7. Дженерики

1. Что такое дженерики?

Generics - набор свойств языка позволяющих определять и использовать обобщенные типы и методы. Обобщенные типы или методы отличаются от обычных тем, что имеют типизированные параметры (T – параметр в котором могут быть разные объекты).

```
ArrayList<Integer> list = new ArrayList<Integer>();
```

Свойства Generics:

- Строгая типизация.
- Единая реализация.
- Отсутствие информации о типе.

Ограничения:

- - в <> могут быть только ссылочные типы, можно String, Integer, int [] array, ~~int, double, 10;~~
- - внутри класса или метода нельзя создать экземпляр класса, массив или использовать метод equals, так как внутри класса не известно о самом классе – компилятор не может понять тип
- ограничение наследования
  Пусть у нас есть тип Foo, который является подтипом Bar, и еще G - наследник Коллекции. То G<Foo> не является наследником G<Bar>.
- не может быть параметром статического класса
- Не может Создать, Поймать, или Бросить Объекты Параметризованных Типов
  ```
  class MathException<T> extends Exception { /* ... */ }    // compile-time error
  ```
- У class не может быть двух перегруженных методов, у которых будет та же самая подпись после стирания типа.

```
public class Example {
   public void print(Set<String> strSet) { }
   public void print(Set<Integer> intSet) { }
}
```

Примером использования обобщенных типов может служить Java Collection Framework. Так, класс LinkedList<E> - типичный обобщенный тип. Он содержит параметр E, который представляет тип элементов, которые будут храниться в коллекции. Создание объектов обобщенных типов происходит посредством замены параметризированных типов реальными типами данных. Вместо того, чтобы просто использовать LinkedList, ничего не говоря о типе элемента в списке, предлагается использовать точное указание типа LinkedList<String>, LinkedList<Integer> и т.п.

2. Для чего нужны дженерики?

Позволяет осуществлять проверку на правильность написания кода во время компиляции, а не в Runtime. Возможность создавать универсальные алгоритмы и структуры данных.
Сделали использование Java Collection Framework проще, удобнее и безопаснее
С их помощью можно объявлять классы, интерфейсы и методы, где тип данных указан в виде параметра. generics (обобщенные типы и методы) позволяют нам уйти от жесткого определения используемых типов.
Компилятор стирает все дженерики. В Runtime дженериков практически нет. Компилятор использует кастование

3. Что такое сырые типы (raw type)?

Сырые типы — это типы без указания "уточненения" в фигурных скобках. Нужны чтобы поддерживать старый код (обратная совместимость). Рекомендуется использовать хоть какие-то параметризованные типы.
Raw type - это имя интерфейса без указания параметризованного типа:
Также Diamond синтаксис связан с понятием "Type Inference", или же выведение типов. Ведь компилятор, видя справа <> смотрит на левую часть, где расположено объявление типа переменной, в

которую присваивается значение. И по этой части понимает, каким типом типизируется значение справа.
List<String> list = new ArrayList<>();
На самом деле, если в левой части указан дженерик, а справа не указан <>, компилятор сможет вывести тип. Однако это будет смешиванием нового стиля с дженериками и старого стиля без них - вы теряете безопасность (типобезопасность) типов (может добавляться какой угодно тип в список, а не то что надо. Когда будет доставаться – то может быть сюрприз)))

ArrayList<String> strings = new ArrayList<>(); // parameterized type
ArrayList arrayList = new ArrayList(); // raw type
arrayList = strings; // Ok
strings = arrayList; // Unchecked assignment (назначение)
arrayList.add(1); //unchecked call

4. Что такое вайлдкарды (Маски)?
Языковая конструкция внутри даймонд-оператора, позволяющая сделать код более универсальным. Решает проблему наследования типов в дженериках. (коллекция <Интежер> не наследник коллекции <Намбер>)
Может быть 3-х типов: инвариантность, аппер и ловер

Class: Cat / Dog -> Pet -> Animal -> Object
<? extends Animal> - тип ? является любой наследник Animal (Upper Bounded (ограничение сверху) Wildcards)
<? super Cat> - тип ? является любой родитель Cat, включая Cat (Lower Bounded Wildcards). Dog не подходит.

List<? extends Number> numberList = new ArrayList<>();
List<Integer> integerList = new ArrayList<>();
numberList = integerList;
integerList.add(5); // норм
numberList.add(5); // ошибка компиляции
 	
public void process (List <? extends Number> list){
list.add(5L); // ошибка компиляции
}
	Означает, что мы можем присвоить в numberList листы <Number и наследники>:
	new ArrayList <Integer> // это компилятор не видит
	new ArrayList <Long> // не видит
	Не положить туда Number и наследники, а присвоить!!
	Туда безопасно можем положить только null.
	Компилятор стирает дженерик типы и видит List<? extends Number>.
	Т.е. безопасно будет:
	List<? extends Number> numberList = new Number<>();
	Компилятор видит будещее, что если мы положим в list Лонг, а затем можем положить туда Интежер – будет не безопасно. Он кладет только то, что безопасно – null

List<? super Number> numberList = new ArrayList<>(); // можно ложить в лист Интежер, Лонг и т.п., нельзя Обжект	Компилятор видит будущее, мы можем присвоить сюда Листы Намберов и Обжектов. Если мы положим в Лист Намберов Интежер, Лонг, Дабл – то ошибки не будет, а если положить туда Обжект – ошибка компиляции

5. Расскажите про принцип PECS
The Get and Put Principle или PECS (Producer Extends Consumer Super) Get and Put Principe
Из одного типа переменных можно только читать, в другой — только вписывать (исключением является возможность записать null для extends и прочитать Object для super).
- <?> Запись вида Collection<?> равносильна Collection<? extends Object> , а значит — коллекция может содержать объекты любого класса, так как все классы в Java наследуются от Object – поэтому подстановка называется неограниченной.
- Ковариация - Если мы объявили wildcard с extends, то это producer. Он только «продюсирует», предоставляет элемент из контейнера, а сам ничего не принимает.

- Контрвариация - Если же мы объявили wildcard с super — то это consumer. Он только принимает, а предоставить ничего не может.

Если метод имеет аргументы с параметризованным типом (например, Collection или Predicate), то в случае, если аргумент - производитель (producer), нужно использовать ? extends T, а если аргумент - потребитель (consumer), нужно использовать ? super T.
Если метод читает данные из аргумента, то этот аргумент - производитель,
Метод передаёт данные в аргумент, то аргумент является потребителем. Важно заметить, что определяя производителя или потребителя, мы рассматриваем только данные типа T.

// Ковариантность, значит producer - аргумент nums отдает (производит) в метод ресурсы
public static double sum(Collection<? extends Number> nums) {
double s = 0.0;
for (Number num : nums)
s += num.doubleValue();
return s;}
	List<Integer>ints = Arrays.asList(1,2,3);
	assert sum(ints) == 6.0; // Integer наследник Nums
	List<Double>doubles = Arrays.asList(2.78,3.14);
	assert sum(doubles) == 5.92;
	List<Number>nums = Arrays.<Number>asList(1,2,2.78,3.14);
	assert sum(nums) == 8.92;

//Контрвариантность - если аргумент метода потребляет в методе что-то /(возможно, изменяется сам)
 	
public static void count(Collection<? super Integer> ints, int n) {
for (int i = 0; i < n; i++) ints.add(i);
}
	List<Integer>ints = new ArrayList<Integer>();
	count(ints, 5);
	assert ints.toString().equals("[0, 1, 2, 3, 4]");
	List<Number>nums = new ArrayList<Number>();
	count(nums, 5); nums.add(5.0);
	assert nums.toString().equals("[0, 1, 2, 3, 4, 5.0]");
	List<Object>objs = new ArrayList<Object>();
	count(objs, 5); objs.add("five");
	assert objs.toString().equals("[0, 1, 2, 3, 4, five]");

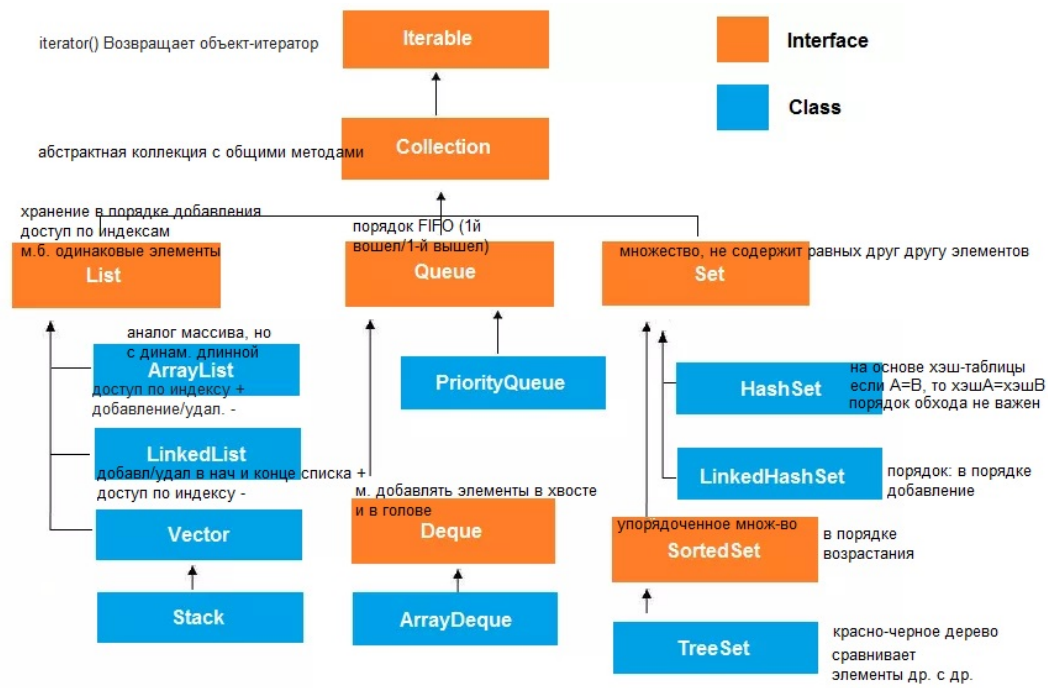
Пример: метод copy () в классе Collectors

Ковариантность — это сохранение иерархии наследования исходных типов в производных типах в том же порядке
Множество<Животные> = Множество<Кошки>
Контравариантность — это обращение иерархии исходных типов на противоположную в производных типах
Множество<Кошки> = Множество<Животные>
Инвариантность — отсутствие наследования между производными типами.

8. Коллекции

1. Что такое «коллекция»?
«Коллекция» - это структура данных, набор каких-либо объектов. Данными (объектами в наборе) могут быть числа, строки, объекты пользовательских классов и т.п.
Коллекции (java) – набор классов и интерфейсов, предназначенных для хранения данных. Каждый класс имеет свою специфику хранения объектов. Классы коллекций являются дженерик-параметрами.

