

# JAVA – получи Чёрный Пояс!



# Comparable & Comparator

Интерфейс Comparable используется для сравнения объектов, используя естественный порядок

```
int compareTo(Element e)
```

Интерфейс Comparator используется для сравнения объектов, используя НЕ естественный порядок

```
int compare(Element e1, Element e2)
```

# Generics

Type Safe & Reusable Code

Parameterized Class

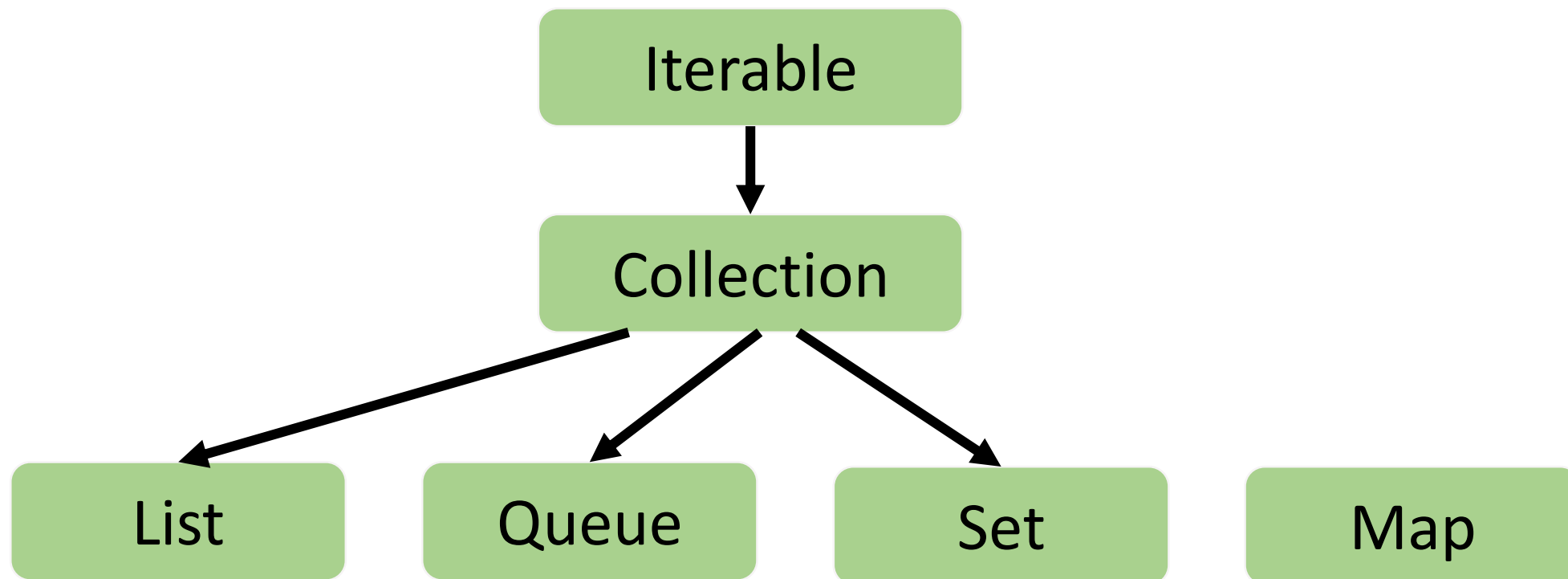
Parameterized Method

**<?>** - любой класс

**<? extends X>** - класс X или любой его subclasses

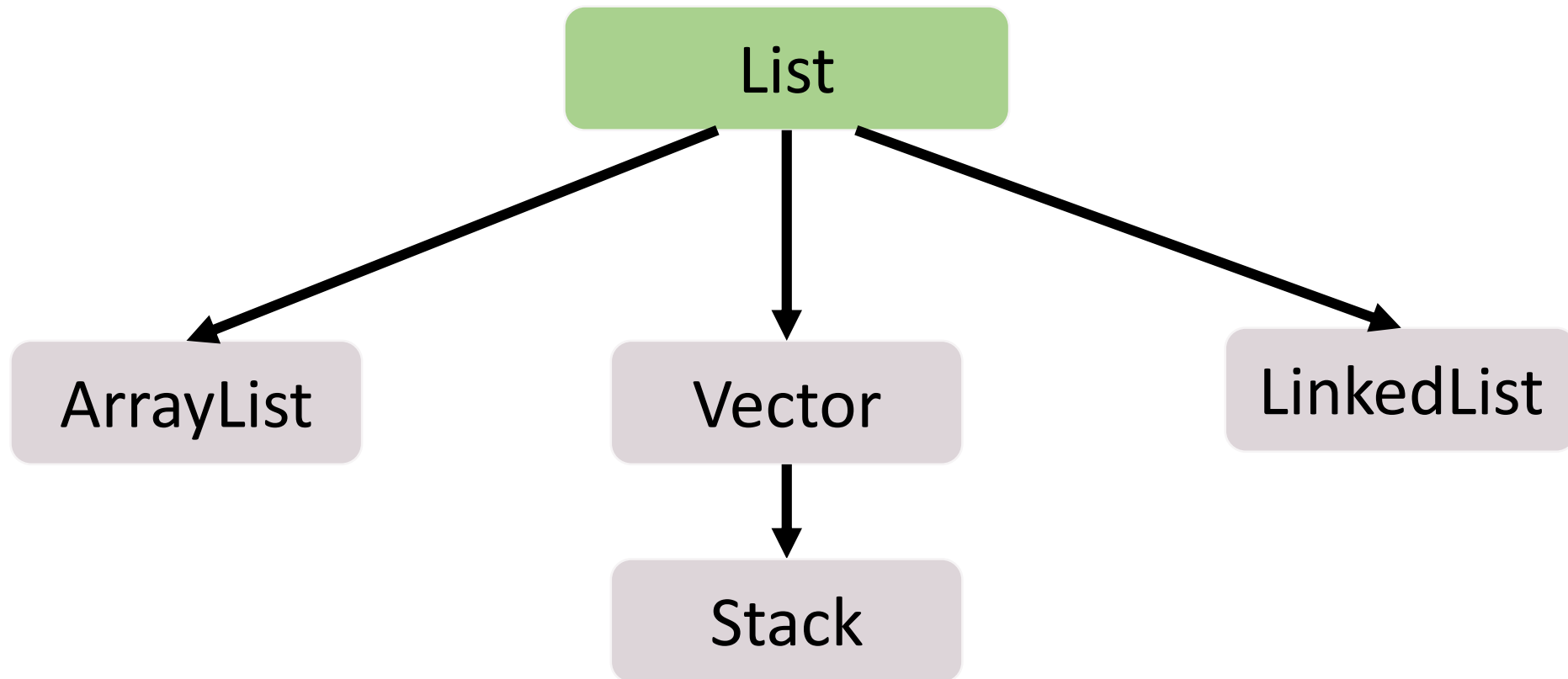
**<? super Y>** - класс Y или любой его суперкласс

# Иерархия коллекций



# List

List – упорядоченная последовательность элементов, позволяющая хранить дубликаты и null. Каждый элемент имеет индекс.



