Є 2 групи методів класу Object. Одні пов’язані з багатопоточністю (wait, notify, notifyAll), інші як з корнем. Всього 9 штук. Методи класу **java.lang.Object**:

public native int **hashCode**() - виводить хеш код об’єкта, він має завжди бути одинаковим. Щось на зразок адресу об’єкта в пам’яті (ще порівнюють з номером паспорта). Створений для того аби упростити роботу з об’єктами оскільки з інтами працювати простіше ніж з ссилками. Інти можна впорядкувати, наприклад. Використовується найпоширеніше в колекціях (хеш мапа).

public boolean **equals**(Object obj) - порівнює ссилки на об’єкти. Можна використовувати для порівняння себе з кимось.

public String **toString**() - виводить клас@хешкод. Потрібно перевизначати в своїх методах.

public final native Class **getClass**()

protected native Object **clone**() throws CloneNotSupportedException - робить поверхневе копіювання.

public final native void **notify**()

public final native void **notifyAll**()

public final void **wait**() throws InterruptedException

public final native void **wait**(long timeout) throws InterruptedException

public final void **wait**(long timeout, int nanos) throws InterruptedException

protected void **finalize**() throws Throwable

----------------------------------------------------------------------------------

**Хеш-код** - результат хеш функції який є індексом по якому в хеш таблиці зберігаються дані пов’язані з ключом. Хеш-функції - механізм, за допомогою якого зберігяються дані в хеш таблицях. лежать в основі таких класів як HashMap, HashSet. Це набір складної математики, результат якої дає нам рівномірне розприділення і групування елементів за схожими значеннями та бореться з колізіями. Саме за рахунок хеша ми можемо вставляти і отримувати дані за час пропорціональний виклику самої хеш-функції ( О(1) ). В методі put() класу HashMap, наприклад, сам метод приймає ключ і значення, а якщо зайти в сорци метода, то ми побачимо, що метод для вставки елементів використовує key.hashCode(), тобто відбувається якась хеш функція над ключом, а потім вже саме місце вставки значення відбувається по шех-коду ключа. В свою чергу пошук даних в колекції фізично відбувається спочатку через так званий словар хеш таблиці по хеш-коду, а вже потім по оригнільному значенню.

HashMap складається з бакетів. Це елменти масиву які зберігають ссилки на списки елементів. При добавленні нової пари ключ-значення вичисляється хеш-код ключа, а на основі хеш код вичисляється бакет в який попаде елемент. Дефолтне значення бакетів - 16.

Хеш код вичисляє бакет, а equals перевіряє сам ключ в бакеті.

**Контракт хеш кода и equals:** equals objects must have equals hash code.

**Колізія хеш-коду** - коли хеш-коди для двох об’єктів одинакові, а самі об’єкти різні.

Хеш код має повертати одне й теж значення при багатократних викликах. Інакше ми можемо не знайти добавлений ключ.

**Хеш код** - це індекс по якому зберігаються дані пов’язані з ключом. Перетворення ключа в хеш код відбувається автоматично, хоча сам хеш код є недоступним (але хеш таблицю й так неможна індексувати безпосередньо). Хеш код і equals тісно пов’язані бо хеш код обраховується на основі вмісту об’єкта.

object1.equals(object2)// должно быть true бо реалізація цього метода в класі Обджект порівнює ссилки:

public boolean equals(Object obj) { return (this == obj); }

object1.hashCode() == object2.hashCode()// должно быть true

public native int hashCode(); //оснований на генераторі випадкових чисел Park-Miller

**Хешування** - механізм за допомогою якого зберігаються дані в хеш таблицях (бакетах). Перевага хешування полягає в тому, що він забезпечує постійність роботи методів add(), contains(), remove(), set() на навіть на великих множинах.

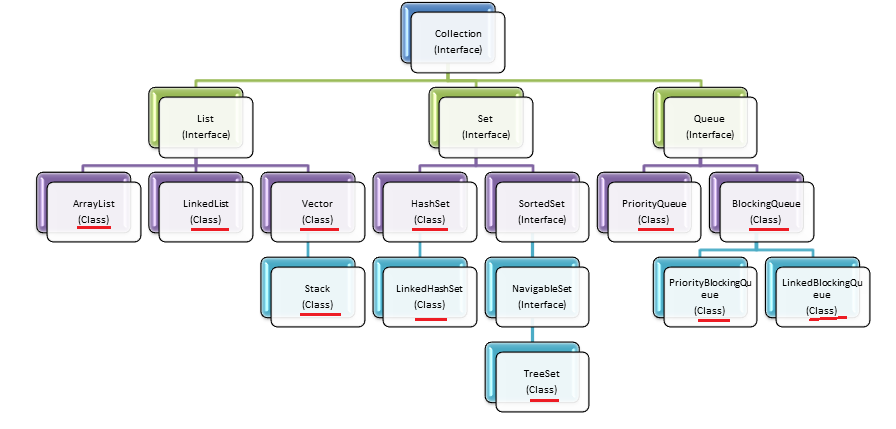
**Контракт** **hash**() & **equals**() - якщо у двох чи більше об’єктів одного класу одинаковий вміст, то і хеш-коди цих об’єктів повиині бути одинаковими. Тому для своїх створених об’єктів потрібно перевизначати методи hash & equals() враховуючи поля об’єкту (щоб потім не було проблем з HashMap яка використовує ці методи або з HashSet який може містити тільки унікальні елементи, а з неперевизначеним методом hashCode можна в Сет буде записати 2 одинакових об’єкта).

public boolean equals(Object o) {  
        if(o == null)  
        {  
            return false;  
        }  
        if (o == this)  
        {  
           return true;  
        }  
        if (getClass() != o.getClass())  
        {  
            return false;  
        }  
       
}

@Override  
 public int hashCode()  
 {  
    final int PRIME = 31;  
    int result = 1;  
    result = PRIME \* result + getId();  
    return result;  
 }

**При роботі з ОРМ** потрібно в методах **hash**() & **equals**() використовувати геттери, а не звертатися до полів напряму, оскільки ОРМ можуть використовувати lazy loading.

----------------------------------------------------------------------------------------------------------------------

Каркас колекцій стандартизує керування групами об’єктів в програмах. Основні колекції: динамічні масиви, зв’язані списки, дерева і хеш-таблиці.

Інтерфейси, які підтримують колекції:

**Collection** - знаходиться на вершині ієрархії і дозволя працювати з колекціями.

**Deque** - розширяє інтерфейс Que для роботи з двухсторонньою чергою.

**List** - Розширює інтерфейс Collection для керування послідовностями. (списками об’єктів). Елементи виводяться по індексу їх положення в списку. Індексація починається з нуля. Список може мати дублюючі елементи.

**NavigableSet** - розширяє інтерфейс SortedSet для діставання елемента за результатами пошуку за найближчими співпадіннями.

**Queue** - розширяє інтерфейс Collection для керуванням спеціальних типів списків елементи в яких видаляються тільки з початку списку.

**Set** - розширяє інтерфейс Collection для керування групами об’єктів, які мають мати тільки одне значення.

**SortedSet** - розширяють інтерфейс Set для керування сортованими групами об’єктів.

Колекції бувають **змінні** і **незмінні**. Це залежить від наявності в них методів які називаються "необов’язковими". Якщо відбувається спроба змінити колекцію, яка є незмінною - виникає виключення UnsuppertedOperationException. Всі встроєні колекції є **змінними**.

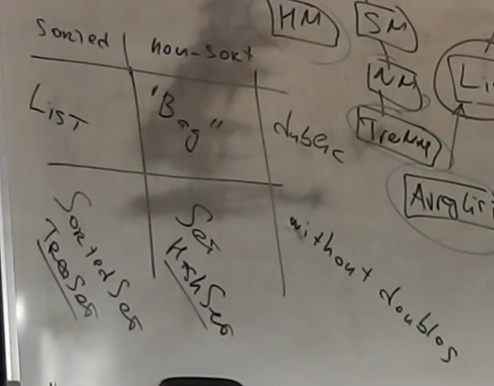
Виключення **CassCastException** виникає в тому випадку, коли ми хочемо добавити в колекцію об’єкт несумісного типу.

**NPE** виникає, коли ми хочемо добавити null в колекцію, яка не може містити пустих значень.

**IllegalArgumentException** - при спробі ввести некоректний аргумент.

**IllegalStateException** - виникає при спробі добавити елемент в заповнену колекцію фіксованої величини.

Виділяють 3 основних інтерфейса: **List, Set** (відосться до колекцій. Наслідують і-с Collection, який наслідує Iterable. Не мають ключів, тільки значення), **Map** (keys - objects).



**Інтерфейс List<E> extends Collextion<E>** – сортований і містить дублікати значень. **Set** – не сортований і не містить дублікати значень. **SortedSet** (і-с, наслідник і-са Set) – сортований але не містить дублікатів.Не існує сутності, яка б містила дублікати і була несортованою (її називають Bag - сумка) – це зобумовлено тим, що при збільшенні функціоналу страждає продуктивність. Замість Bag можна використовувати Map<K, int>, де К – ключ, int–це розмір.

**get(int x)** – взятий значення з ячейки по ключу х.

**set(int x, elem)** - добавити значення elem в ячейку х.

**boolean add(int x, elem)** - добавити ячейку зі значенням elem в позицію х.Істина – якщо добавив, фолс, якщо вже є дублікат в колекції.

**Інтерфейс Collection<T>**має 4 головних метода: **add**(T) в кінець, **remove**(T), **contains**(T), **size**();Його наслідує Set і має такі ж методи. Унаслідує **Iterator**, тому має властивість виводити елементи впорядковано.

Інтерфейс List схожий на абстрактне узагальнення масиву. Відмінність полягає в тому, що ми можемо добавляти елементи в будь яку позицію на відміну від масиву який є immutable. Також ми можемо задавати ключ в довільному порядку, наприклад, перший елемент і другий одразу 1000. В масиві ж прийшлося б заводити тисячу елементів і зайві заповняти нулями. Має методи:

**boolean add(elem)** - добавити ячейку зі значенням elem в позицію х. Істина – якщо добавив, фолс, якщо вже є дублікат в колекції.

**boolean addAll(Collection<? extends E>c)** – добавляє всі елементи колекції с в викликаючу колекцію. Істина – якщо добавив, фолс, якщо вже є дублікат в колекції.

**void clear()** – видаляє всі елементи з колекції.

**boolean contains(Object obj)** – істина, якщо обєкт є в колекції.

**boolean containsAll(Collection<?>c)** - істина, якщо всі обєкти є в колекції.

**boolean equals(Object obj)**– порівнює об’єкти на рівність. Механізм може залежати від конкретної колекції. Можна порівнювати об’єкти на ідентичність, а можна порівнювати ссилки на об’єкти.

**int hashCode()**– повертає хеш-код колекції.

**boolean isEmpty()** – перевіряє чи пуста колекція.

**Iterator<E>iterator()** – повертає ітератор для колекції.

**default Stream<E>parallelStream()** – викликає потік, який бере викликаючу колекцію в якості джерела вводу-виводу елементів. Може підтримувати багатопоточність.