2. Расскажите про иерархию коллекций



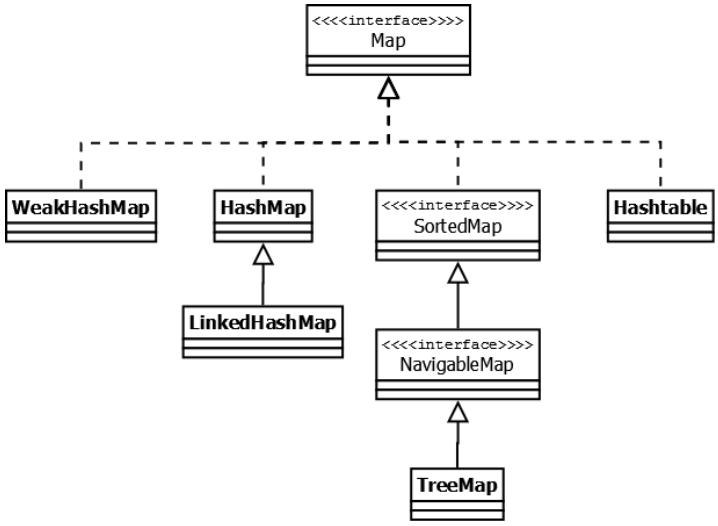
Интерфейс Collection расширяют интерфейсы:

- List (список) представляет собой коллекцию, в которой допустимы дублирующие значения. Элементы такой коллекции пронумерованы, начиная от нуля, к ним можно обратиться по индексу. Реализации:
  - ArrayList - инкапсулирует в себе обычный массив, длина которого автоматически увеличивается при добавлении новых элементов.
  - LinkedList (двунаправленный связный список) - состоит из узлов, каждый из которых содержит как собственно данные, так и две ссылки на следующий и предыдущий узел.
  - Vector — реализация динамического массива объектов, методы которой синхронизированы (доступны по очереди, т.е. может использовать кто-то один).
  - Stack — реализация стека LIFO (last-in-first-out).
- Set (сет) описывает неупорядоченную коллекцию, не содержащую повторяющихся элементов. Реализации:
  - HashSet - использует HashMap для хранения данных. В качестве ключа и значения используется добавляемый элемент. Из-за особенностей реализации порядок элементов не гарантируется при добавлении.
  - LinkedHashSet — гарантирует, что порядок элементов при обходе коллекции будет идентичен порядку добавления элементов.
  - (SortedSet) TreeSet — предоставляет возможность управлять порядком элементов в коллекции при помощи объекта Comparator, либо сохраняет элементы с использованием «natural ordering».
- Queue (очередь) предназначена для хранения элементов с предопределённым способом вставки и извлечения FIFO (first-in-first-out):
  - PriorityQueue — предоставляет возможность управлять порядком элементов в коллекции при помощи объекта Comparator, либо сохраняет элементы с использованием «natural ordering».
  - ArrayDeque — реализация интерфейса Deque, который расширяет интерфейс Queue методами, позволяющими реализовать конструкцию вида LIFO (last-in-first-out).

Интерфейс Map реализован классами:

- Hashtable — хэш-таблица, методы которой синхронизированы. Не позволяет использовать null в качестве значения или ключа и не является упорядоченной.
- HashMap — хэш-таблица. Позволяет использовать null в качестве значения или ключа и не является упорядоченной.
- LinkedHashMap — упорядоченная реализация хэш-таблицы.
- TreeMap — реализация, основанная на красно-чёрных деревьях. Является упорядоченной и предоставляет возможность управлять порядком элементов в коллекции при помощи объекта Comparator, либо сохраняет элементы с использованием «natural ordering».

- WeakHashMap — реализация хэш-таблицы, которая организована с использованием weak references для ключей (сборщик мусора автоматически удалит элемент из коллекции при следующей сборке мусора, если на ключ этого элемента нет жёстких ссылок).



3. Почему Map — это не Collection, в то время как List и Set являются Collection?
Collection представляет собой совокупность некоторых элементов. Map - это совокупность пар «ключ-значение».

4. В чем разница между классами java.util.Collection и java.util.Collections?
Collections - набор статических методов для работы с коллекциями.
Collection - один из основных интерфейсов Java Collections Framework

5. Какая разница между итераторами с fail-fast и fail-safe поведением? (С примерами)
fail-fast поведение означает, что при возникновении ошибки или состояния, которое может привести к ошибке, система немедленно прекращает дальнейшую работу и уведомляет об этом. Использование fail-fast подхода позволяет избежать недетерминированного поведения программы в течение времени.
В Java Collections API некоторые итераторы ведут себя как fail-fast и выбрасывают ConcurrentModificationException, если после его создания была произведена модификация коллекции, т.е. добавлен или удален элемент напрямую из коллекции, а не используя методы итератора.
Реализация такого поведения осуществляется за счет подсчета количества модификаций коллекции (modification count):
- при изменении коллекции счетчик модификаций так же изменяется;
- при создании итератора ему передается текущее значение счетчика;
- при каждом обращении к итератору сохраненное значение счетчика сравнивается с текущим, и, если они не совпадают, возникает исключение.
Итераторы по умолчанию для Collections из java.util package , такие как ArrayList , HashMap и т. д., Являются Fail-Fast

```
ArrayList<Integer> numbers =//...
Iterator<Integer> iterator = numbers.iterator();
while (iterator.hasNext()) {
    Integer number = iterator.next();
    numbers.add(50);
}
```

В приведенном выше фрагменте кода ConcurrentModificationException генерируется в начале следующего цикла итерации после выполнения модификации.

итераторы fail-safe не вызывают никаких исключений при изменении структуры, потому что они работают с клоном коллекции вместо оригинала.
Итератор коллекции CopyOnWriteArrayList и итератор представления keySet коллекции ConcurrentHashMap являются примерами итераторов fail-safe.

6. Чем различаются Enumeration (устаревший) и Iterator?
Хотя оба интерфейса и предназначены для обхода коллекций между ними имеются существенные различия:
- с помощью Enumeration нельзя добавлять/удалять элементы;
- в Iterator исправлены имена методов для повышения читаемости кода (Enumeration.hasMoreElements() соответствует Iterator.hasNext(), Enumeration.nextElement() соответствует Iterator.next() и т.д);
- Enumeration присутствуют в устаревших классах, таких как Vector/Stack, тогда как Iterator есть во всех современных классах-коллекциях.

7. Как между собой связаны Iterable, Iterator и «for-each»?
Интерфейс Iterable имеет только один метод - iterator(), который возвращает объект-Iterator.

Iterable<T>
Iterator<T>iterator() Возвращает объект-итератор.
Класс Iterator отвечает за безопасный проход по списку элементов, имеет всего 3 метода:
hasNext() — возвращает true или false в зависимости от того, есть ли в списке следующий элемент, или мы уже дошли до последнего.
next() — возвращает следующий элемент списка
remove() — удаляет элемент из списка

Классы, реализующие интерфейс Iterable, могут применяться в конструкции for-each, которая использует Iterator.

```
// Iterating over collection 'c' using Iterator
  for (Iterator i = c.iterator(); i.hasNext(); )
     System.out.println(i.next());
```

8. Можно ли итерируясь по ArrayList удалить элемент? Какое вылетит исключение?
Если не через iterator, например через for each - то будет исключение
В Java для удаления элементов во время перебора нужно использовать специальный объект — итератор (класс Iterator)

```
for (Cat cat: cats) {

    if (cat.name.equals("Бегемот")) {
        cats.remove(cat);
    }
}
```

`java.util.ConcurrentModificationException`

```
Iterator<Cat> catIterator = cats.iterator();//создаем итератор
while(catIterator.hasNext()) {//до тех пор, пока в списке есть элементы

  Cat nextCat = catIterator.next();//получаем следующий элемент
  System.out.println(nextCat);//выводим его в консоль
}
```

9. Как поведёт себя коллекция, если вызвать iterator.remove()?
Если вызову iterator.remove() предшествовал вызов iterator.next(), то iterator.remove() удалит элемент коллекции, на который указывает итератор, в противном случае будет выброшено IllegalStateException().

```
Iterator<String> it = names.iterator();
while (it.hasNext()) {
  String el = it.next();
  if (el.equals("")) {
```

```
for (String el: names) {
    if (el.equals("")) {
        names.remove(el); // WRONG!
    }
```

it.remove(); // it's norm
}
	}

10. Чем Set отличается от List?

Set описывает неупорядоченную коллекцию, не содержащую повторяющихся элементов
List представляет собой коллекцию, в которой допустимы дублирующие значения. Элементы такой коллекции пронумерованы, начиная от нуля, к ним можно обратиться по индексу.

11. Расскажите про интерфейс Set.

Представляет собой неупорядоченную коллекцию, которая не может содержать дублирующиеся данные. Является программной моделью математического понятия «множество».
Set не добавляет новых методов, только вносит изменения в унаследованные.
Множество, как и список, и любую коллекцию (и не только) можно обойти в цикле for-each:

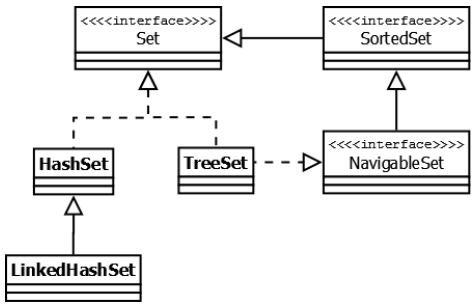
```
for (Integer number : intSet) {
    System.out.println(number);
}
```

Но вот доступ по индексу (порядковому номеру) (метод list.get(int index)) есть только у List, у Set этого метода нет. Это связано с тем, что порядок элементов во множестве не определён.

Интерфейс **Set** включает следующие методы:

Метод	Описание
add(Object o)	Добавление элемента в коллекцию, если он отсутствует. Возвращает true, если элемент добавлен.
addAll(Collection c)	Добавление элементов коллекции, если они отсутствуют.
clear()	Очистка коллекции.
contains(Object o)	Проверка присутствия элемента в наборе. Возвращает true, если элемент найден.
containsAll(Collection c)	Проверка присутсвия коллекции в наборе. Возвращает true, если все элементы содержатся в наборе.
equals(Object o)	Проверка на равенство.
hashCode()	Получение hashCode набора.
isEmpty()	Проверка наличия элементов. Возвращает true если в коллекции нет ни одного элемента.
iterator()	Функция получения итератора коллекции.
remove(Object o)	Удаление элемента из набора.
removeAll(Collection c)	Удаление из набора всех элементов переданной коллекции.
retainAll(Collection c)	Удаление элементов, не принадлежащих переданной коллекции.
size()	Количество элементов коллекции
toArray()	Преобразование набора в массив элементов.
toArray(T[] a)	Преобразование набора в массив элементов. В отличии от предыдущего метода, который возвращает массив объектов типа Object, данный метод возвращает массив объектов типа, переданного в параметре.

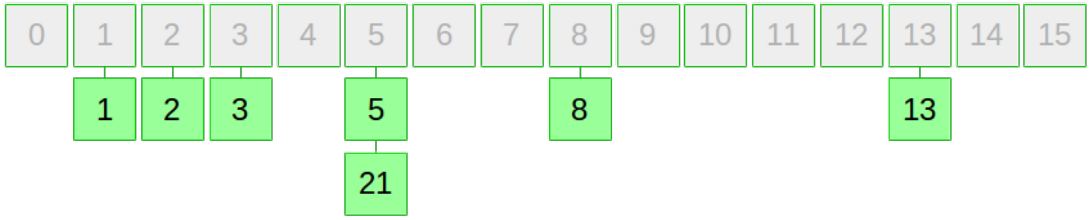
12. Расскажите про реализации интерфейса Set



В множествах Set каждый элемент хранится только в одном экземпляре, а разные реализации Set используют разный порядок хранения элементов.
Все классы, реализующие интерфейс Set, внутренне поддерживаются реализациями Map.
В HashSet порядок элементов определяется по сложному алгоритму.
TreeSet - объекты хранятся отсортированными по возрастанию в порядке сравнения
LinkedHashSet - хранение элементов в порядке добавления.
Множества часто используются для проверки принадлежности, чтобы вы могли легко проверить, принадлежит ли объект заданному множеству, поэтому на практике обычно выбирается реализация HashSet, оптимизированная для быстрого поиска.

HashSet
Класс HashSet реализует интерфейс Set, основан на хэш-таблице, а также поддерживается с помощью экземпляра HashMap. В HashSet элементы не упорядочены, нет никаких гарантий, что элементы будут в том же порядке спустя какое-то время.
HashSet хранит элементы с помощью HashMap. Хоть и для добавления элемента в HashMap он должен быть представлен в виде пары «ключ-значение», в HashSet добавляется только значение.
Операции добавления, удаления и поиска будут выполняться за константное время при условии, что хэш-функция правильно распределяет элементы по «корзинам».

Хэш-функция — это функция, сужающая множество значений объекта до некоторого подмножества целых чисел. Класс Object имеет метод hashCode(), который используется классом HashSet для эффективного размещения объектов, заносимых в коллекцию. В классах объектов, заносимых в HashSet, этот метод должен быть переопределен (override).
HashSet хранит элементы таким образом, чтобы элемент можно было очень быстро найти.
Методы contains и indexOf у ArrayList проходят последовательно по элементам списка в поисках нужного элемента, а метод contains у HashSet ищет элемент многократно быстрее, так как использует совсем другой подход: элементы находятся в так называемых «корзинах» (на иллюстрации — 16 серых корзин), которые выбираются исходя из значений самих элементов (на иллюстрации — 7 зелёных элементов множества).