# ArrayList

В основе ArrayList лежит массив Object

```
ArrayList <DataType> list1 = new ArrayList <DataType> ();
```

```
ArrayList <DataType> list2 = new ArrayList <> ();
```

```
ArrayList <DataType> list3 = new ArrayList <> (30);
```

```
ArrayList <DataType> list4 = new ArrayList <> (list3);
```

# Методы ArrayList

add(DataType element) → boolean  
add(int index, DataType element) → boolean

set(int index, DataType element) → DataType

addAll(ArrayList aL) → boolean  
addAll(int index, ArrayList aL) → boolean

indexOf(Object element) → int

size() → int

contains(Object element) → boolean

get(int index) → DataType

remove(Object element) → boolean  
remove(int index) → boolean

clear() → void

lastIndexOf(Object element) → int

isEmpty() → boolean

toString() → String

# Методы ArrayList и связанные с ArrayList

`Arrays.asList(DataType []) → List<DataType>`

`removeAll(Collection <?> c) → boolean`

`retainAll(Collection <?> c) → boolean`

`containsAll(Collection <?> c) → boolean`

`subList(int fromIndex, int toIndex) → List<E>`

`toArray() → Object []`

`toArray(T [] a) → T []`

`List.of(E ... elements) → List<E>`

`List.copyOf(Collection <E> c) → List<E>`



# Iterator

some code

.....

```
Iterator<DataType> iter = aL.iterator();  
    while (iter.hasNext())  
{        System.out.println(iter.next());    }
```

# LinkedList

Элементы LinkedList – это звенья одной цепочки. Эти элементы хранят определённые данные, а также ссылки на предыдущий и следующий элементы.

Как правило, LinkedList следует использовать когда:

- 1) Невелико количество операций получения элементов;
- 2) Велико количество операций добавления и удаления элементов. Особенно если речь идёт о элементах в начале коллекции.

# ListIterator

some code

.....

```
ListIterator<DataType> listIter = aL.listIterator();
```

```
    while (listIter.hasNext())  
{        System.out.println(listIter.next());    }
```

```
    while (listIter.hasPrevious())  
{        System.out.println(listIter.previous());    }
```

# Vector

Vector – устаревший synchronized класс. В своей основе содержит массив элементов Object.  
Не рекомендован для использования.

add

get

remove

firstElement

lastElement

# Stack

Stack – устаревший `synchronized` класс. Использует принцип LIFO.

Не рекомендован для использования.

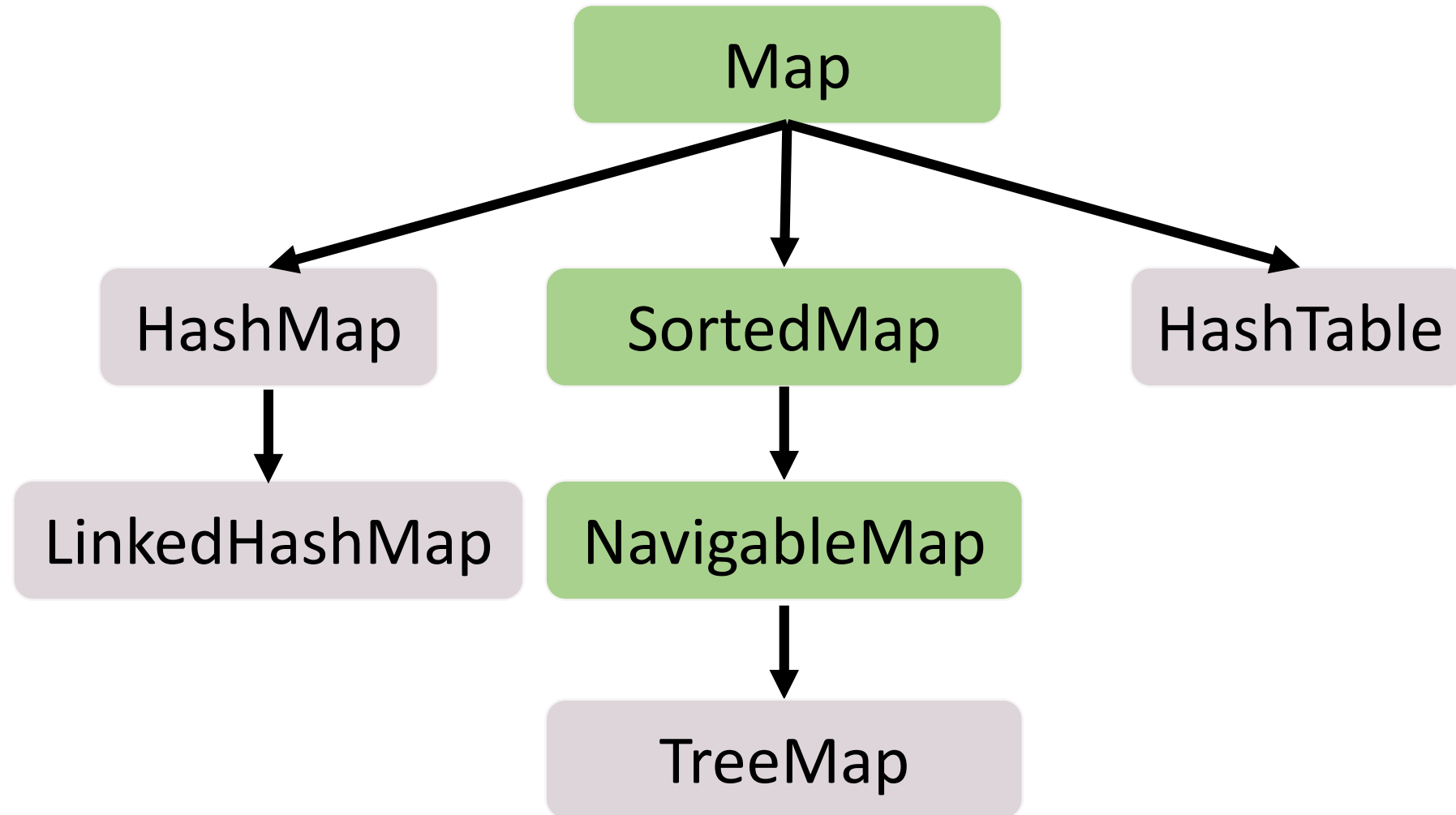
push

pop

peek

isEmpty

# Map



# HashMap

Элементами HashMap являются пары ключ/значение. HashMap не запоминает порядок добавления элементов. Его методы работают очень быстро.

Ключи элементов должны быть уникальными. Ключ может быть null.

Значения элементов могут повторяться. Значения могут быть null.

put

putIfAbsent

get

remove

containsValue

containsKey

keySet

values

entrySet

# Методы equals и hashCode

Если Вы переопределили equals, то переопределите и hashCode.

Результат нескольких выполнений метода hashCode для одного и того же объекта должен быть одинаковым.