**boolean remove(Object obj)** – видаляє елемент.

**boolean removeAll(Collection<?>c)** – видаляє всі елементи викликаючої колекції які є в колекції с.

**default boolean removeIf(Predicate<? superE>pred)** – видаляє всі елементи, які задовільняють умову предиката. (предикат – твердження, істинність якого залежить від значення змінних які входять в нього. Інтерфейс Predicate добавлено в Java 8)

**bolean retainAll(Collection<?>c)** – видаляє з колекції всі елементи крім тих, які задані в колексії с. Істина – якщо колекція змінилася.

**int size().**

**default Spliterator<E>spliterator()-**повертає ітератор розділювач для колекції.

**default Stream<E>stream()** – викликає потік, який бере викликаючу колекцію в якості джерела вводу-виводу елементів. Однопоточний, послідовний.

**Object[] toArray()** – копіює елементи колекції в масив типу Object.

**<T> toArray(T[] array)** - копіює елементи колекції в масив тип якого відповідає типу елементів з колекції. Якщо масив замалий для кількості об’єктів – створюється новий масив більшого розміру. Якщо масив завеликий для колекції – зайві ячейки заповняються null. (більш пріоритетний для використання за попередню версію)

По суті в List приватним полем лежить масив певної величини.(саме тому методи List можуть кидати IndexOutOfBoundsException) Коли ми додаємо в кінець елементи методом add() і досягаємо кінця масиву – виділяється новий масив більшої величини, викликається ArrayCopy і значення зі старого масиву копіюються в новий тим самим утворюючи вільні ячейки. Ссилка приватного поля зі старого масиву перекидується на новий, а старий стерає ГК.

Додаткові методи **List**:

**subList**(int startIndex, int endIndex) - повертає фрагмент списку.

**sort()** - сортує список.

Якщо перед нами стоїть задача швидкого виклику значення зі списку по індексу, то краще використовувати ArrayList, а не LinkedList. Оскільки ArrayListпрацює як масив і доступ до довільного індексу відбувається швидко, а в LinkedList потрібно пробігатися по всьому двосвязному списку аби знайти потрібне значення (можна це робити з двох кінців одразу)Взагалі, використовувати LinkedList краще, коли ми маємо справу з кінцями списку, наприклад, з чергою якоюсь(ЛЛ реалізовує інтерфейси Que,Deque–черга і двустороння черга).

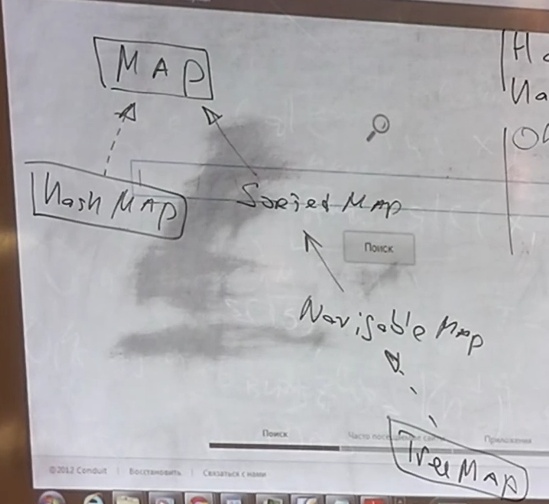
Якщо порівняти метод **remove()** для LinkedList і ArrayList, то отримаємо схожу картину – обидва відпрацюють відносно повільно. ArrayList швидко знайде потрібний індекс, видалить ячейку, але затратить час на копіювання і злиття частин результуючого масиву. В той час LinkedList повільно знайде індекс доки пробіжить від початку чи навпаки, проте швидко перекине ссилки на об’єкти і ГК видалить елемент.

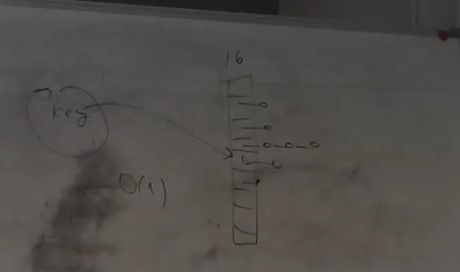
**get()** працює швидко в обох випадках на 0-му елементі. Одинаково на взятті з хвоста. При взятті елемента з середини LinkedList набагато повільніший, оскільки треба пробігтися по всім елементам до середини.

**contains()** працює в ArrayList набагато швидше за LinkedList. В 2-3 раза швидше на 100 000 елементів.Оскільки ArrayList біжить по ячейка масива, а LinkedList виконує постійно метод next().

**add(elem) \ add (elem, index)**в 0-ву позицію (голову) працює набагато швидше в LinkedList, оскільки ArrayList робить новий масив і копіює значення, а LinkedList просто перекидає ссилку. В кінець (хвіст) – одинаково швидко. При вставці елемента в середину LinkedList повільніший, оскільки треба пробігтися по всім елементам до середини і перекинути ссилки – в даному випадку копіювання в ArrayList організовано швидше.

Реалізація **Мар**.





Інтерфейс **Map<K, V>**не входить до колекцій по ієрархії оскільки для метода put(k key, v value), наприклад, йому потрібно 2 значення – ключ і значення. На відміну від List i Set які наслідують інтерфейс Collection. Map схожий до List в якого елементи також лежать по індексу, але відмінність полягає в тому, що ключ в Map може мати довільну форму. Map добавляє елементи, видаляє і отримує швидше за List. При роботі з хеш мапою першим ділом викликається хеш-код ячейки при зверненні до ключа мапи. При вказанні розміра мапи краще намагатися щоб кількість бакетів була рівна кількості елементів в мапі – це сприяє найкращій продуктивності доступу (по дефолту мапа має 16 бакетів в кожному з яких може бути як і жодного значення, так і дкілька зв’язані в лінкед ліст). На відміну від ArrayList, мапа має інший механізм при переповненні – аррай ліст заповняється до кінця, потім виділяється новий більший масив і значення копіюються. В мапі ж є механізм який слідкує за співвідношенням кількості бакетів, до кількості елементівв самій мапі, оскільки в один бакет мапи може добавлятися декілька значень які пов’язуються в лінкед ліст. Це відбувається за допомогою константи DEFAULT\_LOAD\_FACTOR (0,75 по замовчуванню), яка визначає співвідношення між калькістю бакетів (DEFAULT\_INITIAL\_CAPACITY=16;) і кількістю елементів в мапі. Якщо countElem \* 0.75 >D\_I\_C, то виділяється нова мапа розміром size\*2 куди перекидаються данні. Таким чином, в мапі не відбувається повного заповнення як в АррайЛісті. У мапи value не може бути null оскільке таке значення повертає сама мапа, коли по ключу немає значення.

**HashMap** може містити ключ **null** і його хеш завжди буде рівним 0. Дебагер показує, що для null хеш код буде 0.

**HashSet** - та ж сама ситуація, що й **HashMap.**

В **TreeMap** можна покласти єдиний ключ **null**, але після цього операції з мапою (крім size i clear()) не будуть працювати. В непорожню TreeMap покласти ключ null не вийде, оскільки дерево сабалансуюче і буде викликатися метод compareTo() для null. І ми отримаємо NPE.

В **TreeSet** ключом null може бути тільки перший елемент.

Основні методи:

**void put(k, v)** – вставка значення по ключу.

**V get(K key)** – повертає значення типу V по ключу.

**boolean contains(K key)** – перевірє чи є таке значення по ключу.

**void remove(K key)** – видаляє значення по ключу.

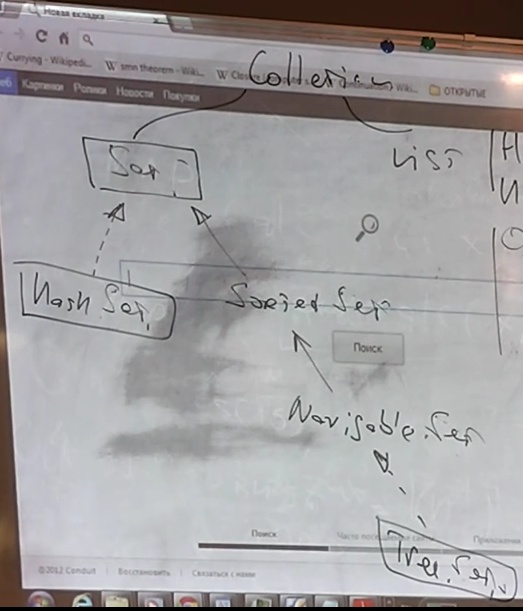
Методи Object:

Якщо ГК копіюючий (як правило всі такі), то він має право проводити дефрагментацію пам’яті після видалення певних об’єктів для того аби не було пробілів вільної пам’яті між об’єктами.

Розглянемо метод int **hashCode()**з яким працюють колекції. Це не адрес об’єкта в пам’яті бо об’єкти можуть переміщатися в пам’яті при дефрагментуючому ГК. Хеш код не обов’язково повинен бути унікальним бо хеш кодів всього є 4 000 000 000 (скільки маєзначення int). То при створенні 4 000 000 001 елемента, наприклад, є як мінімум 2 одинакових хеш кода. Іноді через це бувають колізії ключів – коли по одному хеш коду лежить 2 значення в вигляді лінкед ліста, наприклад, (k1, v1), (k2, v2). HashMap по хеш-коду знаходить.

**==** - перевіряє чи об’єкти ссилаються на один і той же об’єкт. Якщо повертає істину, то equals для цих же об’єктів також верне істину для цих двох ссилок. Їхні хеш коди також будуть одинакові.

**equals**() – перевіряє чи об’єкти є ідентичними. Коли повертає істину, то «==» для цих же об’єктів не завжди може повертати істину, оскільки ссилку можуть вести на ідентичні, але різні об’єкти. Якщо організовувати свій метод equals, то це потрібно робити шляхом порівняння полів одного об’єкта, до полів іншого.

Реалізація **Set** схожа

**interface Set<E> -** інтерфейс в якому визначається група елементів. Не може містити дублікатів, оскільки елементи в сет повинні бути унікальні. Коли ми говоримо про певну кількість унікальних елементів(ключів), то це Set, але якщо до цих ключів добавити значення (які вже можуть дублюватися), то це вже Map. Set при спробі додавання дублюючий елемент в колекцію видасть false.

**Comparable** і **Comparator** – два різних і взаємозамінних інтерфейса. Схожі на пару Iterator – Iterable, але відмінність в тому, що Iterable є нащадком і отримує від Iterator метод iterator().

**interface SortedSet<E> -** розширяє інтерфейсSet і використовується для керування впорядкованими множинами елементів. Методи:

**first**() - повертає перший елемент в сортованій групі елементів.

**last**() - повертає останній елемент в сортованій групі елементів.

**subSet()** - для отримання інтервалу із сортованої групи елементів.

**tailSet() \ headSet()** - повертає інтервал (підмножину) який починається з першого \ останнього елемента вже існуюого інтервалу (підмножини).

**Comparator<? super E> comparator()** - повертає компаратор відсортованої множини.

**interface NavigableSet<E>** - розширяє SortedSet і визначає поведінку колекції, вибір елементів з якої відбувається на основі найбільш точного співпадіння з заданим значенням чи декількома значеннями. Де <E> - тип об’єктів які містяться в колекції. Додатково має методи:

**E celling(E object)** - шукає значення найменшого елемента з умовою e >= object і повертає його. Інакше повертає null.

**E floor(E object)** - шукає значення найбільшого елемента з умовою e <= object і повертає його. Інакше повертає null.

**E lower (E object)** - шукає значення найменшого елемента з умовою e > object і повертає його. Інакше повертає null.

**E higher(E object)** - шукає значення найбільшого елемента з умовою e < object і повертає його. Інакше повертає null.

**Iterator<E> descendingIterator()** - повертає ітератор який обходить колекцію від більшого елемента до меншого. Тобто повертає зворотний ітератор.