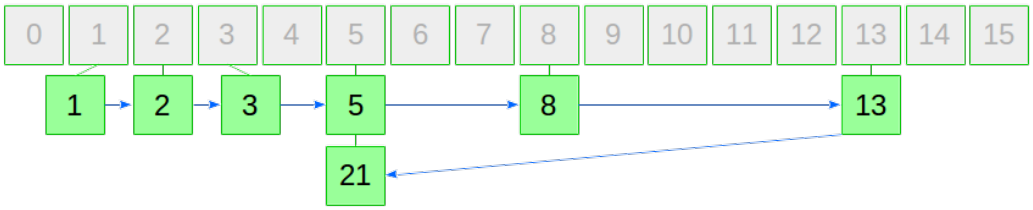
Конструкторы HashSet :
// Создание пустого набора с начальной емкостью (16) и со
// значением коэффициента загрузки (0.75) по умолчанию
public HashSet();
// Создание множества из элементов коллекции

```
public HashSet(Collection c);
// Создание множества с указанной начальной емкостью и со
// значением коэффициента загрузки по умолчанию (0.75)
public HashSet(int initialCapacity);
// Создание множества с указанными начальной емкостью и по умолчанию 16
// коэффициентом загрузки по умолчанию 0.75
public HashSet(int initialCapacity, float loadFactor);
```

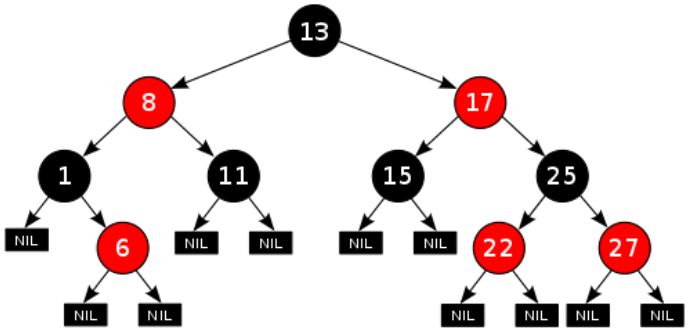
Методы аналогичны методам ArrayList за исключением того, что метод add(Object o) добавляет объект в множество только в том случае, если его там нет. Возвращаемое методом значение — true, если объект добавлен, и false, если нет.
Порядок добавления объектов во множество будет непредсказуемым. HashSet использует хэширование для ускорения выборки. Если вам нужно, чтобы результат был отсортирован, то пользуйтесь TreeSet.
Реализация HashSet не синхронизируется. Если многократные потоки получают доступ к набору хеша одновременно, а один или несколько потоков должны изменять набор, то он должен быть синхронизирован внешне. Это лучше всего выполнить во время создания, чтобы предотвратить случайный несинхронизируемый доступ к набору :

```
Set<E> set = Collections.synchronizedSet(
                                new HashSet<E>());
```

LinkedHashSet
Класс LinkedHashSet расширяет класс HashSet, не добавляя никаких новых методов. Класс поддерживает связный список элементов набора в том порядке, в котором они вставлялись. Это позволяет организовать упорядоченную итерацию вставки в набор.



Также, как и HashSet, LinkedHashSet не синхронизируется. Поэтому при использовании данной реализации в приложении с множеством потоков, часть из которых может вносить изменения в набор, следует на этапе создания выполнить синхронизацию
TreeSet



Проще говоря, TreeSet это отсортированная коллекция, которая расширяет класс AbstractSet и реализует интерфейс NavigableSet.
Особенности:
- Хранит уникальные элементы
- Не сохраняет порядок вставки элементов
- Сортирует элементы в порядке возрастания
- не потокобезопасный

- TreeSet не поддерживает добавление null, однако если бы вы написали свой собственный компаратор, который поддерживает null, то не было бы никаких проблем использовать null в качестве ключа
В этой реализации объекты сортируются и хранятся в порядке возрастания в соответствии с их естественным порядком. TreeSet использует самобалансирующееся двоичное дерево поиска.
По сравнению с HashSet производительность TreeSet находится на нижней стороне. Такие операции, как add , remove и search , занимают O (log n) время, в то время как такие операции, как печать n элементов в отсортированном порядке, требуют O (n) время.
Будучи самобалансирующимся двоичным деревом поиска, каждый узел двоичного дерева содержит дополнительный бит, который используется для определения цвета узла, который является красным или черным. Во время последующих вставок и удалений эти «цветные» биты помогают обеспечить более или менее сбалансированное дерево.
SortedSet
По умолчанию сортировка производится привычным способом, но можно изменить это поведение через интерфейс Comparable.

13. В чем отличия TreeSet и HashSet?
TreeSet обеспечивает упорядоченно хранение элементов в виде красно-черного дерева. Сложность выполнения основных операций не хуже O(log(N)) (Логарифмическое время).
HashSet использует для хранения элементов такой же подход, что и HashMap, за тем отличием, что в HashSet в качестве ключа и значения выступает сам элемент, кроме того HashSet не поддерживает упорядоченное хранение элементов и обеспечивает временную сложность выполнения операций аналогично HashMap.

14. Чем LinkedHashSet отличается от HashSet?
Класс LinkedHashSet расширяет класс HashSet, не добавляя никаких новых методов. Класс поддерживает связный список элементов набора в том порядке, в котором они вставлялись. Это позволяет организовать упорядоченную итерацию вставки в набор

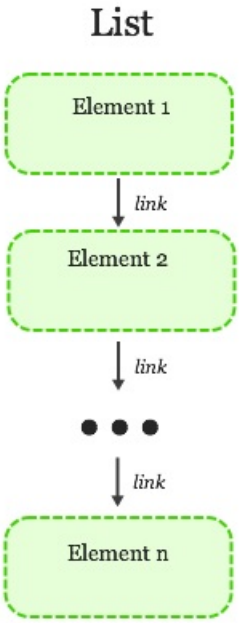
15. Что будет, если добавлять элементы в TreeSet по возрастанию?
такое же дерево, как и обычно.

16. Как устроен HashSet, сложность основных операций.
- Т.к. класс реализует интерфейс Set, он может хранить только уникальные значения;
- Может хранить NULL – значения;
- Порядок добавления элементов вычисляется с помощью хэш-кода;
- HashSet также реализует интерфейсы Serializable и Cloneable.
Для поддержания постоянного времени выполнения операций время, затрачиваемое на действия с HashSet, должно быть прямо пропорционально количеству элементов в HashSet + «емкость» встроенного экземпляра HashMap (количество «корзин»). Поэтому для поддержания производительности очень важно не устанавливать слишком высокую начальную ёмкость (или слишком низкий коэффициент загрузки).
Коэффициент загрузки = Количество хранимых элементов в таблице / размер хэш-таблицы
Например, если изначальное количество ячеек в таблице равно 16, и коэффициент загрузки равен 0,75, то из этого следует, что когда количество заполненных ячеек достигнет 12, их количество автоматически увеличится.
не является структурой данных с встроенной синхронизацией, поэтому если с ним работают одновременно несколько потоков, и как минимум один из них пытается внести изменения, необходимо обеспечить синхронизированный доступ извне.
Set s = Collections.synchronizedSet(new HashSet(...));

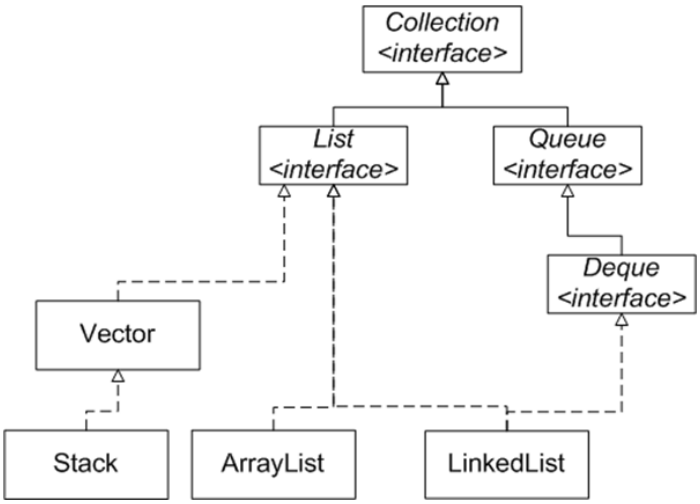
17. Как устроен LinkedHashSet, сложность основных операций.

18. Как устроен TreeSet, сложность основных операций.

19. Расскажите про интерфейс List



Это тип данных, в котором каждый элемент содержит какой-то объект, а также ссылку на следующий элемент списка.
Интерфейс List сохраняет последовательность добавления элементов и позволяет осуществлять доступ к элементу по индексу.
List добавляет следующие методы:
- void add(int index, E obj) - вставляет obj в вызывающий список в позицию, указанную в index. Любые ранее вставленные элементы за указанной позицией вставки смещаются вверх. То есть никакие элементы не перезаписываются.
- boolean addAll (int index,Collection<? extends E> c) - вставляет все элементы в вызывающий список, начиная с позиции, переданной в index. Все ранее существовавшие элементы за точкой вставки смещаются вверх. То есть никакие элементы не перезаписываются. Возвращает true, если вызывающий список изменяется, и false в противном случае.
- E get (int index) - возвращает объект, сохраненный в указанной позиции вызывающего списка.
int indexOf(Object obj) - возвращает индекс первого экземпляра obj в вызывающем списке. Если obj не содержится в списке, возвращается 1.
- int lastIndexOf(Object obj) - возвращает индекс последнего экземпляра obj в вызывающем списке. Если obj не содержится в списке, возвращается 1.
- ListIterator listIterator() - возвращает итератор, указывающий на начало списка.
- ListIterator listIterator(int index) - возвращает итератор, указывающий на заданную позицию в списке.
- E remove(int index) - удаляет элемент из вызывающего списка в позиции index и возвращает удаленный элемент. Результирующий список уплотняется, то есть элементы, следующие за удаленным, сдвигаются на одну позицию назад.
- E set (int index, E obj) - присваивает obj элементу, находящемуся в списке в позиции index.
default void sort(Comparator<? super E> c) - сортирует список, используя заданный компаратор (добавлен в версии JDK 8).
- List subList (int start, int end) - возвращает список, включающий элементы от start до end-1 из вызывающего списка. Элементы из возвращаемого списка также сохраняют ссылки в вызывающем списке.



20. Как устроен ArrayList, сложность основных операций.

ArrayList хранит элементы в динамическом массиве. Элементы ArrayList могут быть абсолютно любых типов в том числе и null.

Основное преимущество такой коллекции над массивом – это расширяемость – увеличение длины при надобности.

Если в этом массиве заканчивается место, то создаётся второй массив побольше, куда копируются все элементы из первого. Затем второй массив занимает место первого, а первый – выбрасывается (будет уничтожен сборщиком мусора).

Длина нового массива рассчитывается так (3*n)/2+1, где n – это длина старого массива. Т.е. если старый массив был длиной 100 элементов, то новый будет 300/2+1 = 151

При добавлении элемента в середину ArrayList, все элементы справа от него копируются на 1 позицию вправо, а затем в пустую ячейку добавляется новый элемент.

По возможности, избегайте операций вставки в середину коллекции. Ведь системе приходится заново пересчитывать индексы элементов.

При удалении элемента из середины, все элементы справа от него копируются на 1 позицию влево. Сам массив не укорачивается (можно укоротить через trimToSize()).