Если, согласно методу equals, два объекта равны, то и hashCode данных объектов обязательно должен быть одинаковым.

Если, согласно методу equals, два объекта НЕ равны, то hashCode данных объектов НЕ обязательно должен быть разным.

Ситуация, когда результат метода hashCode для разных объектов одинаков, называется коллизией. Чем её меньше, тем лучше.



# HashMap

В основе HashMap лежит массив. Элементами данного массива являются структуры LinkedList. Данные структуры LinkedList и заполняются элементами, которые мы добавляем в HashMap.

При создании HashMap мы можем задать 2 параметра, которые очень влияют на производительность:

- Initial capacity – начальный размер массива;
- Load factor – коэффициент того, насколько массив должен быть заполнен, после чего его размер будет увеличен вдвое.

# TreeMap

Элементами TreeMap являются пары ключ/значение. В TreeMap элементы хранятся в отсортированном по возрастанию порядке.

В основе TreeMap лежит красно-чёрное дерево. Это позволяет методам работать быстро, но не быстрее, чем методы HashMap.

put

get

remove

descendingMap

tailMap

headMap

lastEntry

firstEntry

# LinkedHashMap

LinkedHashMap является наследником HashMap. Хранит информацию о порядке добавления элементов или порядке их использования. Производительность методов немного ниже, чем у методов HashMap.

# HashTable

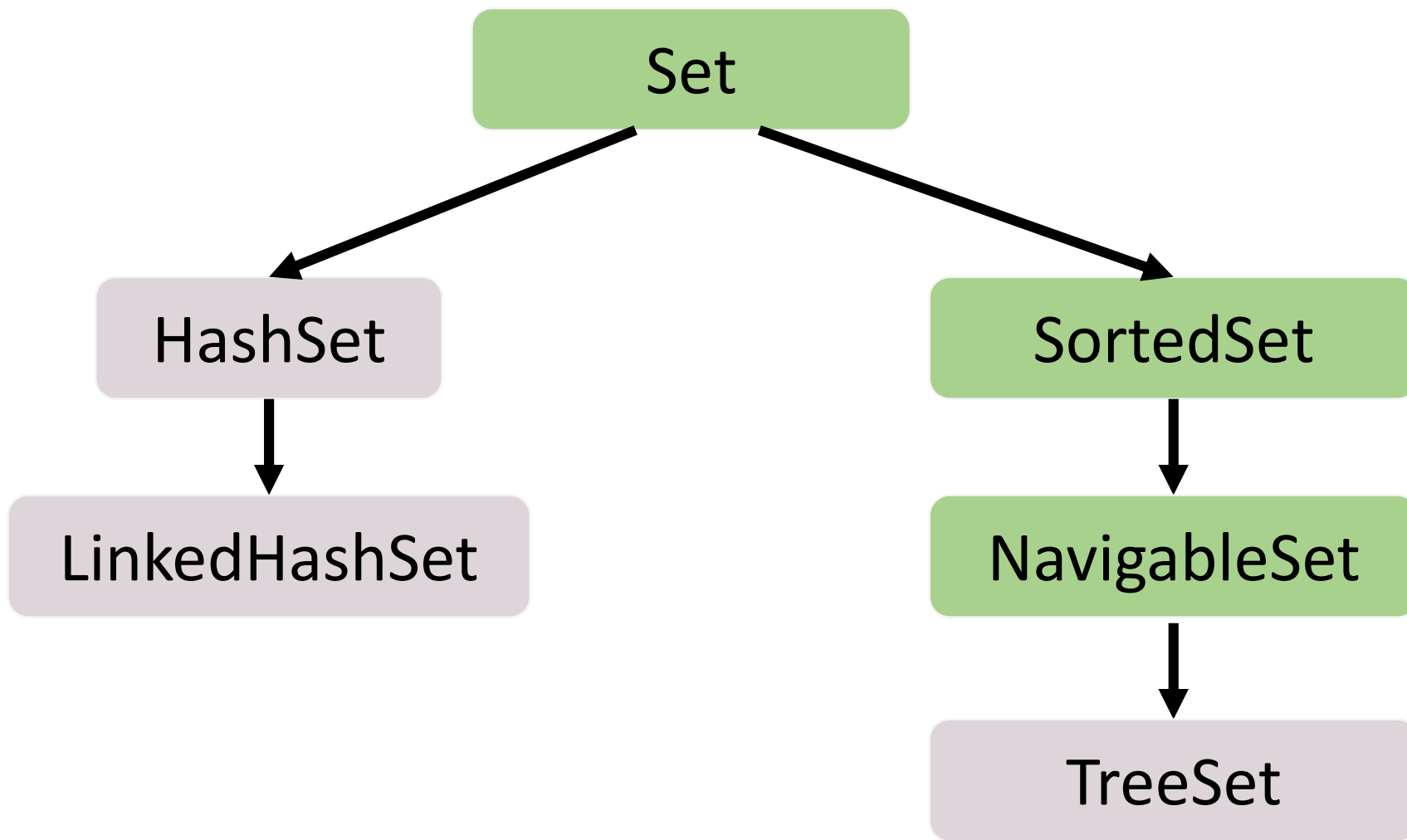
HashTable устаревший класс, который работает по тем же принципам, что и HashMap.

В отличии от HashMap является synchronized. По этой причине его методы далеко не такие быстрые.

В HashTable ни ключ, ни значение не могут быть null.

Даже если нужна поддержка многопоточности HashTable лучше не использовать. Следует использовать ConcurrentHashMap.

# Set



# Set and HashSet

Set – коллекция, хранящая уникальные элементы. Методы данной коллекции очень быстрые.

HashSet не запоминает порядок добавления элементов. В основе HashSet лежит HashMap. У элементов данного HashMap: ключи - это элементы HashSet, значения – это константа-заглушка.

add

remove

size

isEmpty

contains

addAll

retainAll

removeAll

# TreeSet

TreeSet хранит элементы в отсортированном по возрастанию порядке.

В основе TreeSet лежит TreeMap. У элементов данного TreeMap: ключи - это элементы TreeSet, значения – это константа-заглушка.