**E poolFirst()** - повертає перший елемент попутно видаляючи його. Цей елемент з найменшим значенням, оскільки множина відсортована. Якшо в колекції нема елементів - повертає null.

**E poolLast()** - повертає останній елемент попутно видаляючи його. Цей елемент з найбільшим значенням, оскільки множина відсортована. Якшо в колекції нема елементів - повертає null.

**Інтерфейс Queue<E>** - розширяє інтерфейс Collection, який описує поведінку черги, що діє по принципу "перший зайшов - перший обслуговується". Хоча, сам механізм черги може відрізнятися оскільки є різні види черг, організація і порядок в яких задається критерієм. <E> - тип елементів, які будуть зберігатися в черзі. Методи інтерфейсу:

**E element() -** повертає елемент з голови черги. Елемент при цьому не видаляється. Якщо черга порожня - генерує виключення NoSuchElementException.

**boolean offer(E object) -** вводить елемент в чергу. Якщо елемент успішно добавлено - повертає істинну, якщо ні - фолс.

**E peek() -** бере елемент з голови колекції. Не видаляє елемент з черги. Якщо колекція пуста - видає null. ( на відміну від element() ).

**E poll() -** повертає елемент з голови колекції і видаляє його з черги. Повертає null якщо черга пуста.

**E remove() -** видаляє елемент з голови черги і повертає його значення. Якщо черга пуста - викидає NoSuchElementException.

Черга може кидати ClassCastException якщо елемент, який пробують добавити несумісний з типом елементів в черзі. Якщо відбувається спроба добавлення пустого елемента - кидається NullPointerException оскільки пусті елементи в черзі заборонені. При спробі ввести елемент в заповнену чергу фіксованої величини - вилітає IllegalStateException.

**Інтерфейс Deque<E>** - розширяє інтерфейс Queue тільки може вести себе як двухстороння черга, яка діє не тільки по принципу "перший ввійшов - перший обслуговується", я й на подобі стеку "останній ввійшов - перший обслужився". В двухсторонній черзі пусті елементи також недозволені (NPE). Крім методів Queue ще має:

**void addFirst(E object) \ void addLast(E object) -** додає елемент в голову \ хвіст черги. Кидає ISE, якщо черга фіксованої величини і заповнена.

**Iterator<E> descendingIterator()** - повертає зворотній ітератор, який проходить чергу з хвоста до голови.

**E getFirst() \ E getLast() -** повертає перший \ останнійелемент з черги. При цьому елемент не видаляється.

**boolean offerFirst(E object) \ boolean offerLast(E object)** - робить спробу добавити елемент в голову \ хвіст черги. Повертає фолс при невдачі. Таким чином повертає фолс і при спробі додавання елемент в заповнену двухсторонню чергу (а не ISE).

**E peekFirst() \ E peekLast()** - повертає перший елемент з голви \ хвоста. Не видаляє значення. Повертає null якщо черга пуста.

**E pollFirst() \ E pollLast() -** - повертає перший елемент з голви \ хвоста і видаляє значення. Повертає null якщо черга пуста.

**E pop()** - повертає перший елемент з голови двухсторонньої черги і видаляє його. При пустій черзі генерує NSEE. (NoSuchElementException)

**void push (E object)** - записує елемент в голову черги. Якщо черга заповнена - генерує ISE.

**E removeFirst() \ E removeLast()** - повертає перший \ останній елемент з голови \ хвоста черги і попутно видаляє його з черги. При пустій черзі генерує NSEE.

**boolean removeFirstOccerence(Object object)** - видаляє перший елемент з черги. Повертає фолс, якщо такого object немає.

**boolean removeLastOccerence(Object object)** - видаляє останній елемент з черги. Повертає фолс, якщо такого object немає.

**Клас ArrayList<E>** - (розширяє AbstractList і реалізує інтерфейс List). Клас підтримує динамічні масиви, які можуть збільшуватися\зменшуватися по мірі необхідності на відміну від стандартних масивів, які є незмінними. Не сортує елементи автоматично бо подібний масиву. Конструктори класу:

ArrayList(); // створює пустий масив

ArrayList(int size);

ArrayList(Collection <? extends E> c);

При виводі на консоль автоматично перевизначається метод toString(), що перевизначено в AbsractCollection.

**void ensureCapasity(int size)** - збільшує динамічний масив вручну, якщо ми зазделегідь знаємо його розмір.

**void trimToSize()** - обрізає массив до величини яка відповідає кількості елементів в ній.

**Object[] toArray() \ <T> T[] toArray(T array[])** - приводить динамічний масив до вигляду звичайного масиву. Робиться для підвищення продуктивності в деяких випадках, для передачі масиву об’єктів методам які не перегружаються для прийняття колекції, для суміщення нового коду зі старим, який не підтримує колекції.

ArrayList<Integer> arrayListOfIntegers = **new** ArrayList<Integer>();

arrayListOfIntegers.add(1);

arrayListOfIntegers.add(2);

arrayListOfIntegers.add(3);

arrayListOfIntegers.add(4);

Integer[] simpleArray = **new** Integer[arrayListOfIntegers.size()];

simpleArray = arrayListOfIntegers.toArray(simpleArray);

Колекції можуть зберігати тільки ссилочні типи, а не примітивні значення, але завдяки автоупаковці ми можемо добавляти в колекції примітиви які упаковуються автоматично в об’єкті оболонки.

**Клас LinkedList<E>** - (розширяє клас AbstractSequentalList, реалізує інтерфейси List, Queue, Deque). Організовує структуру звязанного списку. Конструктори:

LinkedList();

LinkedList(Collection<? extends E> c);

**interface SortedSet<E>** і **interface** **SortedMap<E>** самі не вміють сортувати значення, тому вони наслідують інтерфейси:

**Cómparable** і **Compárator**. Взагалі, мапи існують для того щоб визначати значення по ключу без необхідності сортування. Але сортування додає функціональності в плані того, що з відсортованою колекцією можна вирізати діапазон з мапи. Наприклад, з колекція можна витягнути людей, які родилися в якомусь певному місяці. Сортування відбувається в лексикографічному порядку (приклад: A, B, Bca, C, Ccf etc.). А коли колекція не сортована, то ми можемо працювати тільки з точковими значеннями. Приклад:

SortedSet<String>sortSet = newTreeSet<>();*//старий код 1,2 не використовував типізацію <>. Generics ввели пізніше за колекції.*

sortSet.add(“B”);

sortSet.add(“C”);

sortSet.add(“A”);

SOP(sortSet); // виведе [A, B, C]. Те саме і з цифрами.

Додавання різних типів (A, C, 5 ) приведе до ClassCastException.

TreeSet (єдина реалізація сортед сета) наслідує SortedSet, який реалізує Iterator. Тому при виводі TreeSet, на консоль наприклад, методом interator() виводить елементи в наростаючому напрямку, від меншого до більшого. Це відбувається тому, що метод toString для колекцій перевизначено і вивід відбувається за допомогою Iterator.

Будова цикла **for:**

for(statement; boolean; statement){ … }

for( ; ; ){ … } – boolean автоматично рахується як true;

for(int k=1; ; k++) – переповнення int не буває, тому стек не переповниться.

for(float k=1; ; k++) – видасть в кінці Infinity.

Якщо ми хочемо в сортований сет добавляти свою сутність, наприклад клас User з його полями, то потрібно реалізовувати в класі User інтерфейс Comparator.

class User{…}

sortSet.add(new User(25, “Mike”));

SOP(sortSet); // видасть ClassCastException якшо User не наслідує Comparator

…

class User implements Comparable<User>{…} // бо Comparable<T>

sortSet.add(new User(25, “Mike”));

SOP(sortSet); // [25, Mike]. SOPприйдеьбся перевизначити аби не виводило хеш теги замість об’єктів.

Тоді реалізувати метод compareTo()

public int compareTo(User that){

return this.age – that.age; // поверне int який - , 0, + для compareTo

}

protected метод можна зробити public в своєму пакеті, якщо зробити наслідника класу де лежить protected метод і в насліднику змінити його на public.

Всі, хто хочуть бути в TreeSet чи TreeMap повинні бути Comparable щоб використовувати compareTo(); Прои чому 2 об’єкта повинні бути Comparable. Можна завести спеціальний клас для порівняння, наприклад, UserAgeComparator implements Comparator<User>; і перевизначити його метод public int compare(User1, User2){ return user1.age – user2.age; }

TreeSet і TreeMap – бінарні дерева. В більшості БД є індекси і ці індекси влаштовані за принципом B3 - binary tree (Oracle, MySQL, PostgreSQL)? практично як і в TreeSet і TreeMap.

**Бінарне дерево** – структура певного типу, яка має значення (value) і дві ссилки (left, right) на об’єкти такого ж типу як і тип структури, які в свою чергу також мають значення і ссилки вліво і вправо. Приклад:

class Node{

int value;

Node left;

Node right;

}

І ця конструкція (бінарне дерево пошуку) влаштована таким чином, що якщо в самому верхньому Node лежить, наприклад, число 14, то зліва всі потомки дерева будуть містити значення менші за 14, а правий потомок верхнього Node містить значення більші за 14. Таким чином бінарний пошук тут реалізовано в вигляді замороженої структури і нам буде легко шукати потрібний нам елемент відкидаючи непідходящі вітки. Тому по результативності ми приходимо до логарифму. Тобто на 1000 елементів потрібно 10 ділень, на 1 000 000 елементів – 20 ділень, 1 000 000 000 – 30 ділень.Цю структуру потрібно підтримувати в збалансованому стані, так щоб зліва вершини і зправа була одинакова кількість елементів – це відбувається автоматично на кожному методі put() і remove().

Тому в TreeMap додавати елементи по зростанню не вийде, адже мала би утворитися конструкція у вигляді лінкед ліста, но в силу механізму червоно-чорного дерева воно буде балансуватися і на вершині буде, наприклад, 14.

**Клас HashSet<E> -** розширяє клас AbstractSet і реалізує інтефейс Set. Слугує для утворення колекцій, дані яких зберігаються в хеш таблиці. Несортований на відміну від TreeSet. Конструктори:

HashSet();

HashSet(Collection<? extends E> c);

HashSet(int size);

HashSet(int size, float loadFactor); //контроль переповнення (0,1 - 1)

**Хешування** - механізм за допомогою якого зберігаються дані в хеш таблицях (бакетах). Перевага хешування полягає в тому, що він забезпечує постійність роботи методів add(), contains(), remove(), set() на навіть на великих множинах.

**Хеш код** - це індекс по якому зберігаються дані пов’язані з ключом. Перетворення ключа в хеш код відбувається автоматично, хоча сам хеш код є недоступним (але хеш таблицю й так неможна індексувати безпосередньо). Хеш код і equals тісно пов’язані бо хеш код обраховується на основі вмісту об’єкта.

object1.equals(object2)// должно быть true бо реалізація цього метода в класі Обджект порівнює ссилки:

public boolean equals(Object obj) { return (this == obj); }

object1.hashCode() == object2.hashCode()// должно быть true

public native int hashCode(); //оснований на генераторі випадкових чисел Park-Miller

**При роботі з ОРМ** потрібно в методах **hash**() & **equals**() використовувати геттери, а не звертатися до полів напряму, оскільки ОРМ можуть використовувати lazy loading.