Особенности

— Быстрый доступ к элементам по индексу за время O(1);

— Доступ к элементам по значению за линейное время O(n);

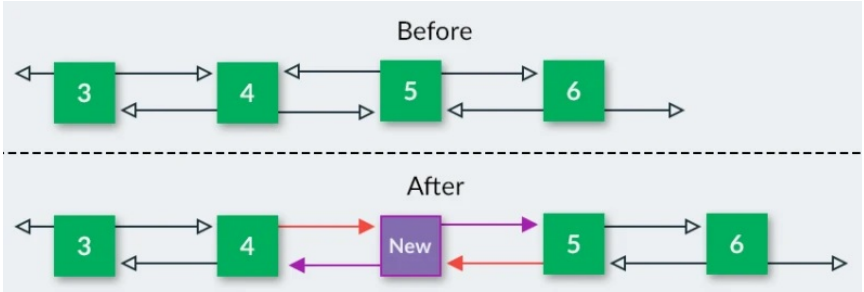
— Медленный, когда вставляются и удаляются элементы из «середины» списка;

— Позволяет хранить любые значения в том числе и null;

— Не синхронизирован.

21. Как устроен LinkedList, сложность основных операций.

Связный список состоит из одинаковых элементов, которые хранят данные и хранят ссылки на следующий и предыдущий элементы.



Вся работа с LinkedList сводится к изменению ссылок.
Однако, у LinkedList есть отдельные методы для работы с началом и концом списка, которых нет в ArrayList: addFirst(), addLast(): методы для добавления элемента в начало/конец списка
Вставка и удаление в середину LinkedList устроены гораздо проще, чем в ArrayList. Мы просто переопределяем ссылки соседних элементов, а ненужный элемент “выпадает” из цепочки ссылок.
Особенности:
— Из LinkedList можно организовать стэк, очередь, или двойную очередь, со временем доступа O(1) [константное время - поскольку выполняется единственная команда для его обнаружения];
— На вставку и удаление из середины списка, получение элемента по индексу или значению потребуется линейное время O(n) [линейное время - Например, процедура, суммирующая все элементы списка, требует время, пропорциональное длине списка]. Однако, на добавление и удаление из середины списка, используя ListIterator.add() и ListIterator.remove(), потребуется O(1);
— Позволяет добавлять любые значения в том числе и null. Для хранения примитивных типов использует соответствующие классы-оберки;
— Не синхронизирован.
LinkedList содержит все те методы, которые определены в интерфейсах List, Queue, Deque.
Некоторые из них:
addFirst() / offerFirst(): добавляет элемент в начало списка
addLast() / offerLast(): добавляет элемент в конец списка
removeFirst() / pollFirst(): удаляет первый элемент из начала списка
removeLast() / pollLast(): удаляет последний элемент из конца списка
getFirst() / peekFirst(): получает первый элемент
getLast() / peekLast(): получает последний элемент

22. Почему LinkedList реализует и List, и Deque?

LinkedList — класс, реализующий два интерфейса — ***List*** и ***Deque***. Это обеспечивает возможность создания двунаправленной очереди из любых (в том числе и *null*) элементов. Каждый объект, помещенный в связанный список, является узлом (нодом). Каждый узел содержит элемент, ссылку на предыдущий и следующий узел. Фактически связанный список состоит из последовательности узлов, каждый из которых предназначен для хранения объекта определенного при создании типа.

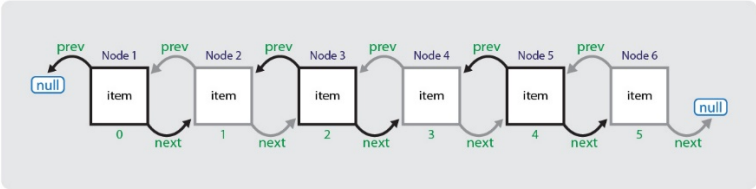


Рис. 1: Общий вид связанного списка

23. Чем отличаются ArrayList и LinkedList?

ArrayList это список, реализованный на основе массива, а LinkedList — это классический двусвязный список, основанный на объектах с ссылками между ними.

ArrayList:
- доступ к произвольному элементу по индексу за константное время O(1);
- доступ к элементам по значению за линейное время O(N);
- вставка в конец в среднем производится за константное время O(1);
- удаление произвольного элемента из списка занимает значительное время т.к. при этом все элементы, находящиеся «правее» смещаются на одну ячейку влево (реальный размер массива (capacity) не изменяется);
- вставка элемента в произвольное место списка занимает значительное время т.к. при этом все элементы, находящиеся «правее» смещаются на одну ячейку вправо;
- минимум накладных расходов при хранении.
LinkedList:
- на получение элемента по индексу или значению потребуется линейное время O(N);
- на добавление и удаление в начало или конец списка потребуется константное O(1);
- вставка или удаление в/из произвольного место константное O(1);
- требует больше памяти для хранения такого же количества элементов, потому что кроме самого элемента хранятся еще указатели на следующий и предыдущий элементы списка.
В целом, LinkedList в абсолютных величинах проигрывает ArrayList и по потребляемой памяти, и по скорости выполнения операций. LinkedList предпочтительно применять, когда нужны частые операции вставки/удаления или в случаях, когда необходимо гарантированное время добавления элемента в список.
Для ArrayList или для LinkedList операция добавления элемента в середину (list.add(list.size()/2, newElement)) медленнее?
Для ArrayList:
-проверка массива на вместимость. Если вместимости недостаточно, то увеличение размера массива и копирование всех элементов в новый массив (O(N));
-копирование всех элементов, расположенных правее от позиции вставки, на одну позицию вправо (O(N));
-вставка элемента (O(1)).
Для LinkedList:
-поиск позиции вставки (O(N));
-вставка элемента (O(1)).
В худшем случае вставка в середину списка эффективнее для LinkedList. В остальных - скорее всего, для ArrayList, поскольку копирование элементов осуществляется за счет вызова быстрого системного метода System.arraycopy().

24. Что такое Queue?

Queue - это очередь, которая обычно (но необязательно) строится по принципу FIFO (First-In-First-Out) - соответственно извлечение элемента осуществляется с начала очереди, вставка элемента - в конец очереди. Хотя этот принцип нарушает, к примеру, PriorityQueue, использующая «natural ordering» или переданный Comparator при вставке нового элемента.
Особенностью PriorityQueue 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.

Используя PriorityQueue, можно, например, реализовать алгоритм Дейкстры для поиска кратчайшего пути от одной вершины графа к другой. Либо для хранения объектов согласно определённого свойства.

25. Что такое Dequeue? Чем отличается от Queue?

Deque (Double Ended Queue) расширяет Queue и согласно документации, это линейная коллекция, поддерживающая вставку/извлечение элементов с обоих концов. Помимо этого, реализации интерфейса Deque могут строится по принципу FIFO, либо LIFO.
Реализации и Deque, и Queue обычно не переопределяют методы equals() и hashCode(), вместо этого используются унаследованные методы класса Object, основанные на сравнении ссылок.

26. Приведите пример реализации Dequeue.

ArrayDeque (также известный как «Array Double Ended Queue», произносится как «ArrayDeck») - это особый вид растущего массива, который позволяет нам добавлять или удалять элементы *с обеих сторон*.
Реализация ArrayDeque может использоваться как Stack (последний пришел-первый вышел) или Queue (первый пришел-первый вышел)
Под капотом ArrayDeque поддерживается массив, который удваивает свой размер при заполнении. Первоначально, массив инициализируется с размером 16. Он реализован как двусторонняя очередь, где он поддерживает два указателя, а именно голову и хвост.

ArrayDeque:

- Это не потокобезопасный
- Нулевые элементы не принимаются
- Работает значительно быстрее, чем синхронизированный *Stack*
- Более быстрая очередь, чем *LinkedList* из-за лучшей локализации. Большинство операций амортизировали сложность с постоянным временем
  - *Iterator* , возвращаемый *ArrayDeque* , является отказоустойчивым
  - *ArrayDeque* автоматически удваивает размер массива, когда head и хвостовой указатель встречает друг друга при добавлении элемента

```
Deque<Integer> stack = new ArrayDeque<>();
stack.push(1);
stack.push(2);
stack.push(3);
while (!stack.isEmpty()) {
   System.out.println(stack.pop());
}
```

Результат:
3
2
1

Выводы

На основании рассмотренных нами интерфейсов и реализаций можно сделать вывод, что для самой простой реализации очереди Queue следует выбрать LinkedList. Если требуется как-то сортировать элементы внутри очереди, то подойдёт PriorityQueue. Если же нам нужна функциональность стека, то надо использовать интерфейс Deque и одну из его реализаций: LinkedList или ArrayDeque

27. Какая коллекция реализует FIFO? Queue

28. Какая коллекция реализует LIFO? Stack

29. Оцените количество памяти на хранение одного примитива типа byte в LinkedList?

Каждый элемент LinkedList хранит ссылку на предыдущий элемент, следующий элемент и ссылку на данные.

```
private static class Node<E> {
        E item;
        Node<E> next;
        Node<E> prev;
//...
```

}
Для 32-битных систем каждая ссылка занимает 32 бита (4 байта). Сам объект (заголовок) вложенного класса Node занимает 8 байт. 4 + 4 + 4 + 8 = 20 байт, а т.к. размер каждого объекта в Java кратен 8, соответственно получаем 24 байта. Примитив типа byte занимает 1 байт памяти, но в JCF примитивы упаковываются: объект типа Byte занимает в памяти 16 байт (8 байт на заголовок объекта, 1 байт на поле типа byte и 7 байт для кратности 8). Также напомню, что значения от -128 до 127 кэшируются и для них новые объекты каждый раз не создаются. Таким образом, в x32 JVM 24 байта тратятся на хранение одного элемента в списке и 16 байт - на хранение упакованного объекта типа Byte. Итого 40 байт.

Для 64-битной JVM каждая ссылка занимает 64 бита (8 байт), размер заголовка каждого объекта составляет 16 байт (два машинных слова). Вычисления аналогичны: 8 + 8 + 8 + 16 = 40байт и 24 байта (объект). Итого 64 байта.

30. Оцените количество памяти на хранение одного примитива типа byte в ArrayList?
ArrayList основан на массиве, для примитивных типов данных осуществляется автоматическая упаковка значения, поэтому 16 байт тратится на хранение упакованного объекта и 4 байта (8 для x64) - на хранение ссылки на этот объект в самой структуре данных. Таким образом, в x32 JVM 4 байта используются на хранение одного элемента и 16 байт - на хранение упакованного объекта типа Byte. Для x64 - 8 байт и 24 байта соотвтетсвенно.
x32=20байт
x64=32байт

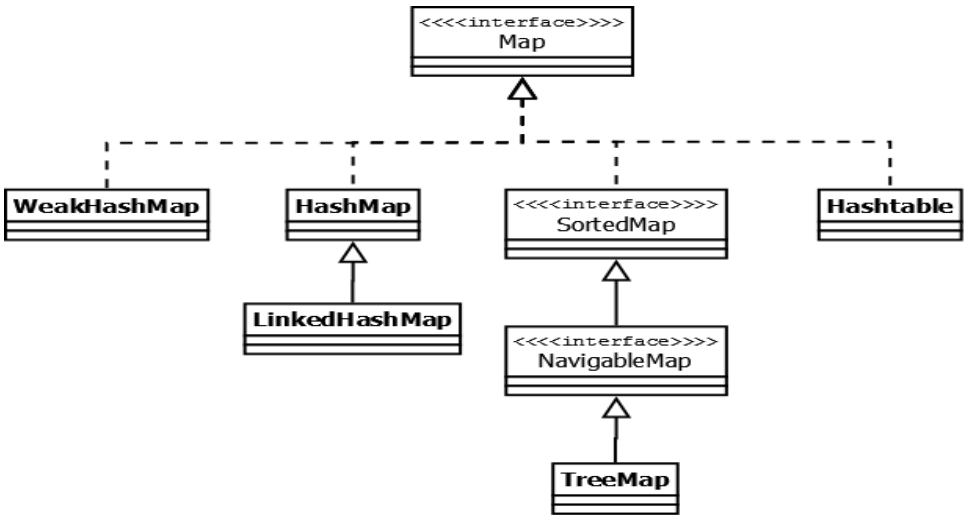
31. Какие существуют реализации Map?
Hashtable — реализация такой структуры данных, как хэш-таблица. Она не позволяет использовать null в качестве значения или ключа. Эта коллекция была реализована раньше, чем Java Collection Framework, но в последствии была включена в его состав. Как и другие коллекции из Java 1.0, Hashtable является синхронизированной (почти все методы помечены как synchronized). Из-за этой особенности у неё имеются существенные проблемы с производительностью и, начиная с Java 1.2, в большинстве случаев рекомендуется использовать другие реализации интерфейса Map ввиду отсутствия у них синхронизации.