first

last

tailSet

headSet

subSet

# LinkedHashSet

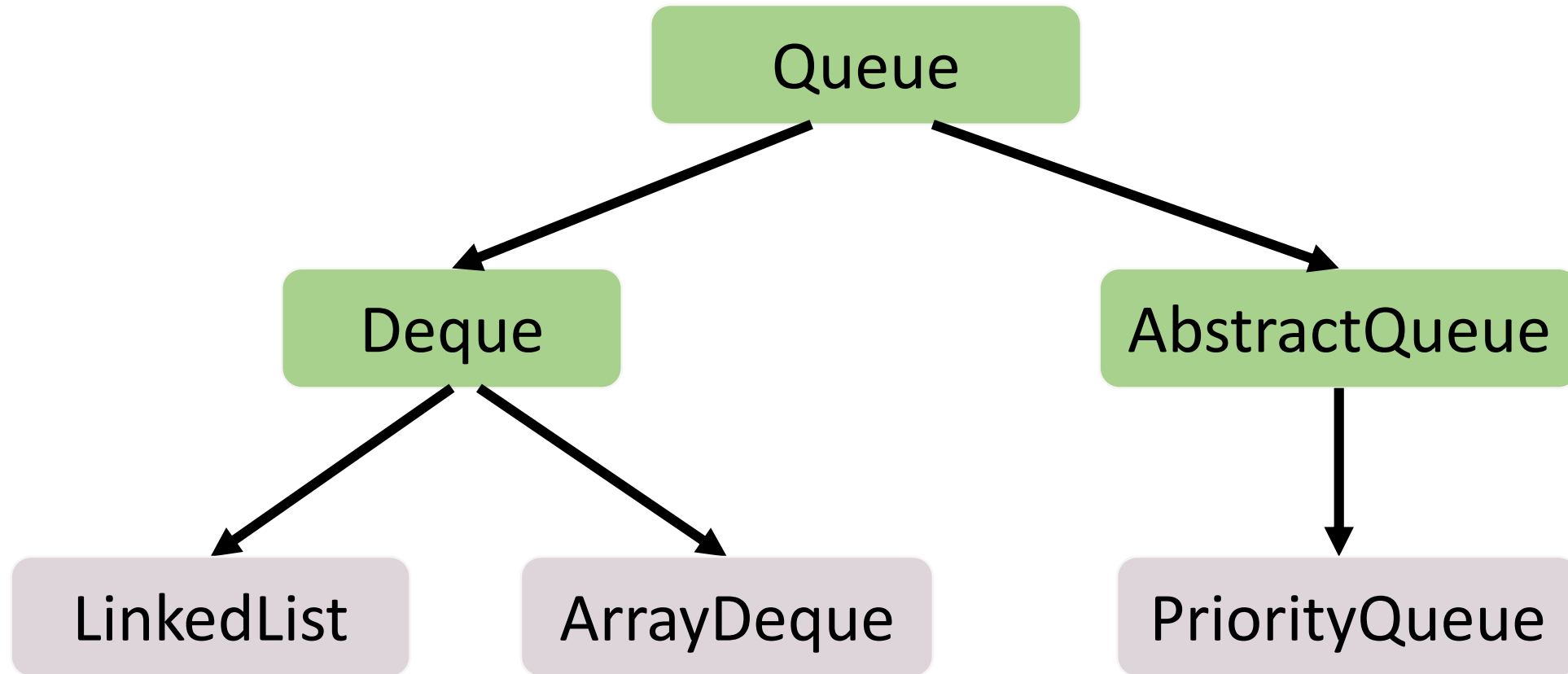
LinkedHashSet является наследником HashSet. Хранит информацию о порядке добавления элементов.

Производительность методов немного ниже, чем у методов HashSet.

В основе LinkedHashSet лежит HashMap. У элементов данного HashMap: ключи - это элементы LinkedHashSet, значения — это константа-заглушка.



# Queue



# Queue and LinkedList

Queue – это коллекция, хранящая последовательность элементов. Добавляется элемент в конец очереди, используется из начала очереди – правило FIFO.

Класс LinkedList implements не только интерфейс List, но и интерфейс Deque.

add

offer

remove

poll

element

peek

# PriorityQueue

PriorityQueue – это специальный вид очереди, в котором используется натуральная сортировка или та, которую мы описываем с помощью Comparable или Comparator. Таким образом используется тот элемент из очереди, приоритет которого выше.

# Deque and ArrayDeque

Deque – double ended queue (двунаправленная очередь). В такой очереди элементы могут использоваться с обоих концов. Здесь работают оба правила – FIFO и LIFO.

Интерфейс Deque реализуется классами LinkedList и ArrayDeque.

addFirst

addLast

offerFirst

offerLast

removeFirst

removeLast

pollFirst

pollLast

getFirst

getLast

peekFirst

peekLast

# **Nested classes**

## **(Вложенные классы)**

```
graph TD; A["Nested classes (Вложенные классы)"] --> B[LOCAL]; A --> C[static]; A --> D[INNER]; A --> E[ANONYMOUS];
```

static

INNER

LOCAL

ANONYMOUS

# static nested класс

- static nested класс очень похож на обычный внешний, но находится внутри другого класса
- Создавая объект static nested класса, нужно указывать и класс, содержащий его
- static nested класс может содержать static и non-static элементы
- static nested класс может обращаться даже к private элементам внешнего класса, но только к static
- Внешний класс может обращаться даже к private элементам static nested класса

# Inner класс

- Каждый объект inner класса всегда ассоциируется с объектом внешнего класса
- Создавая объект inner класса, нужно перед этим создать объект его внешнего класса
- Inner класс может содержать только non-static элементы
- Inner класс может обращаться даже к private элементам внешнего класса
- Внешний класс может обращаться даже к private элементам inner класса, прежде создав его объект

# Local inner класс

- Local inner класс располагается в блоках кода таких, как, например, метод или конструктор
- Local inner класс не может быть static
- Область видимости local inner класса – это блок, в котором он находится
- Local inner класс может обращаться даже к private элементам внешнего класса
- Local inner класс может обращаться к элементам блока, в котором он написан при условии, что они final или effectively final



# Anonymous inner класс

- Anonymous класс не имеет имени
- Anonymous класс – это «объявление» класса и одновременное создание объекта
- В анонимных классах невозможно написать конструктор
- Анонимный класс может обращаться даже к private элементам внешнего класса
- Lambda expressions – это краткая форма для написания анонимных классов

# Lambda expressions

Самый короткий вариант написания лямбда выражения:

```
stud -> stud.avgGrade > 8.5
```

Более полный вариант написания лямбда выражения:

```
(Student stud) -> {return stud.avgGrade > 8.5;}
```

В лямбда выражении оператор стрелка разделяет параметры метода и тело метода.

В лямбда выражении справа от оператора стрелка находится тело метода, которое было бы у метода соответствующего класса, имплементировавшего наш интерфейс с единственным методом.

# Lambda expressions

Вы можете использовать смешанный вариант написания лямбда выражения: слева от оператора стрелка писать короткий вариант, справа – полный. Или наоборот.

Если вы используете полный вариант написания для части лямбда выражения справа от стрелки, то вы должны использовать слово `return` и знак «;»

Левая часть лямбда выражения может быть написана в краткой форме, если метод интерфейса принимает только 1 параметр. Даже если метод интерфейса принимает 1 параметр, но в лямбда выражении вы хотите писать данный параметр используя его тип данных, тогда уже вы должны писать левую часть лямбда выражения в скобках.

Если в правой части лямбда выражения вы пишете более одного `statement`-а, то вы должны использовать его полный вариант написания.