**Контракт** **hash**() & **equals**() - якщо у двох чи більше об’єктів одного класу одинаковий вміст, то і хеш-коди цих об’єктів повиині бути одинаковими. Тому для своїх створених об’єктів потрібно перевизначати методи hash & equals() враховуючи поля об’єкту (щоб потім не було проблем з HashMap яка використовує ці методи або з HashSet який може містити тільки унікальні елементи, а з неперевизначеним методом hashCode можна в Сет буде записати 2 одинакових об’єкта).

public boolean equals(Object o) {  
        if(o == null)  
        {  
            return false;  
        }  
        if (o == this)  
        {  
           return true;  
        }  
        if (getClass() != o.getClass())  
        {  
            return false;  
        }  
       
}

@Override  
 public int hashCode()  
 {  
    final int PRIME = 31;  
    int result = 1;  
    result = PRIME \* result + getId();  
    return result;  
 }

**Клас LinkedHashSet** - розширяє клас HashSet. Те ж саме, що й SortedSet тільки підтримує списки. Сортований. Елементи будуть виводитися в тому порядку в якому вони були введені.

**Клас TreeSet<E>** - реалізує клас AbstractSet і реалізує інтерфейс NavigableSet. Створює колекцію для зберігання даних в якій використовується древовидна структура. Відсортований. Час доступу до елементів малий, тому хороший навіть на великих об’ємах. Конструктори:

TreeSet();

TreeSet(Collection<? extends E> c);

TreeSet(Comparator<? super E> comparator);//задній компаратор

TreeSet(SortedSet<E> sortSet );// дерево з елементами SortedSet

Оскільки відсортована, то треба для subSet(); вводити значення в спаданні по розміру. Оскільки від меншого до більшого кине виключення IAE. Приклад:

System.***out***.println(stringHashSet.subSet("Two", "Five"));// помилка бо від меншого до більшого.

System.***out***.println(stringHashSet.subSet("Five", "Five"));//видасть пусті дужки [ ]

**Фабричний метод** - шаблон проектування. Ще називають Virtual Constructor. Призначений для створення одним інтерфейсом об’єктів різного типу. Використовуються, коли з-зарання невідомо об’єкти якого типу потрібно створювати, коли клас спроектовано так, що його об’єкти специфікується підкласами.

**Ітератор** - об’єкт класу ,який реалізує або інтерфейс Iterator<E>, або інтерфейс ListIterator<E>. Ітерфейс Iterator дозволяє організувати цикл для перебору колекції де можна буде витягувати чи видаляти елементи. А інтерфейс ListIterator розширяє інтерфей Iterator і дозволяє організувати цикл двухстороннього обходу колекції. В кожному класі колекції є свій іератор ,який повертає ітератор на початок колекції. Використовуючи об’єкт ітератора можна получити доступ до всіх елементів колекції по черзі - установлюється ітератор на початок колекції, організовується цикл в якому викликається метод hasNext() і виконується повтор циклу поки метод hasNext() не поверне фолс, получити кожний елемент колекції методом next(). Колекції, які реалізують інтерфейс List можуть отримати ітератор для себе за допомогою метода listIterator() який працює в дві сторони.

Методи інтерфейсу Iterator<E>:

**default void forEachRemaining(Consumer<? super E> action)** - виконує заданий action над кожним необробленим елементом колекції.

**boolean hasNext()** - повертає істину, якщо в колекції ще є елементи, інакше - фолс.

**E next()** - повертає наступний елемент з колекції. Кидає NSEE, якщо достигнуто кінця колекції.

**void remove()** - видаляє поточний елемент з колекції. Кидає IllegalStateException, якщо перед викликом метода remove не було викликано метод next().

Методи інтерфейсу **ListIterator<E>**:

**void add(E object)** - вводить елемент в колекцію перед елементом, що виддається методом next() в даний момент. Тобто по порядку.

**default void forEachRemaining(Consumer<? super E> action)** - див вище.

**boolean hasNext();**

**boolean hasPrevious() -** перевіряє колекцію на наявність попереднього елемента. Повертає істину\фолс.

**E next()** - повертає наступний елемент з колекції. Кидає NSEE, якщо достигнуто кінця колекції.

**E previous();**

**int nextIndex() \ int previousIndex() -** повертає індекс наступного елемента в списку,якщо дійшов до кінця - показує довжину списку \ повертає попередній елемент в списку, якщо попереднього нема - повертає -1.

**void remove()** - видаляє поточний елемент з колекції. Кидає IllegalStateException, якщо перед викликом метода remove не було викликано метод next() або previous().

Приклад:

ArrayList<String> stringArrayList = **new** ArrayList<String>();

stringArrayList.add("One");

stringArrayList.add("Two");

stringArrayList.add("Three");

stringArrayList.add("Four");

System.***out***.println("List contains: ");

Iterator<String> strIter = stringArrayList.iterator();

**while**(strIter.hasNext()){

String element = strIter.next();

System.***out***.println(element + " ");

}

ListIterator<String> listStrIter = stringArrayList.listIterator();

**while**(listStrIter.hasNext()){

String element = listStrIter.next();

listStrIter.set(element + " + ");

}

System.***out***.println("\nList changed: ");

strIter = stringArrayList.iterator();//беремо знову

**while**(strIter.hasNext()){

String element = strIter.next();

System.***out***.println(element + " ");

}

System.***out***.println("\nRoll back list: ");

**while**(listStrIter.hasPrevious()){

String element = listStrIter.previous();

System.***out***.println(element + " ");

}

Якщо ми не плануємо видозмвнювати колекцію чи обійти її з кінця в початок, а хочемо просто перебрати її, тоді можна просто використати цикл в стилі for each оскільки всі колекції реалізують інтерфейс Iterable, що й треба для роботи циклу for each.

ArrayList<Integer> listOfIntegers = **new** ArrayList<Integer>();

listOfIntegers.add(1);

listOfIntegers.add(2);

listOfIntegers.add(3);

listOfIntegers.add(4);

System.***out***.println("List contains: ");

**for**(**int** value : listOfIntegers){

System.***out***.println(value + " ");

}

**int** summ = 0;

**for**(**int** value : listOfIntegers){

summ += value;

}

System.***out***.println("\nSumm of elements: " + summ);

В JDK8 добавлено **ітератор-розділювач** який визначено в інтерфейсі **Spliterator<E>**. Мають той же функціонл, що й Iterator\ListIterator але підтримують паралельну ітерацію колекцій, що дозволяє використовувати їх ефективно в багатопотоковості.

Інтерфейс **RandomAccess** – забезпечує ефективний довільний доступ до елементів колекції. Не містить жодного члена. Маркед інтерфейс, що сповіщає про можливість довільного і ефективного доступу до елементів колекції. Щоб перевірити чи підтримує колекція даний інтерфейс потрібно використовувати оператор instaceof.

**Comparators**.

TreeSet і TreeMap зберігаються об’єкти в відсортованому порядку. По замовчуванням, ці класи використовують «природнє сортування», що означає сортування по зростанню (1, 2, 3… А, Б, В…). Якщо ж нам потрібно відсортувати відображення чи множину іншим чином, то потрібно вказати інший Comaprator який дає можливість точно кертувати порядком збереження елементів в відсортованих колекціях.

**interface Comparator<T>.** До JDK8 мав 2 методи:

**int compare(Object obj1, Object obj2);** // -1, 0, 1

**boolean equals(Object obj); //** true якщо об’єкт також відноситься до типу Comparator і впорядковано за тим же принципом. Перевизначати його неприйнято.

Приклад зворотнього компаратора:

**import** java.util.\*;

**public** **class** MyComp **implements** Comparator<String>{

@Override

**public** **int** compare(String o1, String o2) {

String aStr, bStr;

aStr = o1;

bStr = o2;

//reverse comparing cause bStr comparing to aStr

**return** bStr.compareTo(aStr);

}

}

---------------------------------------------------------------------

**import** java.util.\*;

**public** **class** CompDemo {

**public** **static** **void** main(String[] args) {

TreeSet<String> ts = **new** TreeSet<String>(**new** MyComp());

ts.add("B");

ts.add("F");

ts.add("A");

ts.add("E");

ts.add("D");

**for**(String element: ts){

System.***out***.print(element + " \n"); //F E D B A

}

}

}

-------------------------------------------------------------------------------------------------------

**Потоки «демони»** - потоки, які працюють для обслуговування інших потоків. Приклад: garbage collector і іншу службові потоки створені ВМ. Джава програма завершає свою роботу, коли завершені всі її потоки крім потоків демонів. Щоб зробити власний потік демоном - Thread.setDaemon(). Приклад – потоки з таймерами для інших потоків.

**Thread.currentThread().join()** – найпростіша конструкція, яка «вішає» ВМ.

**Causality loop**– цикли причинності. Для того щоб щось виповнилось потрібно щоб щось виповнилось.

З статичних методів (**static**) можна викликати тільки статичні методи.

**synchronized()** без wait(), notify() і notifyAll() використовувати можна, а wait(), notify() і notifyAll() без **synchronized()** – ні.

**class** A {  
 **synchronized void** method() { }  
 *//****TODO***}

**native** – метод без реалізації (якщо зайти в сорци). Його реалізувала ВМ або ОС. (Можливо на С, С++).

**null** не зберігається в **heap**. При створенні ссилочної змінної, яка не ініціалізована або рівна null. Спеціальне значення null зберігається біля змінної в стеку.

При створенні двох і більше змінних силочного типу, які вказують на один і той же об’єкт – в heap не створюються нові об’єкти, а ссилки від усіх змінних з стека ведуть до цього єдиного о’єкта.

Якшо синхронізуватися по одному об’єкту ( synchronized(newObject()) ), а метод wait() викликати в іншого нового об’єкта – отримаємо екскпшн. (**IllegalMonitorStateExceptions**)

Щоб викликати нестатичний метод ( f() ) в статичному класі (MyClass), потрібно створити новий екземпляр цього ж статичного класу (newMyClass() ) і тоді викликати в нього нестатичний метод (new MyClass().f() ).

У статичного метода немає **this**.

Можна робити ссилки на самі класи. (**Class clas = new MyClass.class**)

**Закон Деметри –** модулі не повинні знати про внутрішню логіку інших об’єктів з якими вони працюють.

**DTO (Data Transfer Object)** – клас з відкритими змінними і без функцій.

**Mutual exclusion (mutex, взаємне виключення)**– ніякі два чи більше потоків не можуть одночасно працювати в одній синхронізованій секції ініційованій однією змінною. Якщо декілька потоків намагаються одночасно викликати один метод рublic synhronized void(int x);, то входження потоків в цей метод буде поодиночне. Тобто, тільки один поток може працювати з методом, всі інші в цей час чекатимуть на його вихід з методу. Тоді наступний потік заходить в метод і т.д. Виключення – коли в методі є конструкція wait(). Тоді інший потоік зможе зайти в метод поки перший спить.Тобто wait() призупиняє роботу одного потока дозволяючи іншим зайти в метод. Перший потік спатиме доки не буде викликано метод notify() для даного синхронізованого об’єкту.

sleep() не відпускає секцію synhronized для входу іншими потоками, wait() – відпускає.