HashMap — коллекция является альтернативой Hashtable. Двумя основными отличиями от Hashtable являются то, что HashMap не синхронизирована и HashMap позволяет использовать null как в качестве ключа, так и значения. Так же как и Hashtable, данная коллекция не является упорядоченной: порядок хранения элементов зависит от хэш-функции. Добавление элемента выполняется за константное время O(1), но время удаления, получения зависит от распределения хэш-функции. В идеале является константным, но может быть и линейным O(n).

LinkedHashMap — это упорядоченная реализация хэш-таблицы. Здесь, в отличии от HashMap, порядок итерирования равен порядку добавления элементов. Данная особенность достигается благодаря двунаправленным связям между элементами (аналогично LinkedList). Но это преимущество имеет также и недостаток — увеличение памяти, которое занимет коллекция.

TreeMap — реализация Map основанная на красно-чёрных деревьях. Как и LinkedHashMap является упорядоченной. По-умолчанию, коллекция сортируется по ключам с использованием принципа "natural ordering", но это поведение может быть настроено под конкретную задачу при помощи объекта Comparator, который указывается в качестве параметра при создании объекта TreeMap.

WeakHashMap — реализация хэш-таблицы, которая организована с использованием weak references. Другими словами, Garbage Collector автоматически удалит элемент из коллекции при следующей сборке мусора, если на ключ этого элеметна нет жёстких ссылок.



	HashMap	LinkedHashMap	TreeMap
Порядок хранения данных	Случайный. Нет гарантий, что порядок сохранится на протяжении времени	В порядке добавления	В порядке возрастания или исходя из заданного компаратора
Время доступа к элементам	O(1)	O(1)	O(log(n))
Имплементированные интерфейсы	Map	Map	NavigableMap SortedMap Map
Имплементация на основе структуры данных	Корзины (buckets)	Корзины (buckets)	Красно-чёрное дерево (Red-Black Tree)
Возможность работы с null-ключем	Можно	Можно	Можно, если не используется компаратор
Потокобезопасна	Нет	Нет	Нет

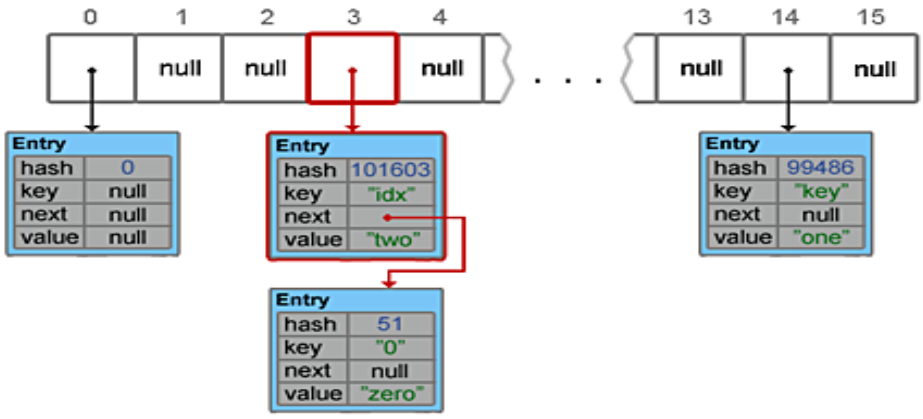
размер сложность	10	20	30	40	50	60
n	0,00001 сек.	0,00002 сек.	0,00003 сек.	0,00004 сек.	0,00005 сек.	0,00005 сек.
n^2	0,0001 сек.	0,0004 сек.	0,0009 сек.	0,0016 сек.	0,0025 сек.	0,0036 сек.
n^3	0,001 сек.	0,008 сек.	0,027 сек.	0,064 сек.	0,125 сек.	0,216 сек.
n^5	0,1 сек.	3,2 сек.	24,3 сек.	1,7 минут	5,2 минут	13 минут
2^n	0,0001 сек.	1 сек.	17,9 минут	12,7 дней	35,7 веков	366 веков
3^n	0,059 сек.	58 минут	6,5 лет	3855 веков	$2x10^8$ веков	$1,3x10^{13}$ веков

O(1) можно прочитать как «сложность порядка 1» (order 1), или «алгоритм выполняется за постоянное/константное время» (constant time).O(1) алгоритмы самые эффективные.
O(n), или «сложность порядка n (order n)». Здесь нужно перебрать все элементы, т.е. операция на каждый элемент. Чем больше массив, тем больше операций.
O(n^2) Алгоритмы с вложенными циклами по той же коллекции всегда O(n^2). Итерирование массива это O(n). У нас есть вложенный цикл, для каждого элемента мы еще раз итерируем — т.е. O(n^2) или «сложность порядка n квадрат».
O(log n) Т.е. в худшем случае делаем столько операций, сколько раз можем разделить массив на две части. Например, сколько раз мы можем разделить на две части массив из 4 элементов? 2 раза. А массив из 8 элементов? 3 раза. Т.е. кол-во делений/операций = log2(n) (где n кол-во элементов массива). В алгоритме «бинарный поиск» на каждом шаге мы делим массив на две части.

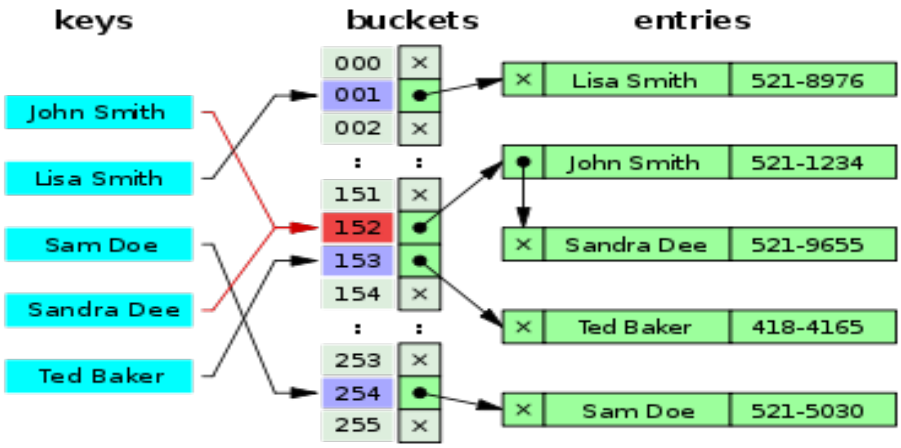
32.Как устроена HashMap, сложность основных операций? (Расскажите про принцип корзин)
HashMap — основан на хэш-таблицах, реализует интерфейс Map (что подразумевает хранение данных в виде пар ключ/значение). Ключи и значения могут быть любых типов, в том числе и null. Данная реализация не дает гарантий относительно порядка элементов с течением времени. Разрешение коллизий осуществляется с помощью метода цепочек.
Особенности:

— Добавление элемента выполняется за время O(1), потому как новые элементы вставляются в начало цепочки;
— Операции получения и удаления элемента могут выполняться за время O(1), если хэш-функция равномерно распределяет элементы и отсутствуют коллизии. Среднее же время работы будет Θ(1 + α), где α — коэффициент загрузки. В самом худшем случае, время выполнения может составить Θ(n) (все элементы в одной цепочке);
— Ключи и значения могут быть любых типов, в том числе и null. Для хранения примитивных типов используются соответствующие классы-оберки;
— Не синхронизирован.
Создание объекта:
Map<String, String> hashmap = new HashMap<String, String>();
Новоявленный объект hashmap, содержит ряд свойств:
- table — Массив типа Entry[], который является хранилищем ссылок на списки (цепочки) значений;
- loadFactor — Коэффициент загрузки. Значение по умолчанию 0.75 является хорошим компромиссом между временем доступа и объемом хранимых данных;
- threshold — Предельное количество элементов, при достижении которого, размер хэш-таблицы увеличивается вдвое. Рассчитывается по формуле (capacity * loadFactor);
size — Количество элементов HashMap-а;

Добавление элементов в объект HasMap:
- Сначала ключ проверяется на равенство null. Если это проверка вернула true, будет вызван метод putForNullKey(value)
- Далее генерируется хэш на основе ключа. Для генерации используется метод hash(hashCode), в который передается key.hashCode()
- С помощью метода indexFor(hash, tableLength), определяется позиция в массиве, куда будет помещен элемент.
- Теперь, зная индекс в массиве, мы получаем список (цепочку) элементов, привязанных к этой ячейке. Хэш и ключ нового элемента поочередно сравниваются с хэшами и ключами элементов из списка и, при совпадении этих параметров, значение элемента перезаписывается.
- Если же предыдущий шаг не выявил совпадений, будет вызван метод addEntry(hash, key, value, index) для добавления нового элемента.



Разрешение коллизий с помощью метода цепочек:



Когда массив table[] заполняется до предельного значения, его размер увеличивается вдвое и происходит перераспределение элементов с помощью методов resize(capacity) и transfer(newTable). Метод transfer() перебирает все элементы текущего хранилища, пересчитывает их индексы (с учетом нового размера) и перераспределяет элементы по новому массиву.
У HashMap есть такая же «проблема» как и у ArrayList — при удалении элементов размер массива table[] не уменьшается. И если в ArrayList предусмотрен метод trimToSize(), то в HashMap таких методов нет.
HashMap имеет встроенные итераторы, такие, что вы можете получить список всех ключей keySet(), всех значений values() или же все пары ключ/значение entrySet(). Ниже представлены некоторые варианты для перебора элементов:

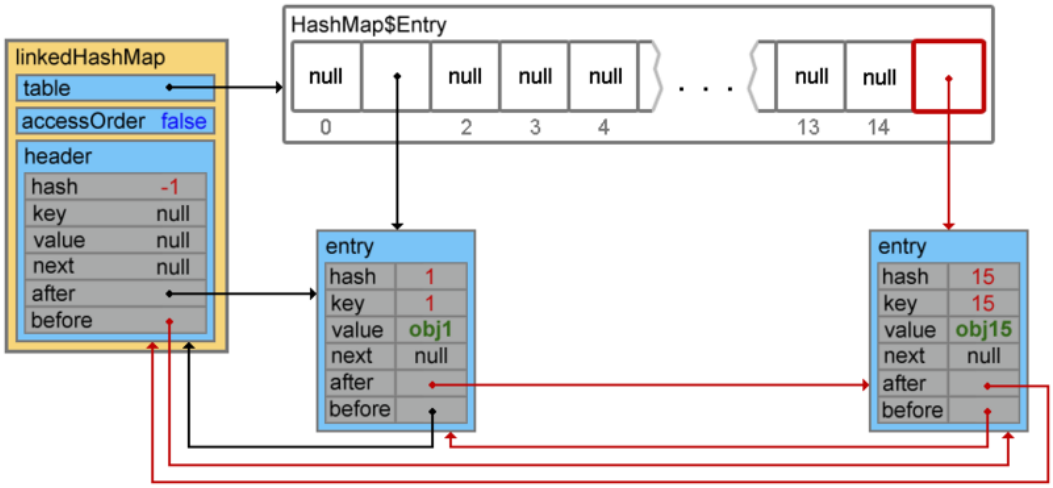
```
// 1. все ключи-значения
for (Map.Entry<String, String> entry: hashmap.entrySet())
    System.out.println(entry.getKey() + " = " + entry.getValue());

// 2. все ключи
for (String key: hashmap.keySet())
    System.out.println(hashmap.get(key));

// 3. получить все пары ключ/значений
Iterator<Map.Entry<String, String>> itr = hashmap.entrySet().iterator();
while (itr.hasNext())
    System.out.println(itr.next());
```

33. Что такое LinkedHashMap?