# Lambda expressions

```
def( () -> 5 );
```

```
def( (x) -> x.length() );
```

```
def( (String x) -> x.length() );
```

```
def( (x, y) -> x.length() );
```

```
def( (String x, String y) -> x.length() );
```

Compile time errors:

```
def( x -> {x.length();} );
```

```
def( x -> {return x.length()} );
```

```
def( x, y -> x.length() );
```

# Lambda expressions

```
method( (int x, int y) -> {int x=5; return10;} );
```

NOT OK

```
method( (int x, int y) -> {x=5; return10;} );
```

OK

```
method( (int x, int y) -> {int x2=5; return10;} );
```

OK

Лямбда выражения работают с интерфейсом, в котором есть только 1 абстрактный метод. Такие интерфейсы называются функциональными интерфейсами, т.е. интерфейсами, пригодными для функционального программирования.

# Пакет java.util.function

Predicate<T>

Используется методом removeIf

```
boolean test(T t);
```

Supplier<T>

```
T get ();
```

Consumer<T>

Используется методом forEach

```
void accept (T t);
```

Function<T, R>

```
R apply (T t);
```

# Stream

Stream – это последовательность элементов, поддерживающих последовательные и параллельные операции над ними.

# Методы Stream

Методы Stream не меняют саму коллекцию или массив, от которой был создан stream

map

(i)

filter

(i)

forEach

(t)

reduce

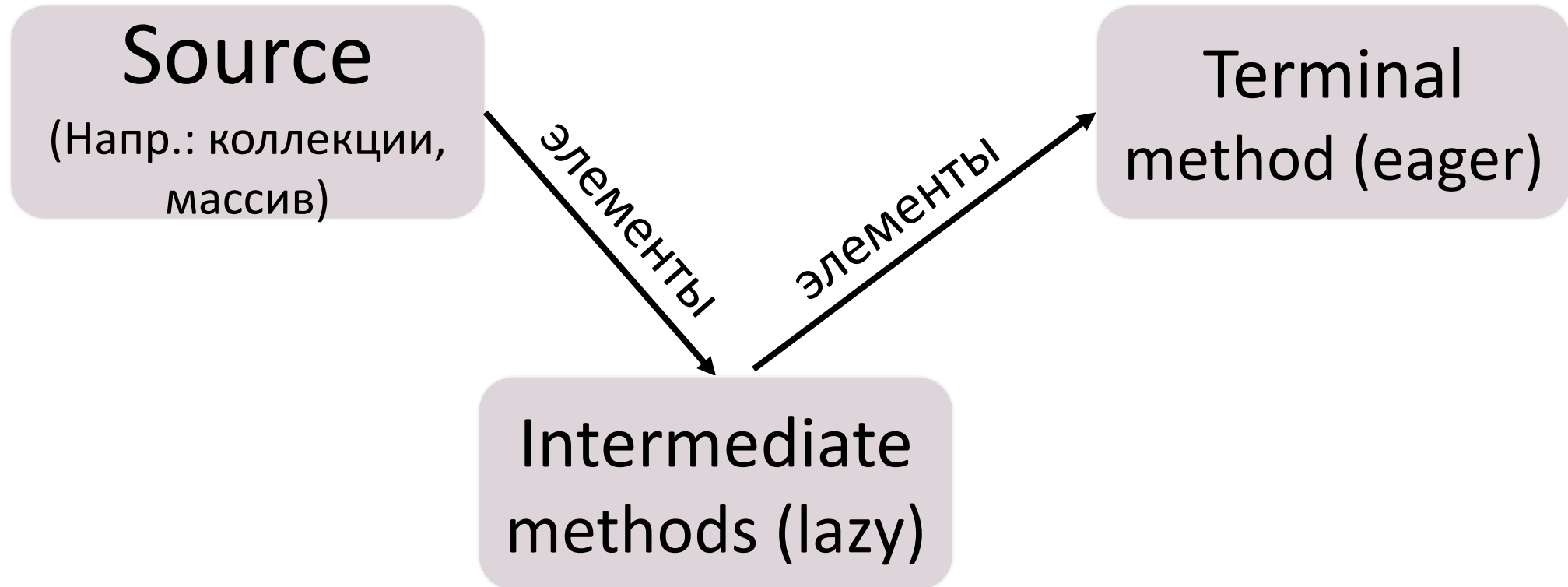
(t)

sorted

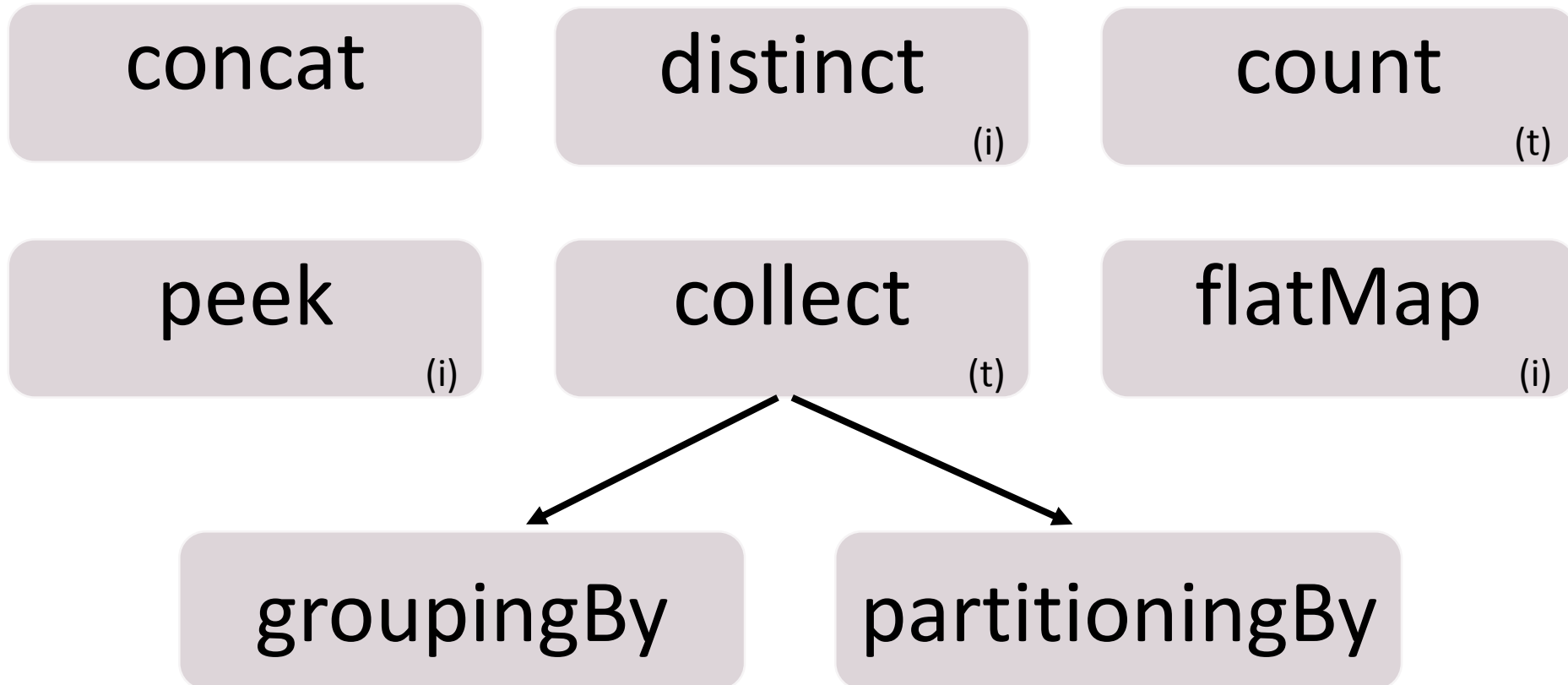
(i)