**Race condition** – стан, коли декілька потоків намагаються захопити монітор.

**deadlock** – взаємне блокування потоками одне-одного. (очікування процесами ресурсів, які зайняті самими ж процесами).

**Взаємна блокіровка (deadlock)** – відбувається, коли потоки мають циклічну залежність від пари синхронізованих об’єктів. Наприклад, перший потік зайшов в монітор об’єкта Х, другий поток зайшов в монітор об’єкта У. Якщо потік в об’єкті Х спробує викликати синхронізований метод об’єкта У, то він буде заблокований монітором. А якщо потік об’єкта У, в свою чергу, спробує викликати синхронізований метод об’єкта Х, то він очікуватиме вічно, оскільки для отримання доступу до об’єкта Х, йому треба зняти свою блокіровку з об’єкта У що б перший потік міг працювати. Дедлок можливий і з одним потоком. (**currentThread().join()** ). Одне з вирішень - використовувати java.util.concurrent.lock де є метод **reentrantLock**, який має ту ж семантику блокування і семантику пам’яті що й synchronized але надає додаткові можливості. Напрклад, встановлює час. Приклад:

**public class** Deadlock {  
 **public static** Object *lock1* = **new** Object();  
 **public static** Object *lock2* = **new** Object();  
  
 **public static void** main(String[] args) {  
 ThreadDemo1 thread1 = **new** ThreadDemo1();  
 ThreadDemo2 thread2 = **new** ThreadDemo2();  
 thread1.start();  
 thread2.start();  
 }  
  
 **private static class** ThreadDemo1 **extends** Thread {  
 **public void** run() {  
 **synchronized** (*lock1*) {  
 System.***out***.println(**"Thread 1. Holding lock1."**);  
 **try** {  
 *sleep*(10);  
 } **catch** (InterruptedException e) {  
 System.***out***.println(**"Interrupted in thread1. "** + e);  
 }  
 System.***out***.println(**"Thread 1. Waiting for lock2."**);  
 **synchronized** (*lock2*) {  
 System.***out***.println(**"Holding lock1 & lock2"**);  
 }  
 }  
 }  
 }  
  
 **private static class** ThreadDemo2 **extends** Thread {  
 **public void** run() {  
 **synchronized** (*lock2*) {  
 System.***out***.println(**"Thread 2. Holding lock2."**);  
 **try** {  
 *sleep*(10);  
 } **catch** (InterruptedException e1) {  
 System.***out***.println(**"Interrupted int Thread2. "** + e1);  
 }  
 System.***out***.println(**"Thread 2. Waiting for lock1."**);  
 **synchronized** (*lock1*) {  
 System.***out***.println(**"Holding lock2 & lock1"**);  
 }  
 }  
 }  
 }  
}

//Thread 2. Holding lock2.

//Thread 1. Holding lock1.

//Thread 2. Waiting for lock1.

//Thread 1. Waiting for lock2.

**"Livelock" (один потік-один метод-два об’єкти, два потоки-один метод-два об’єкти)**- тип взаимной блокировки, при котором несколько потоков продолжают свою работу, но попадают в зацикленность при попытке получения каких-либо ресурсов. Фактической ошибки не возникает, но КПД системы падает до 0. Часто возникает в результате попыток предотвращения deadlock'а.  
  
*Пример:* Метод пытается выполнить какую-либо работу, используя 2 внешних объекта. Сперва он получает блокировку по одному из объектов, а затем проверяет, свободен ли второй объект. Если объект свободен - получает блокировку по нему и выполняет работу, если занят - освобождает первый объект и ждёт, когда они оба освободятся. 2 потока одновременно вызывают этот метод. Поток 1 блокирует первый объект. Поток 2 блокирует второй объект. Оба проверяют, свободен ли второй ресурс - обнаруживают, что он занят и освобождают занятый ресурс. Оба потока обнаруживают, что оба ресурса свободны и начинают процесс блокировки сначала.

В роботі з будь-яким ссилочним типом даних в heap пов’язані 3 речі. **1. Блокування (монітор), 2. blocked-set**(не активні потоки з причини, що їм заважає блокування(монтіор))**, 3. wait-set**(пасивні потоки які сплять і чекають поки їх розбудять). Коли потік заходить в синхронізований метод він спочатку перевіряє стан Блокування. Якщо секція не заблокована, потік блокує її і продовжує роботу. Коли приходить слідуючий потік в цю ж синхронізовану секцію, він не може ввійти в неї оскільки перший потік знаходиться в середині методу і зайняв блокування. Тоді цей другий потік попадає в **blocked-set.** Приходить третій потік, також не може зайти в синхронізовану секцію і поміщається в blocked-set до другого потоку. Так в кучу **blocked-set** добавляються третій, четвертий і т.д. потоки. Як тільки перший потік виходить за фігурну скобку синхронізованої секції, знімається блокування перед нею і ВМ з **blocked-set** вибирає будь-який один з потоків які очікують (**Race condition**). Той займає блокування і процес повторюється. Якщо якийсь з методів, який ввійшов в синхронізовану секцію(СС) самостійно викликає метод wait() , то при цьому знімається блокування на вхід в СС для іншого потоку, а потік, який викликав wait() поміщається в **wait-set** і чекає поки його розбудять. Коли викликається notify() аби розбудити потік з **wait-set**, то випадково вибраний потік з **wait-set** знову переводиться в секцію **blocked-set.** При notifyAll() всі потоки з wait-set переводяться в blockeds-set.

**Преривання** - це сигнал потоку зупинити своє виконання і дії і зробити щось інше. Це зроблено для того, аби ми могли вказати як потік має вести себе при ситуації, коли його буде перервано.

Методи, які кидають **InterruptedException**: wait(), sleep(), join().

**InterruptedException** вилітає, коли потік чекає, спить чи занятий іншим чином і його переривають до чи під час його активності. Також, може бути метод який перевіряє чи перервано виконання потоку, і якщо потік перервано, то кидається InterruptedException.

if (Thread.interrupted()) // Clears interrupted status!

throw new InterruptedException();

**interrupt()** – метод класа Thread, який дозволяє перервати роботу потока. Виставляє флаг перерваності в істину. Метод не є блокуючим. На відміну, наприклад, від wait(). Тобто метод дасть сигнал на завершення якогось потоку і не чекатиме доки той завершиться, а просто продовжить роботу далі.

Thread thread = new Thread(new Runnable{});

thread.interrupt();

**interrupted() -** перевіряє флаг чи перервано поточний потік.

**isInterrupted()** – флаг який вказує чи перервано потік типу Thread який його викликає.

Thread myThread = new Thread.currentThread();

SOP( myThread.isInterrupted() );

Флаг потрібен щоб ми могли виконувати роботу доки потік не перервано. Для цього установлюємо в циклі умову.

Thread myThread = new Thread.currentThread();

while( !myThread.isInterruted() ) { **return**; // or do something }

isInerrupted() флаг виставляяється в умові циклу що дозволяє доводити до кінця математику при кожній ітерації циклу навіть якщо потік перервали в момент, коли математика знаходится посередині виконання. Тоді монітор буде звільнено для інших потоків. П.С. Метод destroy() може перервати математику в циклі посеред виконання і монітор так і не звільниться ніколи.

InterruptedException може вилітати, коли потік в блокованому стані методами sleep(), join(), wait(), а другий потік викликає на цьому потоці interrupt(). При чому методи першого потку завершуються.

|  |  |
| --- | --- |
| Різниця між методами | |
| **isInterrupted()** | **interrupted()** |
| Не static (instance) | static |
| isInterrupted() перевіряє чи перервано потік інстанса типу Thread, який його викликав | interrupted() перевіряє тільки чи перервано поточний потік. |
| Не міняє флаг. Якщо потік перервано, то флаг установлюється true і зберігається в такому вигляді. Тобто скільки раз ми будемо давати запит – буде одна й та ж відповідь. Що і є притаманним еталонному геттеру. | Міняє флаг. Не притаманне еталонному геттеру. Може кожного разу повертати різні значення флага. Фактично, метод запитує чи був він перерваний з моменту, коли він про це питав останнього разу. Тобто, якщо попередній раз він повернув true, то наступного разу він верне false. |
| Узгоджений code convention | Неузгоджений code convention бо схожий на get() але міняє значення, що не притаманно еталонному get() |
|  |  |

**Поток** в якого **флаг перерваності** (флаг, який вказує на те чи зевершили примусово потік) встановлено **true** не може заснути(**sleep()**) - вилітає **InterruptedException.** Якщо перед засипанням потоку поставити **interrupted(),** то він **очищає флаг** і потік зможе заснути.

Потік можна реалізувати двома методами: реалізуючи інтерфейс Runnable (**implements Runnable**) в якому перевизначати метод run() і задати цей об’єкт в конструктор Thread(). І другий метод - розширивши клас Thread. (**extends Thread** і переоприділяючи метод run() )

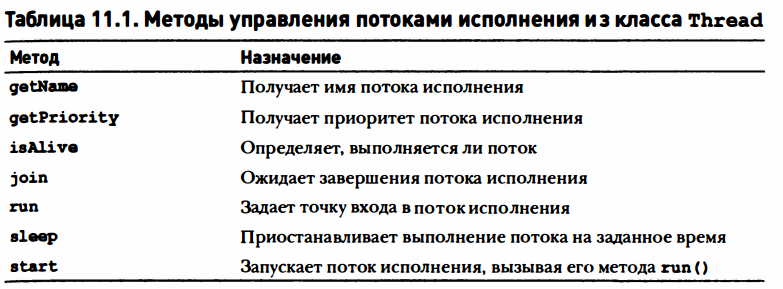
**1. public class** HelloRunnable **implements** Runnable {  
  
 **public static void** main(String args[]) {  
 (**new** Thread(**new** HelloRunnable())).start();  
 *// те ж саме, що HelloRunnable helloRunnable = new HelloRunnable();  
 // helloRunnable.run();* }  
  
 **public void** run() {  
 System.***out***.println(**"Hello from a thread!"**);  
 }  
  
}

**2. public class** HelloThread **extends** Thread {  
  
 **public static void** main(String args[]) {  
 HelloThread thread = **new** HelloThread();  
 thread.start();

*// або в одну строку (new HelloThread()).start();*  
 }  
  
 **public void** run() {  
 System.***out***.println(**"Hello from a thread!"**);  
 }  
  
}

Thread сам по собі implements java.lang.Runnable.

Щоб запустити потік потрібно створити об’єкт Runnable і помістити його в конструктор Thread. Далі запустити. (start() ) Thread t = newThread( newRunnable() ). По суті, метод start() запускає метод run() в Runnable.