LinkedHashMap - упорядоченная реализация хэш-таблицы. Здесь, в отличии от HashMap, порядок итерирования равен порядку добавления элементов.
Данная структура может слегка уступать по производительности родительскому HashMap, при этом время выполнения операций add(), contains(), remove() остается константой — O(1). Понадобится чуть больше места в памяти для хранения элементов и их связей, но это совсем небольшая плата за дополнительные фишечки. Он также принимает нулевой ключ, а также нулевые значения.
LinkedHashMap не синхронизирована.
Вообще, из-за того что всю основную работу на себя берет родительский класс, серьезных отличий в реализации HashMap и LinkedHashMap не много. Можно упомянуть о парочке мелких:
Методы transfer() и containsValue() устроены чуть проще из-за наличия двунаправленной связи между элементами;
В классе LinkedHashMap.Entry реализованы методы recordRemoval() и recordAccess() (тот самый, который помещает элемент в конец при accessOrder = true). В HashMap оба этих метода пустые.

```
Map<Integer, String> linkedHashMap = new LinkedHashMap<Integer, String>();
```

linkedHashMap.put(1, "obj1");
linkedHashMap.put(15, "obj15");



Только что созданный объект linkedHashMap, помимо свойств унаследованных от HashMap (такие как table, loadFactor, threshold, size, entrySet и т.п.), так же содержит два доп. свойства:
header — «голова» двусвязного списка. При инициализации указывает сам на себя;
accessOrder — указывает каким образом будет осуществляться доступ к элементам при использовании итератора. При значении true — по порядку последнего доступа. При значении false доступ осуществляется в том порядке, в каком элементы были вставлены.

34. Как устроена TreeMap, сложность основных операций?
Класс TreeMap<K, V> представляет отображение в виде дерева. Он наследуется от класса AbstractMap и реализует интерфейс NavigableMap, а следовательно, также и интерфейс SortedMap. Поэтому в отличие от коллекции HashMap в TreeMap все объекты автоматически сортируются по возрастанию их ключей.
Время доступа и извлечения элементов достаточно мало, что делает класс TreeMap блестящим выбором для хранения больших объемов отсортированной информации, которая должна быть быстро найдена.
Класс TreeMap имеет следующие конструкторы:
- TreeMap(): создает пустое отображение в виде дерева
- TreeMap(Map<? extends K,? extends V> map): создает дерево, в которое добавляет все элементы из отображения map
- TreeMap(SortedMap<K, ? extends V> smap): создает дерево, в которое добавляет все элементы из отображения smap
- TreeMap(Comparator<? super K> comparator): создает пустое дерево, где все добавляемые элементы впоследствии будут отсортированы компаратором.
Кроме собственно методов интерфейса Map класс TreeMap реализует методы интерфейса NavigableMap. Например, мы можем получить все объекты до или после определенного ключа с помощью методов headMap и tailMap. Также мы можем получить первый и последний элементы и провести ряд дополнительных манипуляций с объектами

SortedMap — интерфейс, который расширяет Map и добавляет методы, актуальные для отсортированного набора данных:

- firstKey(): возвращает ключ первого элемента мапы;
- lastKey(): возвращает ключ последнего элемента;
- headMap(K end): возвращает мапу, которая содержит все элементы текущей, от начала до элемента с ключом end;
- tailMap(K start): возвращает мапу, которая содержит все элементы текущей, начиная с элемента start и до конца;

- subMap(K start, K end): возвращает мапу, которая содержит все элементы текущей,начиная с элемента start и до элемента с ключом end.

NavigableMap — интерфейс, который расширяет SortedMap и добавляет методы для навигации между элементами мапы:

- firstEntry(): возвращает первый пару “ключ-значение”;
- lastEntry(): возвращает последнюю пару “ключ-значение”;
- pollFirstEntry(): возвращает и удаляет первую пару;
- pollLastEntry(): возвращает и удаляет последнюю пару;
- ceilingKey(K obj): возвращает наименьший ключ k, который больше или равен ключу obj. Если такого ключа нет, возвращает null;
- floorKey(K obj): возвращает самый большой ключ k, который меньше или равен ключу obj. Если такого ключа нет, возвращает null;
- lowerKey(K obj): возвращает наибольший ключ k, который меньше ключа obj. Если такого ключа нет, возвращает null;
- higherKey(K obj): возвращает наименьший ключ k, который больше ключа obj. Если такого ключа нет, возвращает null;
- ceilingEntry(K obj): аналогичен методу ceilingKey(K obj), только возвращает пару “ключ-значение” (или null);
- floorEntry(K obj): аналогичен методу floorKey(K obj), только возвращает пару “ключ-значение” (или null);
- lowerEntry(K obj): аналогичен методу lowerKey(K obj), только возвращает пару “ключ-значение” (или null);
- higherEntry(K obj): аналогичен методу higherKey(K obj), только возвращает пару “ключ-значение” (или null);
- descendingKeySet(): возвращает NavigableSet, содержащий все ключи, отсортированные в обратном порядке;
- descendingMap(): возвращает NavigableMap, содержащую все пары, отсортированные в обратном порядке;
- navigableKeySet(): возвращает объект NavigableSet, содержащий все ключи в порядке хранения;
- headMap(K upperBound, boolean incl): возвращает мапу, которая содержит пары от начала и до элемента upperBound. Аргумент incl указывает, нужно ли включать элемент upperBound в возвращаемую мапу;
- tailMap(K lowerBound, boolean incl): функционал похож на предыдущий метод, только возвращаются пары от lowerBound и до конца;
- subMap(K lowerBound, boolean lowIncl, K upperBound, boolean highIncl): как и в предыдущих методах, возвращаются пары от lowerBound и до upperBound, аргументы lowIncl и highIncl указывают, включать ли граничные элементы в новую мапу.

35. Что такое WeakHashMap?

является реализацией интерфейса Map на основе хеш-таблиц, с ключами из WeakReference тип. Запись в WeakHashMap будет автоматически удалена, если ее ключ больше не используется в обычном режиме. Это означает, что не существует ни одной ссылки, указывающая на этот ключ. Когда процесс сборки мусора (GC) отбрасывает ключ, его запись эффективно удаляется с карты, поэтому этот класс ведет себя несколько иначе, чем другие реализации Map.
Применение – реализация простого кэша.

36. Как работает HashMap при попытке сохранить в него два элемента по ключам с одинаковым hashCode(), но для которых equals() == false?

По значению hashCode() вычисляется индекс ячейки массива, в список которой этот элемент будет добавлен. Перед добавлением осуществляется проверка на наличие элементов в этой ячейке. Если элементы с таким hashCode() уже присутствует, но их equals() методы не равны, то элемент будет добавлен в конец списка.

37. Что будет, если мы кладем в HashMap ключ, у которого equals и hashCode определены некорректно?

Некорректно equals и hashcode - не найдем корзину и не найдем элемент
Если некорректно equals – как минимум найдем корзину хэш-таблицы, в которой объект будет лежать,
Если некорректно hascode - помещая некий объект в хэш-таблицу, мы рискуем не получить его обратно по ключу

38. Возможна ли ситуация, когда HashMap выродится в список даже с ключами имеющими разные hashCode()?

Это возможно в случае, если метод indexFor(hash, tableLength), определяющий номер корзины будет возвращать одинаковые значения

39. Почему нельзя использовать byte[] в качестве ключа в HashMap?

Хэш-код массива не зависит от хранимых в нем элементов, а присваивается при создании массива (метод вычисления хэш-кода массива не переопределен и вычисляется по стандартному Object.hashCode() на основании адреса массива). Так же у массивов не переопределен equals и выполняется сравнение указателей. Это приводит к тому, что обратиться к сохраненному с ключом-массивом элементу не получится при использовании другого массива такого же размера и с такими же элементами, доступ можно осуществить лишь в одном случае — при использовании той же самой ссылки на массив, что использовалась для сохранения элемента

40. Будет ли работать HashMap, если все добавляемые ключи будут иметь одинаковый hashCode()?

Да, будет, но в этом случае HashMap вырождается в связный список и теряет свои преимущества

41. Какое худшее время работы метода get(key) для ключа, которого нет в HashMap?
42. Какое худшее время работы метода get(key) для ключа, который есть в HashMap?

O(N). Худший случай - это поиск ключа в HashMap, вырожденного в список по причине совпадения ключей по hashCode() и для выяснения хранится ли элемент с определённым ключом может потребоваться перебор всего списка.

9. Функциональные интерфейсы

1. **Что такое функциональный интерфейс? Примеры**
2. **Для чего нужна аннотация @FunctionalInterface?**
3. **Какие встроенные функциональные интерфейсы вы знаете?**

Функциональный интерфейс - это интерфейс, который определяет только один абстрактный метод.
Основное назначение – использование в лямбда выражениях и method reference.
Чтобы точно определить интерфейс как функциональный, добавлена аннотация @FunctionalInterface, работающая по принципу @Override. Она обозначит замысел и не даст определить второй абстрактный метод в интерфейсе.
Интерфейс может включать сколько угодно default методов и при этом оставаться функциональным, потому что default методы - не абстрактные.

встроенные функциональные интерфейсы:
- Predicate<T> Проверяет соблюдение некоторого условия. Если оно соблюдается, то возвращается значение true. В качестве параметра лямбда-выражение принимает объект типа T
- Consumer<T> выполняет некоторое действие над объектом типа T, при этом ничего не возвращая
- Function<T,R> представляет функцию перехода от объекта типа T к объекту типа R
- Supplier<T> не принимает никаких аргументов, но должен возвращать объект типа T
- UnaryOperator<T> принимает в качестве параметра объект типа T, выполняет над ними операции и возвращает результат операций в виде объекта типа T
- BinaryOperator<T> принимает в качестве параметра два объекта типа T, выполняет над ними бинарную операцию и возвращает ее результат также в виде объекта типа T

Predicate<T>	public interface Predicate<T> { boolean test(T t);	import java.util.function.Predicate;

	}	public class LambdaApp {
		public static void main(String[] args) {
		Predicate<Integer> isPositive = x -> x > 0;
		System.out.println(isPositive.test(5)); // true
		System.out.println(isPositive.test(-7)); // false
		}
		}

BinaryOperator<T>	public interface
	BinaryOperator<T> {
	T apply(T t1, T t2);
	}
		import java.util.function.BinaryOperator;
		
		public class LambdaApp {
		public static void main(String[] args) {
		BinaryOperator<Integer> multiply = (x, y) -> x*y;
		System.out.println(multiply.apply(3, 5)); // 15
		System.out.println(multiply.apply(10, -2)); // -20
		}
		}

UnaryOperator<T>	public interface
	UnaryOperator<T> {
	T apply(T t);
	}
		import java.util.function.UnaryOperator;
		
		public class LambdaApp {
		public static void main(String[] args) {
		UnaryOperator<Integer> square = x -> x*x;
		System.out.println(square.apply(5)); // 25
		}
		}

Function<T,R>	public interface Function<T, R> {
	R apply(T t);
	}
		import java.util.function.Function;
		
		public class LambdaApp {
		public static void main(String[] args) {
		Function<Integer, String> convert = x-> String.valueOf(x)
		+ " долларов";
		System.out.println(convert.apply(5)); // 5 долларов
		}
		}

Consumer<T>	public interface Consumer<T> {
	void accept(T t);
	}
		import java.util.function.Consumer;
		
		public class LambdaApp {
		public static void main(String[] args) {
		Consumer<Integer> printer = x-> System.out.printf("%d
		долларов \n", x);
		printer.accept(600); // 600 долларов
		}
		}

Supplier<T>	public interface Supplier<T> {
	T get();
	}
		import java.util.Scanner;
		import java.util.function.Supplier;
		
		public class LambdaApp {
		public static void main(String[] args) {
		Supplier<User> userFactory = ()->{
		
		Scanner in = new Scanner(System.in);
		System.out.println("Введите имя: ");
		String name = in.nextLine();
		return new User(name);
		};
		
		User user1 = userFactory.get();
		User user2 = userFactory.get();
		
		System.out.println("Имя user1: " + user1.getName());
		System.out.println("Имя user2: " + user2.getName());
		}
		
		Введите имя:
		Том
		Введите имя:
		Сэм
		Имя user1: Том
		Имя user2: Сэм

4. Что такое ссылка на метод?