# Работа метод chaining в stream



# Методы Stream



# Методы Stream

findFirst

(t)

min

(t)

max

(t)

limit

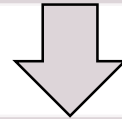
(i)

mapToInt

(i)

skip

(i)



sum

average

min

max

# Parallel stream

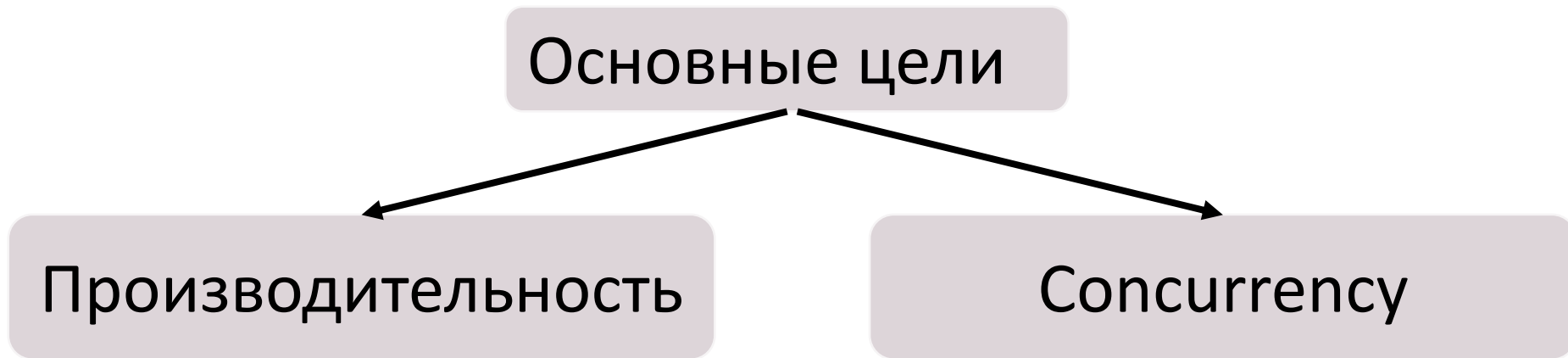
Parallel stream – это возможность использования нескольких ядер процессора при выполнении каких-либо операций со stream.

```
list.parallelStream(). ...
```

```
Stream<T> s = Stream.of(...);  
s.parallel(). ...
```

# Multithreading

Многопоточность – это принцип построения программы, при котором несколько блоков кода могут выполняться одновременно.



# Варианты создания нового потока

```
//Создание  
class MyThread extends Thread{  public void run() {  код  }  }  
//Запуск  
new MyThread().start();
```

```
//Создание  
class MyRunnableImpl implements Runnable{  public void run() { код }  }  
//Запуск  
new Thread(  new MyRunnableImpl()  ).start();
```

Из за того, что в Java отсутствует множественное наследование, чаще используют 2-ой вариант.

# Методы Thread

setName

getName

setPriority

getPriority

sleep

join

# Concurrency / Parallelism

## Asynchronous / Synchronous

Concurrency означает выполнение сразу нескольких задач. В зависимости от процессора компьютера concurrency может достигаться разными способами.

Parallelism означает выполнение 2-х и более задач в одно и то же время, т.е. параллельно. В компьютерах с многоядерным процессором concurrency может достигаться за счёт parallelism.

В синхронном программировании задачи выполняются последовательно друг за другом.

В асинхронном программировании каждая следующая задача НЕ ждёт окончания выполнения предыдущей. Асинхронное программирование помогает достичь concurrency.



# Ключевое слово `volatile`

Ключевое слово `volatile` используется для пометки переменной, как хранящейся только в основной памяти «main memory».

Для синхронизации значения переменной между потоками ключевое слово `volatile` используется тогда, когда только один поток может изменять значение этой переменной, а остальные потоки могут его только читать.

# Data race и synchronized методы

Data race – это проблема, которая может возникнуть когда два и более потоков обращаются к одной и той же переменной и как минимум 1 поток её изменяет.

Пример метода:

```
public synchronized void abc() { method body }
```

# Понятие «монитор» и `synchronized` блоки

Монитор – это сущность/механизм, благодаря которому достигается корректная работа при синхронизации.  
В Java у каждого класса и объекта есть привязанный к нему монитор.

Пример блока:

```
static final Object lock = new Object();  
public void abc() {  
    synchronized(lock) {  
        block body  
    }  
}
```

*method body*

*method body*

# Методы `wait` и `notify`

Для извещения потоком других потоков о своих действиях часто используются следующие методы:

`wait` - освобождает монитор и переводит вызывающий поток в состояние ожидания до тех пор, пока другой поток не вызовет метод `notify()`;

`notify` – НЕ освобождает монитор и будит поток, у которого ранее был вызван метод `wait()`;

`notifyAll` – НЕ освобождает монитор и будит все потоки, у которых ранее был вызван метод `wait()`;

# Возможные ситуации в многопоточном программировании

Deadlock – ситуация, когда 2 или более потоков залочены навсегда, ожидают друг друга и ничего не делают.

Livelock – ситуация, когда 2 или более потоков залочены навсегда, ожидают друг друга, проделывают какую-то работу, но без какого-либо прогресса.

Lock starvation – ситуация, когда менее приоритетные потоки ждут долгое время или всё время для того, чтобы могли запуститься.

# Lock и ReentrantLock

Lock – интерфейс, который имплементируется классом ReentrantLock.

Также как ключевое слово `synchronized`, Lock нужен для достижения синхронизации между потоками.

`lock()`

`unlock()`

`tryLock()`

# Даemon потоки

Даemon потоки предназначены для выполнения фоновых задач и оказания различных сервисов User потокам.

При завершении работы последнего User потока программа завершает своё выполнение, не дожидаясь окончания работы Даemon потоков.

`setDaemon()`

`isDaemon()`

# Прерывание потоков

У нас есть возможность послать сигнал потоку, что мы хотим его прервать.

У нас также есть возможность в самом потоке проверить, хотят ли его прервать. Что делать, если данная проверка показала, что поток хотят прервать, должен решать сам программист.

`interrupt()`

`isInterrupted()`



# Thread pool и ExecutorService

Thread pool – это множество потоков, каждый из которых предназначен для выполнения той или иной задачи.

В Java с thread pool-ами удобнее всего работать посредством ExecutorService.

Thread pool удобнее всего создавать, используя factory методы класса Executors:

`Executors.newFixedThreadPool(int count)` – создаст pool с 5-ю потоками;

`Executors.newSingleThreadExecutor()` – создаст pool с одним потоком.