Приклад implements Runnable:

**public class** CurrentThreadDemo {  
 **public static void** main(String[] args) {  
 **new** NewThread();

*//можна запустити декілька потоків задавши в конструкторі name*

*// new NewThread("Second");  
 // new NewThread("Third");*

**try** {  
 **for** (**int** i = 5; i > 0; i--) {  
 System.***out***.println(**"General thread: "** + i);  
 Thread.*sleep*(1000);  
 }  
 } **catch** (InterruptedException e1) {  
 e1.printStackTrace();  
 }  
 System.***out***.println(**"General thread finished."**);  
 }  
}

**public class** NewThread **implements** Runnable {  
 Thread **t**;  
  
 NewThread() {  
 **t** = **new** Thread(**this**, **"Second thread"**);  
 System.***out***.println(**"Second thread created."** + **t**);  
 **t**.start();  
 }  
  
 @Override  
 **public void** run() {  
 **try** {  
 **for** (**int** i = 5; i > 0; i--) {  
 System.***out***.println(**"Second thread: "** + i);  
 Thread.*sleep*(500);  
 }  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.***out***.println(**"Second thread finished."**);  
 }  
}

//Second thread created.Thread[Second thread,5,main]

General thread: 5

Second thread: 5

Second thread: 4

General thread: 4

Second thread: 3

Second thread: 2

Second thread: 1

General thread: 3

Second thread finished.

General thread: 2

General thread: 1

General thread finished.

Приклад extends Thread:

**public class** NewThread **extends** Thread {  
  
 NewThread() {  
 **super**(**"Second thread!"**);  
 System.***out***.println(**"Second thread created!"** + **this**);  
 start();  
 }  
  
 @Override  
 **public void** run() {  
 **try** {  
 **for** (**int** i = 5; i > 0; i--) {  
 System.***out***.println(**"Second thread: "** + i);  
 Thread.*sleep*(1000);  
 }  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.***out***.println(**"Second thread finished!"**);  
 }  
}

**public class** CurrentThreadDemo {  
 **public static void** main(String[] args) {  
 **new** NewThread();  
  
 **try** {  
 **for** (**int** i = 5; i > 0; i--) {  
 System.***out***.println(**"General thread: "** + i);  
 Thread.*sleep*(500);  
 }  
 } **catch** (InterruptedException e1) {  
 e1.printStackTrace();  
 }  
 System.***out***.println(**"General thread finished!"**);  
 }  
}

Коли Джава програма запускається починає виконуватися один потік - його називають головним потоком. Він породжує інші потоки і, як правило, має завершатися останнім з у усіх потоків.

Thread thread = Thread.*currentThread*();

Багатозадачність буває основаною на процесах і основаною на потоках.

В Джава потоки отримують приорітет по відношення до інших, який записується цілим числом. Звідси й походить **витісняючи багатозадачність. (1-10. static final int** MIN\_PRIORITY=1, NORM\_PRIOITY=5, MAX\_PRIORITY=10**).** Релізація механізму пріоритетності потоків залежить від версій ВМ і ОС, тому іноді може різнитися. **final int getPriority()** –отримати приорітет, **final void setPriority(int level)** –задати пріоритет потока. Сумнівний по ефективності бо потоки з меншим пріоритетом можуть голодати.

**Монітор** – конструкція, яка приймає тільки один потік в одиницю часу. Як тільки один потік зайшов в монітор і «зайняв» його. Всі інші потоки чекають поки перший потік не вийде з монітора і аж тоді продовжують роботу.

**final void wait()** – призупиняє потік який його викликав і віддає монітор іншому потоку до тих пір поки інший потік не зайде в монітор і не викличе метод notify(). (IE)

**final void notify()** – відновлює роботу потоку, який викликав wait() в тому ж об’єкті.

**final void notifyAll()** - відновлює роботу всіх потоків, які викликали wait() в тому ж об’єкті.

Ці три методи реалізовані як фінальні в класі Object, тому вони доступні всім класам. wait(), notify() notifyAll – нейтівні методи.

**yield()** - віддає в потоці квант часу процесора іншому потоку, а даний потік переміщається вниз в чергу потоків. Сумнівний по дії і функіонуванню, як і задавання приорітетів.

**final void join()** – змушує викликаючий метод потік чекати на завершення потоку, що вказано.

Thread thread = new Thread(new Runnable{});

thread.start();

thread.join(); //метод main буде чекати на завершення потоку thread.

**final Boolean isAlive()** – перевіряє чи вказаний потік не перервано. Повертає істину, якщо потік ще працює.

Приклад використання методів **join**() і **isAlive**():

**public class** NewThread **implements** Runnable {  
 String **threadName**;  
 Thread **thread**;  
  
 NewThread(String threadName) {  
 **this**.**threadName** = threadName;  
 **thread** = **new** Thread(**this**, threadName);  
 System.***out***.println(**"New thread: "** + **thread**);  
 **thread**.start();  
 }  
  
 @Override  
 **public void** run() {  
 **try** {  
 **for** (**int** i = 3; i > 0; i--) {  
 System.***out***.println(**"Second thread: "** + i);  
 Thread.*sleep*(1000);  
 }  
 } **catch** (InterruptedException e) {  
 e.printStackTrace();  
 }  
 System.***out***.println(**"Second thread finished!"**);  
 }  
}

**public class** CurrentThreadDemo {  
 **public static void** main(String[] args) {  
 NewThread one = **new** NewThread(**"One"**);  
 NewThread two = **new** NewThread(**"Two"**);  
 NewThread three = **new** NewThread(**"Three"**);  
  
 System.***out***.println(**"Does 3 threads is running: "** +  
 one.**thread**.isAlive() + **", "** +  
 two.**thread**.isAlive() + **", "** +  
 three.**thread**.isAlive());  
  
 **try** {  
 one.**thread**.join();  
 two.**thread**.join();  
 three.**thread**.join();  
 } **catch** (InterruptedException e1) {  
 e1.printStackTrace();  
 }  
  
 System.***out***.println(**"Does 3 threads is running: "** +  
 one.**thread**.isAlive() + **", "** +  
 two.**thread**.isAlive() + **", "** +  
 three.**thread**.isAlive());  
  
 System.***out***.println(**"General thread finished!"**);  
 }  
}

//New thread: Thread[One,5,main]

New thread: Thread[Two,5,main]

New thread: Thread[Three,5,main]

Second thread: 3

Does 3 threads is running: true, true, true

Second thread: 3

Second thread: 3

Second thread: 2

Second thread: 2

Second thread: 2

Second thread: 1

Second thread: 1

Second thread: 1

Second thread finished!

Second thread finished!

Second thread finished!

Does 3 threads is running: false, false, false

General thread finished!

**public static native void sleep(long millis)** - змушує потік призупинитися на кількість мілісекунд. (IE)

**try** {  
 **for** (**int** i = 0; i < 5; i++) {  
 System.***out***.println(i);  
 Thread.*sleep*(1000);  
 }

**volatile** – параметр змінної(методу), який дає інструкцію процесору читати значення змінної з ОЗУ, а не з її керованої версії в кеші процесора. Таким чином всі потоки, які оперують даною змінною, працюють з однією областю пам’яті де знаходиться змінна. (застосовуються зазвичай для флагів між потоками)

**happen-before -** принцип, який говорить, що виконання інструкцій в часі є прямолінійно поступовим. Тобто, інструкції виконуються «зверху вниз» і код в певний момент часу знає всі інструкції і дії які виконувалися до нього, але не знає про події, які мають відбутися після нього.

**Spurious wakeup** (хибне пробудження) – ефкт, коли об’экт з wait самовільно прокидається просто так, без вказівок. (пов’язано з апаратним перериванням на деяких ОС)

**synhronized** може обгортати змінну (яка не повинна бути null бо синхронізація відбувається по об’єкту в хіпі а не стеку. Відповідно, якщо два об’єкта в стеку мають посилання на один об’єкт в хіпі, то можна сихронізуватися по буд-якому з них), або метод. Приклад: synhronized(ref), public synhronized void main();

Можливі вкладені synhronized.

Віртуальна машина запам’ятовує кількість входження потока в синхронізовані секції.

В синхронізованій секції можна викликати **this.notify**, **this.notifyAll**.

Передавати чи повертати з методів об’єкт який рівний null–вважається поганою практикою. Краще кинути виключення.

Integer etc. – ссилочні типи. Тому вони можуть мати значення null.

destroy(), suspend(), resume, stop() - @deprecated методи. При чому stop() мягший за dostroy. stop() кидає ThreadDeath яке є Error.

ВМ працює тільки з іменованими сутностями. Саме тому компілятор анонімним об’єктам дає ім’я. (Приклад: $1)

З виставленим флагом true метод sleep() у потока викликати вже неможливо. Бо перерваний потік не може заснути. Вилітає ІЕ. Щоб обманути компілятор, можна перед sleep() викликати interupted() який в силу свого дизайну міняє флаг на протилежний.

**Thread.State** **getState()** – метод для отримання стану потока. Визначений в класі Thread з параметрами в перечисленнях. Thread.State getState(); Можливі значення перечислень (іменованих констант): NEW (ще не запущено), RUNNABLE (виконується), BLOCKED (призупинено бо чекає на блокування), TERMINATED (закінчив виконання), TIMED\_WAITING (призупинив виконання на певний час. Наприклад після sleep(millis), join(), wait() ), WAITING (зупинив виконання бо чекає на виконання якоїсь дії. Приклад wai(), join() ).

Thread thr = new Thread(Runnable);

Thread.State ts = thr.getState();

if(thr == Thread.Stete.RUNNABLE) // …

------------------------------------------------------------------------------------------------------------------------------

**Паттерни**:

**Singleton** - гарантує, що клас матиме тільки один екземпляр і надає до нього глобальну точку входу. Методи статичні, змінні і конструктор приватні. Має проблеми з багатопоточністю, з lazy loading etc.

**Adapter/Decorator.**

**Observer (наглядач) -** дозволяє екземпляру класа отримувати сповіщення зі зміною стану від інших класів тим самим наглядаючи за ними. Тобто якщо один об’єкт міняє стан, то всі об’єкти які залежать від нього - сповіщаються про це. Зазвичай використовують для реалізації:

**interface Obervable**

**interface Observer**

**class ConcreteObservable** implements Obervable

**class ConcretObserver** implements Observer

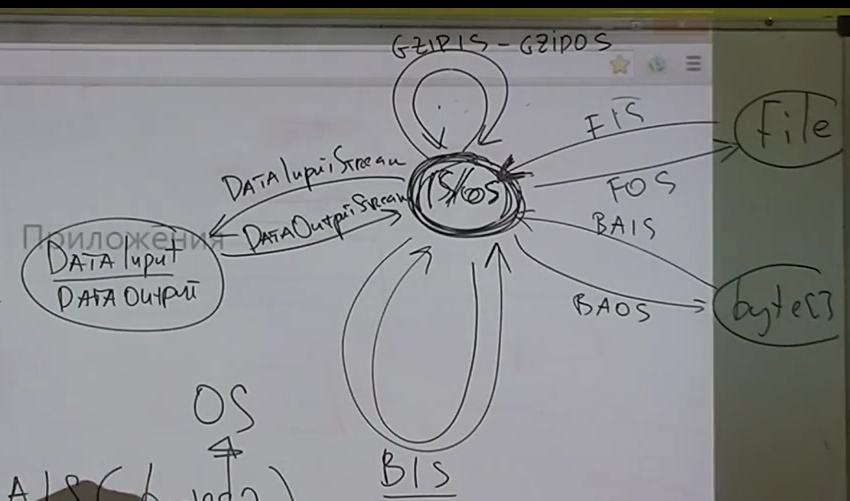
**Factory Method (Фабрика)** - використовується для створення екземпляра конкретного об’єкта, який відноситься до якогось загального інтерфейсу. Тобто в момент створення підкласи цього інтерфейсу можуть визначити екземпляр якого класу створювати.