Ссылки на методы (Method References) – это компактные лямбда выражения для методов, у которых уже есть имя.

Ссылки на методы бывают четырех видов:

- Ссылка на статический метод - ContainingClass::staticMethodName

```
Function<String, Boolean> function = e -> Boolean.valueOf(e);
System.out.println(function.apply("TRUE"));
Function<String, Boolean> function = Boolean::valueOf;
System.out.println(function.apply("TRUE"));,
```

- Ссылка на нестатический метод конкретного объекта - containingObject::instanceMethodName

```
Consumer<String> consumer = e -> System.out.println(e);
consumer.accept("OCPJP 8");
Consumer<String> consumer = System.out::println;
consumer.accept("OCPJP 8");
```

- Ссылка на нестатический метод любого объекта конкретного типа

ContainingType::methodName

```
Function<String, String> function = s -> s.toLowerCase();
System.out.println(function.apply("OCPJP 8"));
Function<String, String> function = String::toLowerCase;
System.out.println(function.apply("OCPJP 8"));
```

- Ссылка на конструктор ClassName::new

```
Function<String, Integer> function = (d) -> new Integer(d);
System.out.println(function.apply("4"));
Function<String, Integer> function = Integer::new;
System.out.println(function.apply("4"));
```

5. Что такое лямбда-выражение? Чем его можно заменить?

Представляет набор инструкций, которые можно выделить в отдельную переменную и затем многократно вызвать в различных местах программы. Образует реализацию метода, определенного в функциональном интерфейсе. При этом важно, что функциональный интерфейс должен содержать только один единственный метод без реализации.

список параметров выражения -> тело лямбда-выражения (действия)

Параметры лямбда-выражения должны соответствовать по типу параметрам метода из функционального интерфейса.

в лямбда-выражении использование обобщений не допускается. В этом случае нам надо типизировать объект интерфейса определенным типом, который потом будет применяться в лямбда-выражении

```
public class LambdaApp {

    public static void main(String[] args) {

        Operationable<Integer> operation1 = (x, y)-> x + y;
        Operationable<String> operation2 = (x, y) -> x + y;

        System.out.println(operation1.calculate(20, 10)); //30
        System.out.println(operation2.calculate("20", "10")); //2010
    }
}
interface Operationable<T>{
    T calculate(T x, T y);
```

}

Одним из ключевых моментов в использовании лямбд является отложенное выполнение (deferred execution). То есть мы определяем в одном месте программы лямбда-выражение и затем можем его вызывать при необходимости неопределенное количество раз в различных частях программы. Отложенное выполнение может потребоваться, к примеру, в следующих случаях:

Выполнение кода отдельном потоке

Выполнение одного и того же кода несколько раз

Выполнение кода в результате какого-то события

Выполнение кода только в том случае, когда он действительно необходим и если он необходим

10. Stream API

1. Что такое Stream API? Для чего нужны стримы?
Интерфейс java.util.Stream представляет собой последовательность элементов, над которой можно производить различные операции.
Нужны для упрощения работы с наборами данных, в частности, упростить операции фильтрации, сортировки и другие манипуляции с данными.

IntStream.of(50, 60, 70, 80, 90, 100, 110, 120).filter(x -> x < 90).map(x -> x + 10)
.limit(3).forEach(System.out::print);
- создаем экземпляр Stream
Пустой стрим: Stream.empty()
Стрим из List: list.stream()
Стрим из Map: map.entrySet().stream()
Стрим из массива: Arrays.stream(array)
Стрим из указанных элементов: Stream.of("1", "2", "3")
Стрим из BufferedReader с помощью метода lines (); нужно закрывать close ().
- Промежуточные (“intermediate”, “lazy”) — обрабатывают поступающие элементы и возвращают стрим. Может быть много, а может и не быть ни одной.
- Терминальные (“terminal”, ещё называют “eager”) — обрабатывают элементы и завершают работу стрима, может быть только один.
Важные моменты:
- Обработка не начнётся до тех пор, пока не будет вызван терминальный оператор. list.stream().filter(s -> s > 5) (не возьмёт ни единого элемента из списка);
- Экземпляр стрима нельзя использовать более одного раза;

Коллекции	Streams
Конечны (хранят набор элементов)	Бесконечны
Индивидуальный доступ к элементам	Нет индивид. доступа к элементам
Можно менять (добавлять/удалять) элементы, в т.ч. через итератор	Если как то обрабатываем данные, то не влияет на источник

Кроме универсальных объектных существуют особые виды стримов для работы с примитивными типами данных int, long и double: IntStream, LongStream и DoubleStream. Эти примитивные стримы работают так же, как и обычные объектные, но со следующими отличиями:
- используют специализированные лямбда-выражения, например, IntFunction или IntPredicate вместо Function и Predicate;

- поддерживают дополнительные конечные операции sum(), average(), mapToObj()

2. Почему Stream называют ленивым?

Ленивое программирование -- технология, которая позволяет вам отсрочить вычисление кода до тех пор, пока не понадобится его результирующее значение.
Блок обработки – промежуточные операции не выполняются, пока не вызовется терминальная.

3. Какие существуют способы создания стрима?

Из коллекции:
Stream<String> fromCollection = Arrays.asList("x", "y", "z").stream();
Из набора значений:
Stream<String> fromValues = Stream.of("x", "y", "z");
Из массива:
Stream<String> fromArray = Arrays.stream(new String[]{"x", "y", "z"});
Из файла (каждая строка в файле будет отдельным элементом в стриме):
Stream<String> fromFile = Files.lines(Paths.get("input.txt"));
Из строки:
IntStream fromString = "0123456789".chars();
С помощью Stream.builder():
Stream<String> fromBuilder = Stream.builder().add("z").add("y").add("z").build();
С помощью Stream.iterate() (бесконечный):
Stream<Integer> fromIterate = Stream.iterate(1, n -> n + 1);
С помощью Stream.generate() (бесконечный):
Stream<String> fromGenerate = Stream.generate(() -> "0");

4. Как из коллекции создать стрим?

Stream<String> fromCollection = Arrays.asList("x", "y", "z").stream();

5. Какие промежуточные методы в стримах вы знаете?

6. Расскажите про метод peek() “быстрый взгляд”.

- peek (принимает Consumer)
integerStream.peek(System.out::println)
позволяет подсмотреть какие элементы летают на данном этапе с помощью System.out::println
Stream<T> peek(Consumer<? super T> action);

7. Расскажите про метод map() “маппинг – из одного в другое”.

Отображение или маппинг позволяет задать функцию преобразования одного объекта в другой, то есть получить из элемента одного типа элемент другого типа.
принимает Function.
<R> Stream<R> map(Function<? super T, ? extends R> mapper);
map(n -> n.toString())

8. Расскажите про метод flatMap(). “плоский маппинг”

Плоское отображение выполняется тогда, когда из одного элемента нужно получить несколько.

```
Stream
    .of("H e l l o", "w o r l d !")
    .flatMap((p) -> Arrays.stream(p.split(" ")))
    .toArray(String[]::new);//["H", "e", "l", "l", "o", "w", "o", "r", "l", "d", "!"]
```

Например, в примере выше мы выводим название телефона и его цену. Но что, если мы хотим установить для каждого телефона цену со скидкой и цену без скидки. То есть из одного объекта Phone нам надо получить два объекта с информацией, например, в виде строки. Для этого применим flatMap:
Stream<Phone> phoneStream = Stream.of(new Phone("iPhone 6 S", 54000), new Phone("Lumia 950", 45000),

```
                new Phone("Samsung Galaxy S 6", 40000));

    phoneStream
        .flatMap(p->Stream.of(
                String.format("название: %s  цена без скидки: %d", p.getName(), p.getPrice()),
                String.format("название: %s  цена со скидкой: %d", p.getName(), p.getPrice() -
    (int)(p.getPrice()*0.1))
        ))
        .forEach(s->System.out.println(s));
```

9. Чем отличаются методы map() и flatMap().

10. Расскажите про метод filter()

Метод filter() является промежуточной операцией принимающей предикат, который фильтрует все элементы, возвращая только те, что соответствуют условию.

```
Stream<String> citiesStream = Stream.of("Париж", "Лондон", "Мадрид","Берлин", "Брюссель");
citiesStream.filter(s->s.length()==6).forEach(s->System.out.println(s));
```

11. Расскажите про метод limit()

limit(long n) применяется для выборки первых n элементов потоков. Этот метод также возвращает модифицированный поток, в котором не более n элементов.

```
Stream<String> phoneStream = Stream.of("iPhone 6 S", "Lumia 950", "Samsung Galaxy S 6", "LG G 4",
"Nexus 7");

phoneStream.skip(1)
    .limit(2)
    .forEach(s->System.out.println(s)); // Lumia 950 Samsung Galaxy S 6
```

12. Расскажите про метод skip()

skip(long n) используется для пропуска n элементов. Этот метод возвращает новый поток, в котором пропущены первые n элементов.

13. Расскажите про метод sorted()

Для простой сортировки по возрастанию применяется метод sorted(). Подходит только для сортировки тех объектов, которые реализуют интерфейс Comparable.

Если же у нас классы объектов не реализуют этот интерфейс или мы хотим создать какую-то свою логику сортировки, то мы можем использовать другую версию метода sorted(), которая в качестве параметра принимает компаратор.

```
class Phone{

    private String name;
    private String company;
    private int price;

    public Phone(String name,
String comp, int price){
        this.name=name;
        this.company=comp;
        this.price = price;
    }

    public String getName() { return
name; }
    public int getPrice() { return
price; }
    public String getCompany() {
return company; }
}
```

```
import java.util.Comparator;
import java.util.stream.Stream;

public class Program {

    public static void main(String[] args) {

        Stream<Phone> phoneStream = Stream.of(new Phone("iPhone
X", "Apple", 600),
            new Phone("Pixel 2", "Google", 500),
            new Phone("iPhone 8", "Apple",450),
            new Phone("Nokia 9", "HMD Global",150),
            new Phone("Galaxy S9", "Samsung", 300));

        phoneStream.sorted(new PhoneComparator())
            .forEach(p->System.out.printf("%s (%s) - %d \n",
                p.getName(), p.getCompany(), p.getPrice()));

    }
}
```

	class PhoneComparator implements Comparator<Phone>{
	
	public int compare(Phone a, Phone b){
	
	return a.getName().toUpperCase().compareTo(b.getName().toUpperCase());
	}
	}

14. Расскажите про метод distinct() “особый”

возвращает только ункальные элементы в виде потока

```
Stream<String> people = Stream.of("Tom", "Bob", "Sam", "Tom", "Alice", "Kate", "Sam");
people.distinct().forEach(p -> System.out.println(p));
```

15. Какие терминальные методы в стримах вы знаете?

- forEach (принимает Consumer) например System.out::println (покажет все элементы, которые остались в стриме)
- findFirst () первый в порядке следования элемент из стрима (возвращается OptionalInt, т.к. может вернуться 0)
- allMatch (принимает предикат) позволяет удостовериться, удовлетворяют ли все элементы стрима определенному условию
- min () возвращает минимальный элемент из стрима
- count () возвращает количество элементов, оставшееся в стриме
- sum () 0
- collect () собирает элементы стрима в новое хранилище, например список. Куда может собрать – смотри список Collectors:
    - groupingBy группирует по параметрам
    - averagingInt среднеарифметическое какого то параметра
    - summarizingInt дает кол-во элементов, суммму, мин, ср.арифм, макс значения
- reduce () – позволяет вычислить свертку элементов стрима (результат применения некоторого бинарного оператора к паре элементов из стрима, пока от стрима не останется один единсттвенный элемент)

свёртка — это математическая операция, применённая к двум функциям f и g, порождающая третью функцию, которая иногда может рассматриваться как модифицированная версия одной из первоначальных

16. Расскажите про метод collect() “собирать”

собирает элементы стрима в новое хранилище, например список

Эта функция представляет объект Collector, который определен в пакете java.util.stream. Java уже предоставляет ряд встроенных функций, определенных в классе Collectors:

- toList(): преобразование к типу List
- toSet(): преобразование к типу Set
- toMap(): преобразование к типу Map
- groupingBy() - разделяет коллекцию на несколько частей и возвращает Map<N, List<T>>;
- summingInt(), summingDouble(), summingLong() - возвращает сумму;

```
List<String> filteredPhones = phones.stream()
        .filter(s->s.length()<10)
        .collect(Collectors.toList());
```

17. Расскажите про метод reduce() “уменьшить”

Позволяет выполнять какое-то действие на всей коллекции и возвращать один результат

вычислим произведение набора чисел:

```
Stream<Integer> numbersStream = Stream.of(1,2,3,4,5,6);
```

```
Optional<Integer> result = numbersStream.reduce((x,y)->x*y);
System.out.println(result.get()); // 720
```

18. Расскажите про класс Collectors и его методы.

