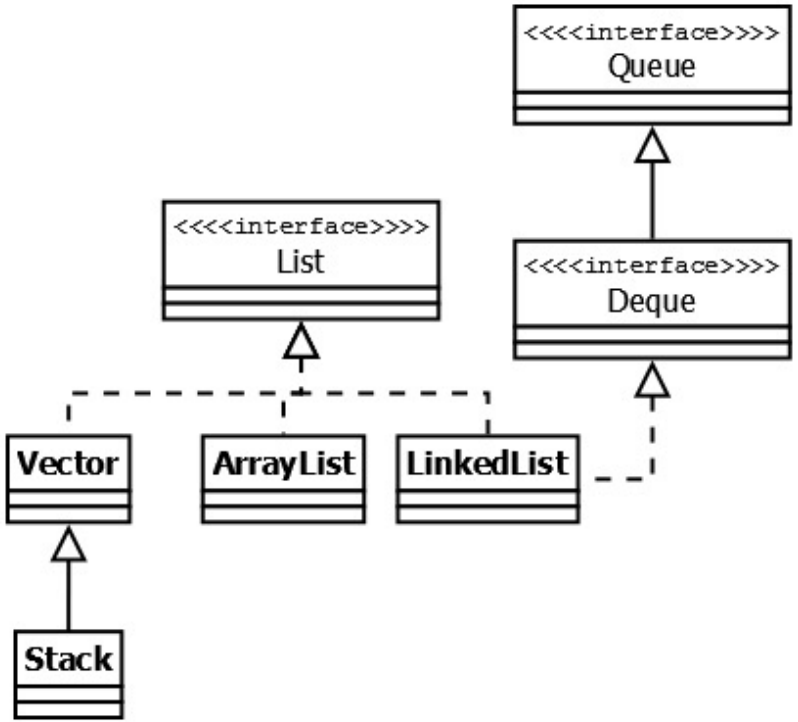


Collection - простая последовательность занчений
Map – набор пар “ключ значение”


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[]

Интерфейс List

Реализации этого интерфейса представляют собой упорядоченные коллекции. Кроме того, разработчику предоставляется возможность доступа к элементам коллекции по индексу и по значению



Интерфейс List

Vector — реализация динамического массива объектов. Позволяет хранить любые данные, включая null в качестве элемента.

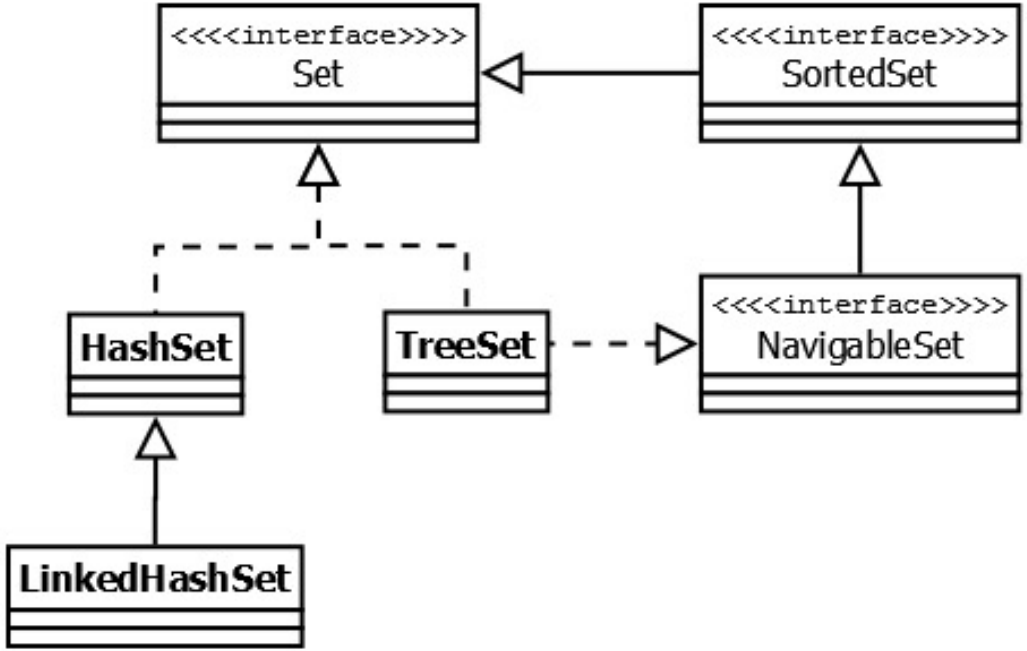
Stack — данная коллекция является расширением коллекции Vector с реализацией стека LIFO (last-in-first-out).