**Facade(Фасад)** - дозволяє приховати всю складність системи шляхом зведення всіх можливих викликів до одного об’єкту, який вже делегує їх відповідним об’єктам системи. Використовується, коли політика по відношенню до іншої групи об’єктів повинна бути більш відкритою, а коли потрібна більш скрита і акуратна політика відношення - використовують паттерн Proxy.

**Proxy (Заступник)** - виконує функцію контейнера. Перехоплює всі запити до іншого об’єкту і контролює їх.

**Strategy(Стратегія)** - дозволяє використовувати різну логіку і алгоритми залежно від контексту поточного стану системи.

-------------------------------------------------------------------------------------------------------------------



Грубо кажучи в JavaIO можна розглядати 3 головних розділи: **InputStream/OutputStream**, **DataInput/DataOutput**, **Reader/Writer**.

DataOutput не має методів flush() і close(). Тому потрібно окремо створювати OutputStream і передавати ссилку на нього в конструктор DataOutput.

При роботі з ІО ми не завжди знаємо кількість даних які нам мають прийти (наприклад з інтернету), то му потрібно ставити перевірку на відсутність даних (-1).

ІС і ОС не мають розміру. Також ми не можемо взяти і прочитати і-й байт. Беремо їх послідовно.

Є методи які не можна використовувати в багатопоточності двома і більше потоками попередньо не синхронізувавши їх. Є ж методи, які можна використовувати в багатопотоковості. Як правило це вкзано в документації до них і вони називаються **thread safe methods**. Є ж методи **thread local**, які забороняється використовувати навіть при синхронізації потоків.

BufferedInputStream буферизує шматок даних навіть якщо йому дали запит на один байт. Це робиться для того аби пришвидшити наступні можливі запити до буфера.

BufferedReader i BufferedWriter декоратори класів Reader i Writer.

При роботі з DataInput та DataOutput потрібно пам’ятати в якій послідовності ми пишемо різні ти пи даних (int, short, String) в файл, щоб потім їх коректно вичитати з цього ж файла. Це зумовлено тим, що типи даних які ми пишемо в файлі представляють собою просто суцільну послідовність байт.

Для ІС потрібно робити close(), для ОС flush() iclose().

**abstract InputStream** – (реалізує AutoClosable, Closable) вхідний потік читання (newFileInputStream(“C:/MyFile”)), по URL (new URL(“www.google.com”).openStream()), масиву байт (new ByteArrayInputStream(new byte[] {2, 3…} )) тощо.Методи для зчитування **abstract** int **read() –** приводить зчитаний байт до інта**,** int read(byte[] b) – читає байти в масв і повертає кількість прочитаних байтів, int read(byte[] b, int off, int len) – читає побайтно. Якщо ми зчитуємо int, то при виводі його на екран, його треба привести до (char) – InputStream in; int oneByte = in.read(); SOP( (char)oneByte);

Читати дані можна шляхом:

int oneByte;

while ((oneByte = in.read) != -1) {

SOP( (char) oneByte)

}

З буфером в вигляді масиву байт:

InputStraem in;

byte[] buff = new byte[5];

int count;

while(count = in.read(buff) )!= -1){

SOP(new String(buff, 0, count, “UTF-8”));

}

Зробивши один раз read ми можемо декілька раз робити write оскільки байти в ІС зберігаються.

int **available()** – повертає кількість байт для вводу доступних на даний момент для читання. Не єточним оскільки ми не завжди знаємо скільки даних ми маємо отримати. Так і написано в Джава доці.

void **mark(**int кількБайт**)** – ставить мітку в поточну на даний момент точку вхідного потоку, яка залишається актуальною, поки не прочитається певна кількБайт.

void **reset()** – скидає вхідний вказівник на встановлену раніше мітку.

boolean **markSupported()** – повертає truе, якщо викликаючий потік підтримує методи mark() і reset().

long **skip(**long кількБайт**)** – пропускає кількість байт заданих в кількБайт. Повертає кількість дійсно проігнорованих байт.

В IO зчитані байти приводяться до типу int. При чому в IO йде відповідність байта до інта в сегменті 0-255. (Хоча інт має значення від -2 біліонів до 2 біліонів, а байт значення від -128 до 127 – враховується 0). Байт – комбінація бітів. Бітів є 8. Тобто це комбінація з 1 і 0: 01101001. Тобто 2 значення (0 і 1) у 8 степені = 256.

Щоб InputStream було видно і в try i в finally, його потрібно ініціювати перед початком try. Інакше InputStream не буде видно в finally.

Клас **FileInputStream** (extends InputStream) – створює об’єкт класа InputStream. Використовується для читання байтів з файлу. Конструктори:

FileInputStream(StringшляхДоФайлу);

FileInputStream(Fileоб’єктФайла);

Приклад: FileInputStream fInput = new FileInputStream(String шляхДоФайлу);

Клас **ByteArrayInputStream**(extends InputStream) – реалізує вхідний потік, який в якості даних бере масив байтів. Метод close() для нього не викликається, хоча це не буде помилкою. Конструктори:

ByteArrayInputStream(byte[] масив);

ByteArrayInputStream(byte[] масив, intпочаток, intкількБайт);

Приклад: String tmp = “Hello”;

byte[] byteArr = tmp.getBytes();

ByteArrayInputStream b = new ByteArrayInputStream(byteArr);

Клас **FilterInputStream**(extends InputStream)–сам абстрактний клас не використовується в додатках Джава, більш корисні методи є в класах, які його розширяють. Переоприділяє методи InputStream . Конструктор:

FilterInputStream(InputStream is**);**

Клас **BufferedInputStream**(extends FilterInputStream) - буфер, який дозволяє читати більше ніж один байт за раз. Дозволяє заключити в оболочку об’єкти InputStream, тим самим збільшити продуктивність його роботи. Конструктори:

BufferedInputStream(InputStream is**);**

BufferedInputStream(InputStream is, int розмірБуфера**);**// **8192 байта** по дефолту

Приклад: String tmp = “Hello”;

byte[] byteArr = tmp.getBytes();

ByteArrayInputStream bAIS = new ByteArrayInputStream(byteArr);

BufferedInputStream buff = new BufferedInputStream(bAIS);

Клас **PushbackInputStream** (extends FilterInputStream) –механізм, який читає байт(байти) з вхідного потоку і “повертає їх на місце” (тобто не витягуючи дані з потоку). Клас робить неможливим виклик методів reset() і mark(). Конструктори:

PushbackInputStream(InputStream is);

PushbackInputStream(InputStream is, int кількБайт);

Методи:

void **unread**(int b) - вштовхує в потік молодший байт b; void **unread**(byte[] буфер)–вштовхує байти в буфер; void **unread**(byte[] буфер, int of, int len) – вштовхує кількість байтів len в буфер з позиції of.

Клас **SequenceInputStream** (extends InputStream) – дозволяє об’єднати разом декілька об’єктів класа InputStream. Конструктори:

SequenceInputStream(InputStream перший, InputStream другий);

SequenceInputStream(Enumeration<? ExtendsInputStream>перелікПотоків);

Під час роботи клас виконує запити на читання спочатку першого файлу до кінця, потім другого. Після закінчення читання з файлу пов’язаний з ним потік закривається. Закриття об’єкта SequenceInputStream призводить до закриття всіх потоків які в ньому знаходились.

Клас **DataInputStream**(extends FilterInputStream, implements DataInput) **-** використовують для читання даних формату Джава. (примітивних типів + String). Реалізує інтерфейс DataInput методи якого перетворюють елементарні дані в форму послідовності байт.

**public class** Template {  
 **public static void** main(String[] args) {  
 *//write data* **try** (DataOutputStream outData = **new** DataOutputStream(**new** FileOutputStream(**"C://data.dat"**))) {  
 outData.writeBoolean(**true**);  
 outData.writeInt(12);  
 outData.writeChar(**'x'**);  
 } **catch** (FileNotFoundException e) {  
 System.***out***.println(**"Can't open the file."**);  
 **return**;  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
  
 *//read data* **try** (DataInputStream inputData = **new** DataInputStream(**new** FileInputStream(**"C://data.dat"**))) {  
 **boolean** b = inputData.readBoolean();  
 **int** i = inputData.readInt();  
 **char** c = inputData.readChar();  
 System.***out***.println(b + **", "** + i + **", "** + c);  
 } **catch** (FileNotFoundException e) {  
 System.***out***.println(**"Can't open the file."**);  
 **return**;  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
   
 }  
}

**abstract OutputStream** – (реалізує AutoClosable, Closable, Flushable) - вихідний потік запису.

OutputStream outStream = new ByteArrayOutputStream();

Має 3 метода **abstract write()** симетричні до 3-х методів read() в InputStream. ( void write(int b), void write(byte[] b), void write(byte[] b, int off, int len) ). І має 2 метода закінчення (**close()**– закриває весь ланцюжок запису. (приклад: JVM->OS->HDD)). **flush()**, - записує дані, скільки б їх не було, оскільки дані методом write передаються в JVM, OS, HDD, а OS i HDD не одразу записують дані з OutputStream тому, що мають свої кеші з розмірами якими їм зручніше працювати. Тобто дані з OutputStream ніби записуються в буфери OS i HDD і не завжди заповняють повністю ці буфери тим самим записуючи дані з кеша на носій. Бажано завжди перед викликом метода close() викликати метод flush() (який викликає ланцюг flush у JVM, flush у OS, flush у HDD) бо, до прикладу, якщо в кешах будуть накопичуватися дані i відключити HDD то дані на нього не встигнуться записати оскільки буфер кешу не буде заповнений і не відбудеться автоматичний запис даних.

closeQuietly() - закриває потік. На відміну від close() не викидає ніяких ексепшенів. (catch IOException)

closeAndFlushQuietly() - робить flush(), потім close() не викидаючи ексепшенів. (catch IOException)

Клас **FileOutputStream** (extends OutputStream) - створює об’єкт класа OutputStream. Використовується для запису байтів в файл. Створення об’єкта не залежить від того чи є файл в системі, FileOutputStream сам створює його перед відкриттям. Конструктори:

FileOutputStream(String шляхДоФайлу);

FileOutputStream(File об’єктФайла);

Приклад: FileOutputStream fOut = new FileOutputStream(String шляхДоФайлу);

Копіювання файла. Приклад:

Клас **ByteArrayOutputStream** (extends OutputStream) – реалізує вихідний потік, який виводить масив байтів. Щоб створити масив з байтів потрібно вказати його розмір, а розмір масиву байтів з вихідного потоку дізнатися неможливо. Тому використовується ByteArrayOutputStream який бере на себе відповідальність за визначення розміру масива. Розмір буфера збільшується автоматично в залежності від потреб. Відбувається це на рівні класу шляхом створення нового масиву більшого розміру в який копіюються старі дані. Метод close() викликати не обов’язково. ByteArrayOutputStream є "адаптром" між OutputStream i масивом байтів byte[] arr. Конструктори:

ByteArrayOutputStream(); // створюється буфер **в 32 байта**

ByteArrayOutputStream(int кількБайтів);// вказуємо розмір явно

Клас **FilterOutputStream**(extends OutputStream) - сам абстрактний клас не використовується в додатках Джава, більш корисні методи є в класах, які його розширяють. Переоприділяє методи OutputStream. Конструктор:

FilterOutputStream(OutputStream os**);**

Клас **BufferedOutputStream**(extends FilterOutputStream)- буфер, який дозволяє писати більше ніж один байт за раз. Конструктор:

BufferedOutputStream(OutputStream os**);**

BufferedOutputStream(OutputStream os, int кількБайт**);**

Клас **PrintStream**(extends FilterOutputStream, implements Appendable, Closeable, Flushable)- представляє собою всі виводи, якими ми користуємося з дескриптором System.out. Конструктори:

PrintStream**(**OutputStream os**);**

PrintStream**(**OutputStream os, boolean скидБуфПриНовСтроці**);**

PrintStream**(**OutputStream os, boolean скидБуфПриНовСтроці, String кодування); // throw UnsuportedEncodingException

PrintStream(File вихіднийФайл); // throw FNFE. Файл створюється автоматично.