В Java 8 в классе Collectors реализовано несколько распространённых коллекторов:
- toList(), toCollection(), toSet() - представляют стрим в виде списка, коллекции или множества;
- toConcurrentMap(), toMap() - позволяют преобразовать стрим в Map;
- averagingInt(), averagingDouble(), averagingLong() - возвращают среднее значение;
- summingInt(), summingDouble(), summingLong() - возвращает сумму;
- summarizingInt(), summarizingDouble(), summarizingLong() - возвращают SummaryStatistics с разными агрегатными значениями;
- -partitioningBy() - разделяет коллекцию на две части по соответствию условию и возвращает их как Map<Boolean, List>;
- groupingBy() - разделяет коллекцию на несколько частей и возвращает Map<N, List<T>>;
- mapping() - дополнительные преобразования значений для сложных Collector-ов.
- Так же существует возможность создания собственного коллектора через Collector.of():

```
Collector<String, List<String>, List<String>> toList = Collector.of(
    ArrayList::new,
    List::add,
    (l1, l2) -> { l1.addAll(l2); return l1; }
);
```

19. Расскажите о параллельной обработке в Java 8.

Стримы могут быть последовательными и параллельными. Операции над последовательными стримами выполняются в одном потоке процессора, над параллельными — используя несколько потоков процессора. Параллельные стримы используют общий ForkJoinPool доступный через статический ForkJoinPool.commonPool() метод. При этом, если окружение не является многоядерным, то поток будет выполняться как последовательный. Фактически применение параллельных стримов сводится к тому, что данные в стримах будут разделены на части, каждая часть обрабатывается на отдельном ядре процессора, и в конце эти части соединяются, и над ними выполняются конечные операции.

Для создания параллельного потока из коллекции можно также использовать метод parallelStream() интерфейса Collection.

Чтобы сделать обычный последовательный стрим параллельным, надо вызвать у объекта Stream метод parallel(). Метод isParallel() позволяет узнать является ли стрим параллельным.

С помощью, методов parallel() и sequential() можно определять какие операции могут быть параллельными, а какие только последовательными. Так же из любого последовательного стрима можно сделать параллельный и наоборот:

```
collection
.stream()
.peek(...) // операция последовательна
.parallel()
.map(...) // операция может выполняться параллельно,
.sequential()
.reduce(...) // операция снова последовательна
```

Как правило, элементы передаются в стрим в том же порядке, в котором они определены в источнике данных. При работе с параллельными стримами система сохраняет порядок следования элементов. Исключение составляет метод forEach(), который может выводить элементы в произвольном порядке. И чтобы сохранить порядок следования, необходимо применять метод forEachOrdered().

Критерии, которые могут повлиять на производительность в параллельных стримах:

- Размер данных - чем больше данных, тем сложнее сначала разделять данные, а потом их соединять.
- Количество ядер процессора. Теоретически, чем больше ядер в компьютере, тем быстрее программа будет работать. Если на машине одно ядро, нет смысла применять параллельные потоки.
- Чем проще структура данных, с которой работает поток, тем быстрее будут происходить операции. Например, данные из ArrayList легко использовать, так как структура данной коллекции предполагает последовательность несвязанных данных. А вот коллекция типа LinkedList - не

лучший вариант, так как в последовательном списке все элементы связаны с предыдущими/последующими. И такие данные трудно распараллелить.

- Над данными примитивных типов операции будут производиться быстрее, чем над объектами классов.
- Крайне не рекомендуется использовать параллельные стримы для скольких-нибудь долгих операций (например, сетевых соединений), так как все параллельные стримы работают с одним ForkJoinPool, то такие долгие операции могут остановить работу всех параллельных стримов в JVM из-за отсутствия доступных потоков в пуле, т.е. параллельные стримы стоит использовать лишь для коротких операций, где счет идет на миллисекунды, но не для тех где счет может идти на секунды и минуты;
- Сохранение порядка в параллельных стримах увеличивает издержки при выполнении и если порядок не важен, то имеется возможность отключить его сохранение и тем самым увеличить производительность, использовав промежуточную операцию unordered():

```
collection.parallelStream()
    .sorted()
    .unordered()
    .collect(Collectors.toList());
```

20. Что такое IntStream и DoubleStream?

Кроме универсальных объектных существуют особые виды стримов для работы с примитивными типами данных int, long и double: IntStream, LongStream и DoubleStream. Эти примитивные стримы работают так же, как и обычные объектные, но со следующими отличиями:

- используют специализированные лямбда-выражения, например, IntFunction или IntPredicate вместо Function и Predicate;
- поддерживают дополнительные конечные операции sum(), average(), mapToObj()

11. Java 8

1. Какие нововведения появились в java 8?

- Методы интерфейсов по умолчанию;
- Лямбда-выражения;
- Функциональные интерфейсы;
- Ссылки на методы и конструкторы;
- Повторяемые аннотации;
- Аннотации на типы данных;
- Рефлексия для параметров методов;
- *Stream API* для работы с коллекциями;
- Параллельная сортировка массивов;
- Новое API для работы с датами и временем;
- Новый движок JavaScript *Nashorn*;
- Добавлено несколько новых классов для потокобезопасной работы;
- Добавлен новый API для Calendar и Locale;
- Добавлена поддержка *Unicode 6.2.0*;
- Добавлен стандартный класс для работы с *Base64*;
- Добавлена поддержка беззнаковой арифметики;
- Улучшена производительность конструктора java.lang.String(byte[], *) и метода java.lang.String.getBytes();
- Новая реализация AccessController.doPrivileged, позволяющая устанавливать подмножество привилегий без необходимости проверки всех остальных уровней доступа;
- *Password-based* алгоритмы стали более устойчивыми;
- Добавлена поддержка *SSL/TLS Server Name Indication (NSI)* в *JSSE Server*;
- Улучшено хранилище ключей (KeyStore);
- Добавлен алгоритм *SHA-224*;
- Удален мост *JDBC - ODBC*;
- Удален *PermGen*, изменен способ хранения мета-данных классов;
- Возможность создания профилей для платформы Java SE, которые включают в себя не всю платформу целиком, а некоторую ее часть;
- Инструментарий
  - Добавлена утилита jjs для использования *JavaScript Nashorn*;
  - Команда java может запускать *JavaFX* приложения;

- Добавлена утилита jdeps для анализа *.class*-файлов.

2. Какие новые классы для работы с датами появились в java 8?

Новый же Java 8 Date/Time API содержит неизменные, потокобезопасные классы с продуманным дизайном на любой вкус и цвет. Содержатся они в пакете java.time

- **LocalDate –** дата без времени и временных зон;
- **LocalTime –** время без даты и временных зон;
- **LocalDateTime –** дата и время без временных зон;
- **ZonedDateTime –** дата и время с временной зоной;
- **DateTimeFormatter –** форматирует даты в строки и наоборот, только для классов **java.time;**
- **Instant –** колличество секунд с **Unix epoch time** (полночь 1 января 1970 UTC);
- **Duration –** продолжительность в секундах и наносекундах;
- **Period –** период времени в годах, месяцах и днях;
- **TemporalAdjuster –** корректировщик дат (к примеру, может получить дату следующего понедельника);

3. Расскажите про класс Optional

Опциональное значение Optional — это контейнер для объекта, который может содержать или не содержать значение null. Такая обёртка является удобным средством предотвращения NullPointerException, т.к. имеет некоторые функции высшего порядка, избавляющие от добавления повторяющихся if null/notNull проверок:

```
Optional<String> optional = Optional.of("hello");
optional.isPresent(); // true
optional.ifPresent(s -> System.out.println(s.length())); // 5
optional.get(); // "hello"
optional.orElse("ops..."); // "hello"
```

4. Что такое Nashorn? не используется в современных версиях

Nashorn - это движок JavaScript, разрабатываемый на Java компанией Oracle. Призван дать возможность встраивать код JavaScript в приложения Java. В сравнении с Rhino, который поддерживается Mozilla Foundation, Nashorn обеспечивает от 2 до 10 раз более высокую производительность, так как он компилирует код и передает байт-код виртуальной машине Java непосредственно в памяти. Nashorn умеет компилировать код JavaScript и генерировать классы Java, которые загружаются специальным загрузчиком. Так же возможен вызов кода Java прямо из JavaScript.

5. Что такое jjs?

jjs это утилита командной строки, которая позволяет исполнять программы на языке JavaScript прямо в консоли.

6. Какой класс появился в Java 8 для кодирования/декодирования данных? используется в читалках (кодирование / декодирования в формат эл. книг)

Base64 - потокобезопасный класс, который реализует кодировщик и декодировщик данных, используя схему кодирования base64 согласно RFC 4648 и RFC 2045.
Base64 содержит 6 основных методов:
getEncoder()/getDecoder() - возвращает кодировщик/декодировщик base64, соответствующий стандарту RFC 4648; getUrlEncoder()/getUrlDecoder() - возвращает URL-safe кодировщик/декодировщик base64, соответствующий стандарту RFC 4648; getMimeEncoder()/getMimeDecoder() - возвращает MIME кодировщик/декодировщик, соответствующий стандарту RFC 2045.

7. Как создать Base64 кодировщик и декодировщик?

```
// Encode
String b64 = Base64.getEncoder().encodeToString("input".getBytes("utf-8")); //aW5wdXQ==
// Decode
new String(Base64.getDecoder().decode("aW5wdXQ=="), "utf-8"); //input
```

8. Какие дополнительные методы для работы с ассоциативными массивами (maps) появились в Java 8?

- putIfAbsent() добавляет пару «ключ-значение», только если ключ отсутствовал:
- map.putIfAbsent("a", "Aa");
- forEach() принимает функцию, которая производит операцию над каждым элементом:
- map.forEach((k, v) -> System.out.println(v));
- compute() создаёт или обновляет текущее значение на полученное в результате вычисления (возможно использовать ключ и текущее значение):
- map.compute("a", (k, v) -> String.valueOf(k).concat(v)); //["a", "aAa"]
- computeIfPresent() если ключ существует, обновляет текущее значение на полученное в результате вычисления (возможно использовать ключ и текущее значение):
- map.computeIfPresent("a", (k, v) -> k.concat(v));
- computeIfAbsent() если ключ отсутствует, создаёт его со значением, которое вычисляется (возможно использовать ключ):
- map.computeIfAbsent("a", k -> "A".concat(k)); //["a","Aa"]
- getOrDefault() в случае отсутствия ключа, возвращает переданное значение по-умолчанию:
- map.getOrDefault("a", "not found");
- merge() принимает ключ, значение и функцию, которая объединяет передаваемое и текущее значения. Если под заданным ключем значение отсутствует, то записывает туда передаваемое значение.
- map.merge("a", "z", (value, newValue) -> value.concat(newValue)); //["a","Aaz"]

9. Что такое LocalDateTime?

LocalDateTime объединяет вместе LocaleDate и LocalTime, содержит дату и время в календарной системе ISO-8601 без привязки к часовому поясу. Время хранится с точностью до наносекунды. Содержит множество удобных методов, таких как plusMinutes, plusHours, isAfter, toSecondOfDay и т.д.

10. Что такое ZonedDateTime?

java.time.ZonedDateTime — аналог java.util.Calendar, класс с самым полным объемом информации о временном контексте в календарной системе ISO-8601. Включает временную зону, поэтому все операции с временными сдвигами этот класс проводит с её учётом.