# Thread pool и ExecutorService

Метод **execute** передаёт наше задание (task) в thread pool, где оно выполняется одним из потоков.

После выполнения метода **shutdown** ExecutorService понимает, что новых заданий больше не будет и, выполнив поступившие до этого задания, прекращает работу.

Метод **awaitTermination** принуждает поток в котором он вызвался подождать до тех пор, пока не выполнится одно из двух событий: либо ExecutorService прекратит свою работу, либо пройдёт время, указанное в параметре метода **awaitTermination**.

# ScheduledExecutorService

ScheduledExecutorService мы используем тогда, когда хотим установить расписание на запуск потоков из пула.

Данный pool создаётся, используя factory метод класса Executors:

```
Executors.newScheduledThreadPool(int count)
```

schedule

scheduleAtFixedRate

scheduleWithFixedDelay

# Интерфейсы Callable и Future

Callable, также как и Runnable, представляет собой определённое задание, которое выполняется потоком.

В отличии от Runnable Callable:

- имеет return type не void;
- может выбрасывать Exception.

Метод **submit** передаёт наше задание (task) в thread pool, для выполнения его одним из потоков, и возвращает тип Future, в котором и хранится результат выполнения нашего задания.

Метод **get** позволяет получить результат выполнения нашего задания из объекта Future.

# Semaphore

Semaphore – это синхронизатор, позволяющий ограничить доступ к какому-то ресурсу. В конструктор Semaphore нужно передавать количество потоков, которым Semaphore будет разрешать одновременно использовать этот ресурс.

acquire

release

# CountDownLatch

CountDownLatch – это синхронизатор, позволяющий любому количеству потоков ждать пока не завершится определённое количество операций. В конструктор CountDownLatch нужно передавать количество операций, которые должны завершиться, чтобы потоки продолжили свою работу.

await

countDown

getCount

# Exchanger

Exchanger – это синхронизатор, позволяющий обмениваться данными между двумя потоками, обеспечивает то, что оба потока получают информацию друг от друга одновременно.

exchange

# AtomicInteger

AtomicInteger – это класс, который предоставляет возможность работать с целочисленным значением `int`, используя атомарные операции.

`incrementAndGet`

`getAndIncrement`

`addAndGet`

`getAndAdd`

`decrementAndGet`

`getAndDecrement`



# Коллекции для работы с многопоточностью



## Synchronized collections

Получаются из  
традиционных коллекций  
благодаря их обёртыванию

## Concurrent collections

Изначально созданы для  
работы с многопоточностью

`Collections.synchronizedXYZ(коллекция)`

# ConcurrentHashMap

ConcurrentHashMap implements the `ConcurrentMap` interface, which in turn inherits from the `Map` interface.

In `ConcurrentHashMap`, any number of threads can read elements without blocking.

In `ConcurrentHashMap`, due to its segmentation, when any element is modified, only the `bucket` it belongs to is blocked.

In `ConcurrentHashMap`, neither `key` nor `value` can be `null`.

# CopyOnWriteArrayList

CopyOnWriteArrayList имплементирует интерфейс List.

CopyOnWriteArrayList следует использовать тогда, когда вам нужно добиться потокобезопасности, у вас небольшое количество операций по изменению элементов и большое количество по их чтению.

В CopyOnWriteArrayList при каждой операции по изменению элементов создаётся копия этого листа.

# ArrayBlockingQueue

ArrayBlockingQueue – потокобезопасная очередь с ограниченным размером (capacity restricted).

Обычно один или несколько потоков добавляют элементы в конец очереди, а другой или другие потоки забирают элементы из начала очереди.

put

take

# Character Streams and Byte Streams

Stream (поток) для работы с файлами – это упорядоченная последовательность данных

Файлы разделяют на

- читабельные для человека – text files;
- нечитабельные для человека – binary files.

При работе с текстовыми и бинарными файлами нам необходимо использовать разные типы стримов.

# FileReader & FileWriter

FileReader и FileWriter используются для работы с текстовыми файлами.

```
FileWriter writer = new FileWriter("file1.txt");
```

```
FileReader reader= new FileReader("file1.txt");
```

Никогда не забывайте закрывать стримы после использования.

# Try with resources

```
try(    FileWriter writer = new FileWriter("file1.txt");
        FileReader reader= new FileReader("file1.txt");    )
{
    //SOME CODE...
}
```

Ресурс, который используется в Try with resources должен имплементировать интерфейс AutoCloseable

# BufferedReader & BufferedWriter

Использование буферизации в стримах позволяет достичь большей эффективности при чтении файла или записи в него.

```
BufferedWriter writer =  
new BufferedWriter (new FileWriter("file1.txt"));
```

```
BufferedReader reader =  
new BufferedReader (new FileReader("file1.txt"));
```



# FileInputStream & FileOutputStream

FileInputStream и FileOutputStream используются для работы с бинарными файлами.

```
FileInputStream inputStream =  
new FileInputStream("test2.bin");
```

```
FileOutputStream outputStream =  
new FileOutputStream("test2.bin");
```

# DataInputStream & DataOutputStream

DataInputStream и DataOutputStream позволяют записывать в файл и читать из него примитивные типы данных.

```
DataInputStream inputStream =  
new DataInputStream (new FileInputStream("test2.bin"));
```

```
DataOutputStream outputStream =  
new DataOutputStream (new FileOutputStream("test2.bin"));
```

# Serialization

Сериализация - это процесс преобразования объекта в последовательность байт.

Десериализация - это процесс восстановления объекта, из этих байт.

```
ObjectInputStream inputStream =  
new ObjectInputStream (new FileInputStream("test2.bin"));
```

```
ObjectOutputStream outputStream =  
new ObjectOutputStream (new FileOutputStream("test2.bin"));
```

# Serialization

Для того, чтобы объект класса можно было сериализовать, класс должен имплементировать интерфейс `Serializable`.

Поля класса, помеченные ключевым словом `transient`, не записываются в файл при сериализации.

В сериализируемом классе необходимо использовать `serialVersionUID` для обозначения версии класса.

# RandomAccessFile

Класс RandomAccessFile позволяет читать информацию из любого места файла и записывать информацию в любое место файла.

```
RandomAccessFile file =  
new RandomAccessFile ("test1.txt", "rw");
```

# Класс File

Класс File позволяет управлять информацией о файлах и директориях.

```
File file = new File ("test1.txt");
```

getAbsolutePath

isAbsolute

isDirectory

exists

createNewFile

mkdir

length

delete

listFiles

isHidden

canRead

canWrite

canExecute

# Buffers and Channels

Buffer – Это блок памяти, в который мы можем записывать информацию, а также читать её.

В отличии от стримов Channel может как читать файл, так и записывать в него.

Чтении файла: Channel читает информацию из файла и записывает в Buffer.

Запись в файл: Channel читает информацию из Buffer и записывает её в файл.