LinkedList — позволяет хранить любые данные, включая null. Особенностью реализации данной коллекции является то, что в её основе лежит двунаправленный связный список.

ArrayList — является реализацией динамического массива объектов. Позволяет хранить любые данные, включая null в качестве элемента, реализован на основе обычного массива.

Интерфейс Set

Представляет собой неупорядоченную коллекцию, которая не может содержать дублирующиеся данные. Является программной моделью математического понятия «множество».



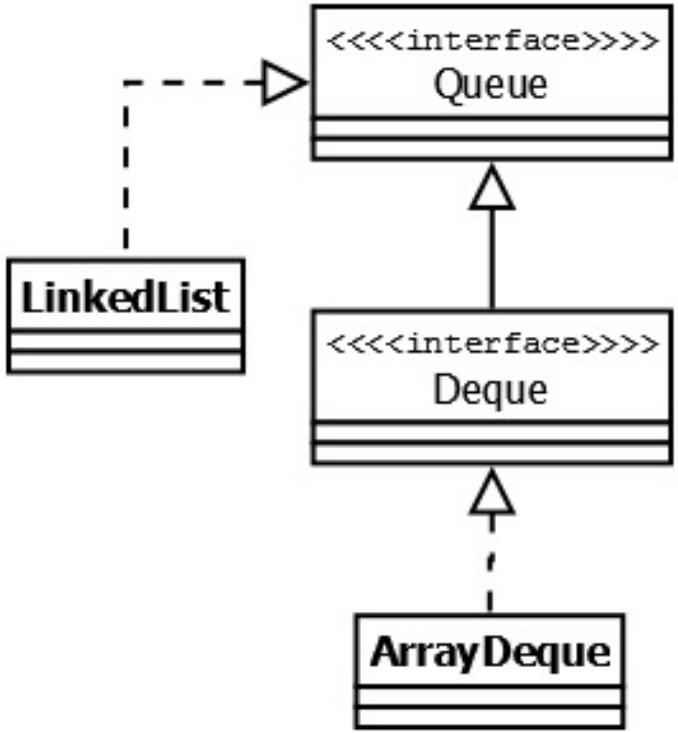
HashSet — реализация интерфейса Set, базирующаяся на HashMap. Внутри использует объект HashMap для хранения данных. В качестве ключа используется добавляемый элемент, а в качестве значения — объект-пустышка (new Object()).

LinkedHashSet — отличается от HashSet только тем, что в основе лежит LinkedHashMap вместо HashSet. Благодаря этому отличию порядок элементов при обходе коллекции является идентичным порядку добавления элементов.

TreeSet — аналогично другим классам-реализациям интерфейса Set содержит в себе объект NavigableMap, что и обуславливает его поведение. Предоставляет возможность управлять порядком элементов в коллекции при помощи объекта Comparator, либо сохраняет элементы с использованием "natural ordering".

Интерфейс Queue

Этот интерфейс описывает коллекции с предопределённым способом вставки и извлечения элементов, а именно — очереди FIFO (first-in-first-out).



PriorityQueue — является единственной прямой реализацией интерфейса Queue , не считая класса LinkedList. Особенностью данной очереди является возможность управления порядком элементов. По-умолчанию, элементы сортируются с использованием «natural ordering», но это поведение может быть переопределено при помощи объекта Comparator, который задаётся при создании очереди. Данная коллекция не поддерживает null в качестве элементов.

ArrayDeque — реализация интерфейса Deque, который расширяет интерфейс Queue методами, позволяющими реализовать конструкцию вида LIFO (last-in-first-out). Эта коллекция представляет собой реализацию с использованием массивов, подобно ArrayList, но не позволяет обращаться к элементам по индексу и хранение null. Как заявлено в документации, коллекция работает быстрее чем Stack, если используется как LIFO коллекция, а также быстрее чем LinkedList, если используется как FIFO.

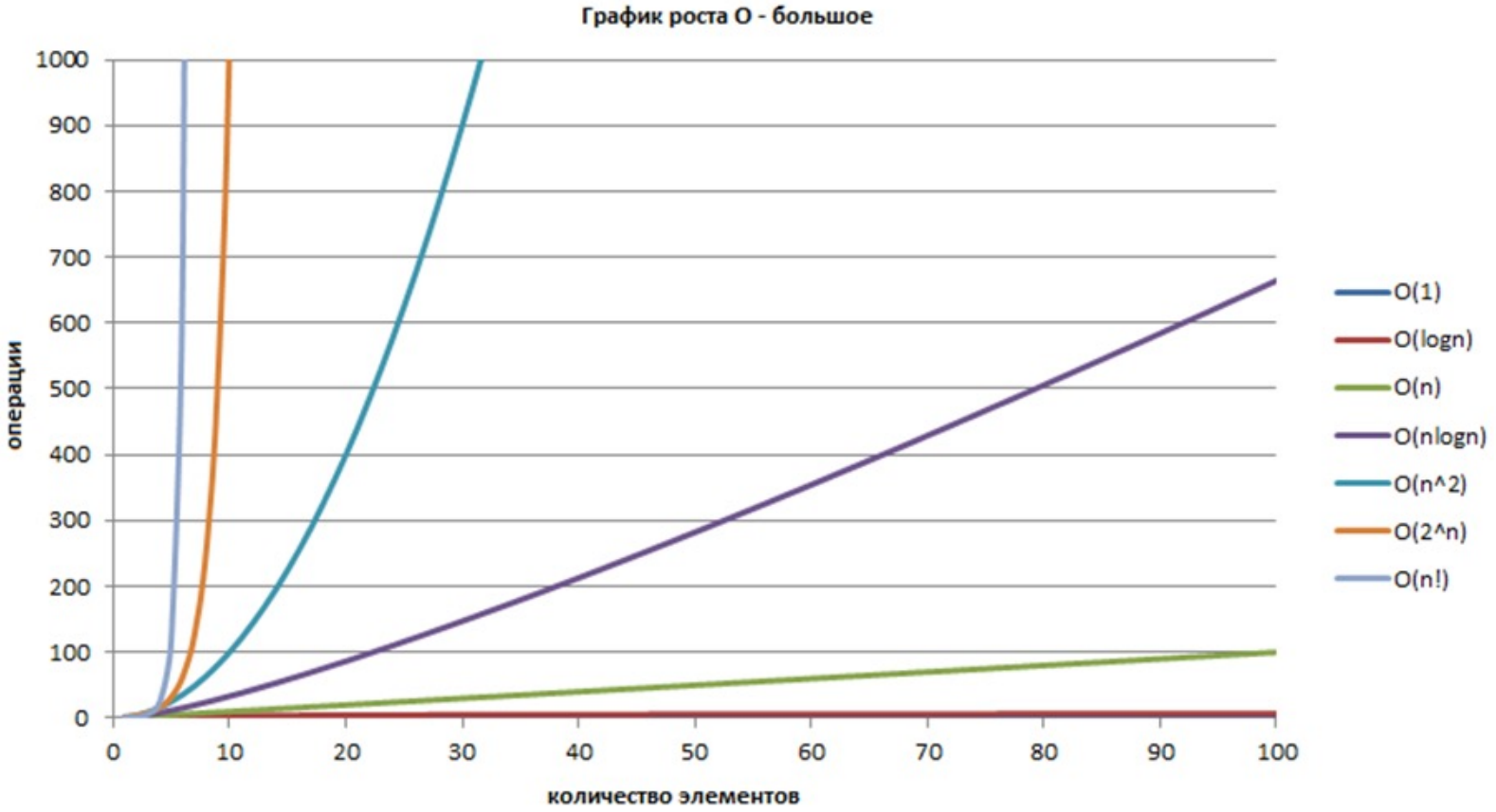


График роста O - большое
операции
1000
900
800
700
600
500
400
300
200
100
0
0
10
20
30
40
50
60
70
80
90
100
количество элементов
O(1)
O(logn)
O(n)
O(nlogn)
O(n^2)
O(2^n)
O(n!)

	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))