# Buffers and Channels

```
FileChannel channel = file.getChannel();
```

```
ByteBuffer buffer = ByteBuffer.allocate(100);
```

```
channel.read(buffer)
```

```
buffer.flip()
```

```
buffer.hasRemaining()
```

```
buffer.get()
```

```
buffer.clear()
```

```
buffer.put( ... )
```

```
channel.write(buffer)
```

```
buffer.rewind()
```

```
buffer.compact()
```

```
buffer.mark()
```

```
buffer.reset()
```



# Interface Path & class Files

Объект типа Path представляет собой путь к файлу или директории.

```
Path path = Paths.get("text1.txt");
```

```
path.getFileName()
```

```
path.getParent()
```

```
path.getRoot()
```

```
path.is Absolute()
```

```
path.toAbsolutePath()
```

```
path1.resolve(path2)
```

```
path1.relativeize(path2)
```

# Interface Path & class Files

`Files.exists(path)`

`Files.createFile(path)`

`Files.createDirectory(path)`

`Files.isReadable()`

`Files.isWritable()`

`Files.isExecutable()`

`Files.isSameFile(path1, path2)`

`Files.size()`

`Files.getAttribute(path, attribute_name)`

`Files.readAttributes(path, attributes)`

# Interface Path & class Files

```
Files.copy(path1, path2, copy_options)
```

```
Files.move(path1, path2, copy_options)
```

```
Files.delete(path)
```

```
Files.write(path, byte_array)
```

```
Files.readAllLines(path)
```

# Files.walkFileTree

Метод `Files.walkFileTree(Path start, FileVisitor visitor)` используется для обхода дерева файлов.

Логика обхода дерева файлов заключается в классе, имплементирующем интерфейс `FileVisitor`.

**preVisitDirectory** - срабатывает перед обращением к элементам папки;

**visitFile** - срабатывает при обращении к файлу;

**postVisitDirectory** - срабатывает после обращения ко всем элементам папки;

**visitFileFailed** - срабатывает когда файл по каким-то причинам недоступен.

# enum FileVisitResult

Значения FileVisitResult:

**CONTINUE** – означает, что нужно продолжать обход по файлам;

**TERMINATE** – означает, что нужно немедленно прекратить обход по файлам;

**SKIP\_SUBTREE** – означает, что в данную директорию заходить не надо;

**SKIP\_SIBLINGS** – означает, в данной директории продолжать обход по файлам не нужно.

# REGular EXpressions

Регулярные выражения необходимы для создания шаблонов, с помощью которых производят такие операции, как поиск, сравнение, замена.

Регулярные выражения - это совокупность символов, некоторые из которых являются специальными - метасимволами, т.е. обладают каким-то функционалом.

# Часто используемые символы в REGEX

**abc** – Соответствует последовательно идущим abc

**[abc]** – Соответствует или a, или b, или c

**[d-j]** – Соответствует одной из букв из диапазона d - j

**[3-8]** – Соответствует одной из цифр из диапазона 3-8

**[B-Fd-j3-8]** – Соответствует одной из букв из обоих диапазонов или одной из цифр из диапазона 3 - 8

**a|b** – Соответствует либо букве a, либо букве b

# Часто используемые символы в REGEX

**[^d-j]** – Данный символ, стоящий в начале этих скобок, означает отрицание. Соответствует одной из букв НЕ из диапазона d - j

**.** – Соответствует одному любому символу.  
Исключение: символ новой строки

**^**выражение – Соответствует выражению в начале строки

выражение**\$** – Соответствует выражению в конце строки



## Часто используемые МЕТАсимволы в REGEX

**\d** – Соответствует одной цифре

**\D** – Соответствует одной НЕ цифре

**\w** – Соответствует одной букве, цифре или «\_»

**\W** – Соответствует одному символу, который НЕ буква, НЕ цифра и НЕ «\_»

**\s** – Соответствует пробельному символу

**\S** – Соответствует НЕ пробельному символу

## Часто используемые METАсимволы в REGEX

**\A** – Соответствует выражению в начале String-а

**\Z** – Соответствует выражению в конце String-а

**\b** – Соответствует границе слова или числа

**\B** – Соответствует границе НЕ слова и НЕ числа

# Часто используемые символы в REGEX, обозначающие количество повторений

**выражение?** – Соответствует 0 или 1 повторению

**выражение\*** – Соответствует 0 или большему количеству повторений

**выражение+** – Соответствует 1 или большему количеству повторений

**выражение{n}** – Соответствует количеству повторений «n»

**выражение{m, n}** – Соответствует количеству повторений от «m» до «n»

**выражение{n,}** – Соответствует n или большему количеству повторений

# Классы Pattern и Matcher

```
Pattern myPattern = Pattern.compile("«REGEX»");
```

```
Matcher myMatcher = myPattern.matcher(myString);
```

Пакет: `java.util.regex`

## Методы matches и split

```
public boolean matches (String regex)
```

```
public String [] split (String regex)
```

# Методы printf и format

`%[flags][width][.precision]datatype_specifier`

## flags

«-» выравнивание по  
левому краю

«0» добавление нулей  
перед числом

«,» разделитель  
разрядов в числах

## DT specifiers

«b» boolean

«c» character

«s» String

«d» целое  
число

«f» десятичное  
число

# enum

enum – это способ ограничения определённого рода информации конкретным списком возможных вариантов

Конструктор в enum имеет access modifier private и не нуждается во внешнем вызове

enum является дочерним классом для `java.lang.Enum`

Часто используемые методы: `valueOf`, `values`

# Scanner

```
Scanner sc = new Scanner(System.in); //работа с консолью
```

nextByte()

nextShort()

nextInt()

nextLong()

nextFloat()

nextDouble()

nextBoolean()

next()

nextLine()

hasNext()

hasNextLine()

hasNextInt()

.....

.....

.....



# Reflection

Рефлексия - это механизм исследования данных о программе во время её выполнения.

Рефлексия позволяет исследовать информацию о полях, методах, конструкторах и других составляющих классов.

```
Class clazz = Class.forName("package_name.Class_name");
```

```
Class clazz = Class_name.class;
```

```
Class_name c = new Class_name();  
Class clazz = c.getClass();
```

# Reflection methods

getField

getType

getFields

getName

getDeclaredFields

getMethod

getReturnType

getParameterTypes

getMethods

getDeclaredMethods

isPublic

getModifiers

getConstructor

getParameterCount

getConstructors

newInstance

invoke

setAccessible

get

set

# Annotation

Аннотации – это специальные комментарии/метки/метаданные, которые нужны для передачи определённой информации.

@Target показывает область кода, к которой Аннотация может быть применима. Самые распространённые области кода:

- TYPE – class, interface, enum;
- FIELD – поле класса;
- METHOD – метод класса;
- PARAMETER – параметры метода или конструктора

# Annotation

@Retention описывает жизненный цикл

Аннотации

- SOURCE – Аннотация видна в source коде, отбрасывается компилятором и уже в byte коде не видна;
- CLASS – Аннотация видна в byte коде, отбрасывается JVM во время выполнения программы;
- RUNTIME – Аннотация видна во время выполнения программы