PrintStream(File вихіднийФайл, String кодування); // throw UEE

Методи : **print**(); **println**();

**printf**() – дозволяє задати точний формат виводу даних (використовує клас **Formatter**). Форми:

PrintStream printf(String формСтрока, Object…аргументи);

PrintStream printf(Locale регіон, String формСтрока, Object…аргументи);

**public class** Template {  
 **public static void** main(String[] args) {  
 System.***out***.println(**"Digit values with formatter."**);  
 System.***out***.printf(**"%d %(d %+d %05d\n"**, 3, -3, 3, 3);  
  
 System.***out***.printf(**"Digit default format: %f\n"**, 1234567.123);  
 System.***out***.printf(**"Digit format with comma: %,f\n"**, 1234567.123);  
 System.***out***.printf(**"Negative difit format: %f\n"**, -1234567.123);  
 System.***out***.printf(**"Another negative gigit format: %,(f\n"**, -1234567.123);  
 }  
}

//Digit values with formatter.

3 (3) +3 00003

Digit default format: 1234567,123000

Digit format with comma: 1 234 567,123000

Negative difit format: -1234567,123000

Another negative gigit format: (1 234 567,123000)

Клас **DataOutputStream** (extends FilterOutputStream, implements DataOutput) - використовують для запису даних формату Джава. (примітивних типів + String). Реалізує інтерфейс DataOutput методи якого перетворюють елементарні дані в форму послідовності байт.

**Читання консольного вводу** (з клавіатури) :

**BufferedReader** buf = newBufferedReader(new InputStreamReader(System.in)).

Після цього об’єкт buf представляє собою символьний потік який зв’язаний з консоллю через потік System.in. **buf.read()** – зчитує посимвольно, **bufReadLine()** –зчитує строку. Також рекомендують обгортати буфером (BufferedInputStream) обєкти FileInputStream і FileOutputStream оскільки зчитування і запис виконуються швидше. Це обумовлено тим, що для побайтного зчитування і запису потрібна велика кількість запитів на прийом-передачу даних, а використовуючи буфер кількість запитів зменшується, що суттєво збільшує швидкість зчитування і запису. (Наприклад, у нас є метод зчитування, який звертається до буфера, а буфер в свою чергу зчитує байти з файлу(FileInputStream). Метод, звертається до буфера і просить зчитати 1 байт, буфер в свою чергу звертається до файлу і закачує в себе кількість байт, яка рівна його буферу який вказали в конструкторі. Наприклад, його розмір встановили 8 Кб. Тоді в буфер заганяється цих 8Кб і один байт з буфера передається методу, який хотів зчитати один байт. Наступного разу, коли метод захоче зчитати байт, вже не буде відбуватися ланцюжок від метода до файлу, а потрібний байт буде швидко передано методу з буфера. Тобто збільшується швидкість передачі даних за рахунок зменшення кількості запитів від метода до файлу.)

BufferedReader використовує паттерн "Декоратор".

**Шаблон Декоратор** - це клас, який бере один тип, виконує з ним якісь дії і повертає його таким же типом. Головна його властивість не міняти функціональності і добавляти нових властивостей.

**Шаблон Адаптер** - це клас, який бере один тип, виконує з ним якісь дії і приводить його до до іншого типу.

**Запис в консоль** вконується за допомогою класу **PrintWriter**. Це один з символьних класів. Рекомендують використовувати саме його при виводі на консоль, а не System.out.println(), тому-що PrintWriter забезпечує можливість інтернаціоналізації для програм.

PrintWriter prWrtr = new PrintWriter(System.out, true); – об’єкт класу PrintWriter, який підключений до консольного виводу. Тут System.out – потік виводу, true – автоматичне скидання буфера після кожної строки. Приклад: prWrtr.println(“ ”);

**InputStream**/**OutputStream** орієнтовані на читання\запис байтів.(byte[])

**Reader**/**Writer** орієнтовані на читання\запис символів.(char[]). Потрібно враховувати кодування (charset). Аналогічні по будові і принципу до ІС і ОС тільки ті працюють з байтами.

**Reader** implements AutoCloseable, Closeable, Readable. (close(), mark(), read(), ready() )

**FileReader** extends InputStreamReader (який extends Reader). Використовується для читання вмісту файлу.

**Writer** implements AutoCloseable, Closeable, Flushable. (append(), close(), write(), flush() )

**FileWriter** exteds OutputStreamWriter (який extends Writer). Використовується для запису даних в файл.

**Кодування** (charset) - співставлення набору байтів до його символьного представлення.

**InputStreamReader**- клас адаптер, який працюючи з типом Reader приводить його до типу InputStream. (InputStream/ InputStreamReader/Reader). Перетворює байти в символи.

**OutputStreamWriter** - клас адаптер, який працюючи з типом Writer приводить його до типу OutputStream. (OutputStream / OutputStreamWriter/Writer).

**Клас File** (пакет java.io.file) – клас для роботи з файлами. Не працює з потоками на відміну від інших класів ІО. Має справу безпосередньо з файлом і файловою системою. Описує властивості файлів. Шлях, час, права доступу, тощо. Був введений в ранніх версіях мови. Приклад:

File someFile = new File(“c:/tmp/image.png”);

В силу поганого дизайну даного API є вірогідність створити сутність, яка є і не файлом, і не директорією.

File someFile = new File(“abc,,,”);

**someFile.isFile()** – перевіряє чи створена сутність є файлом, повертає true/false, також повертає false для деяких спеціальних файлів (драйвера, тощо).

**someFile.isDirectory()** - перевіряє чи створена сутність є директорією, повертає true/false;

**someFile.listFiles** – повертає масив типу File з переліком файлів.

**Interface FileFilter** – має один метод **boolean accept(),** використовується для фільтрування файлів. Приклад:

SOP(Arrays.toString(someFile.listFiles(new MyFilter() ))).

В даному випадку class MyFilter implements FileFilter в якому перевизначено метод public boolean **accept(Filepath)** { returnpath.isFile(); }. Якщо фільтр викликається одноразово, то не рекомендується заводити для нього цілий клас MyFilter implements FileFilter, а краще використати анонімний іннер класу - перевизначити методо accept() в класі в якому ви працюєте.

**boolean exists()** – перевіряє чи існує файл.

**someFile.createNewFile()** – створює новий файл (повертає boolean),

**someFile.delete()** – видалає файл (повертає boolean),

**someFile.mkdir()** – створює нову директорію (повертає boolean).

**getParent()** –повертає ім.’я родительского каталогу.

**canWrite(), canRead( )** –доступний для запису\читання?

**lastModified()** – час останньої зміни файлу (millis).

**renameTo(String новеІм’я)** – перейменовує файл.

При чому API влаштовано так, що ми не можемо видалити папку, якщо в ній є файли – потрібно заходити в папку, видаляти файли і рекурсивно видаляти папки, які є в середині.

Ми не можемо створити екземпляр інтерфейса, тому його некоректно називати ссилочним типом. Проте ми можемо сворювати анонімні іннери класу\інтерфейсу, як на манер до new Runnable в кострукторі Thread і перевизначати методи інтерфейсу.

SOP(Arrays.toString(someFile.listFiles(**newFileFilter(){**

public boolean accept(File f); …// filter anonymus class inner

**}** ))

Ми можемо за допомогою File створювати файли, видаляти. Але не можемо за допомогою File писати в них чи читати з них. Потрібно застосовувати FIS і FOS.

**Interface FilenameFilter** – повертає список імен які визначені певним шаблоном або фільтром. Для цього використовується друга форма метода list().Приклад:

String[] list(FilenameFilter об’єкт);

Тут об’єкт реалізує інтерфейс FilenameFilter який визначає один метод accept(). **boolean** **accept(File каталог, String ім’яФайлу)** – метод повертає truе для файлів з каталогу, які відповідають параметру ім’яфайлу і повертає значення false для фйлів які потрібно з списку виключити.

Каталог – об’єкт класа File. Зберігає в собі список файлів і других каталогів.

boolean **mkdir**() – (метод класа File). Створює каталог по визначеному шляху.

boolean **mkdirs**() – (метод класа File). Створює каталог (включаючи каталоги, яких немає) по заданому шляху.

За допомогою FileInputStream зчитується потік байтів з файлу.

За допомогою FileOutputStream записується потік байтів в файл.

Коли вихідний потік відкриває файл, то файл який існував раніше видаляється.

Рекомендують закривати вхідні і вихідні потоки в блоці finally{}, а роботу з вхідними і вихідними потоками робити в try{} обробляючи виключення в catch(IOException...){}. Закриття потоку в finally{} відбувається шляхом перевірки його на null оскільки в блоці rty{} потік може кинути виключення (не знайшовши файл, наприклад), тоді методу close() не буде чого закривати. Приклад:

finally {

if (fileInput != null) {

try{

fileInput.close();

} catch (IOException){ //NOP }

}

}

Проте, найдоцільніше буде використовувати try з ресурсами. (з Джава 7)

Але, використовуючи FileInputStream (чи з іншими засобами які оперують даними за межами heap–sockets etc...), потрібно викликати метод close() аби завершити ланцюг викликів. (приклад JVM ->OS ->HDD)

**try з ресурсами** можуть використовувати тільки ті класи, які реалізують інтерфейс **AutoCloseable**. (пакет java.lang) Цей інтерфейс оприділяє метод close() який неявно викликається після завершення блока try. Область видимості створеного ресурса обмежується блоком оператора try. Інтерфейс Closeable (що реалізований в пакеті java.io.) наслідується від інтерфейса AutoClosable. Обидва інтерфейси ініціюються потоковими класами. Таким чином, try з ресурсами може використовуватися при роботі з потоками включаючи файлові потоки.У випадку, коли в try викинуто виключення (наприклад, файл не найдено), то в finally, коли ми закриватимемо цей незнайдений файл вилетить інше виключення і перше з try буде подавлено. У випадку з **try з ресурсами** перше виключення не подавляється, а подавляється друге в finally.Проте друге не губиться, а добавляється в список подавлених виключень пов’язаних з першим виключенням (метод getSuppressed() повертає список подавлених виключень).

В циклі **for-each** можна працювати лише з масивами і класами, які реалізують **інтерфейс Iterable.**(List, Set...)

Ми можемо отримати “A.equals(B) - true” i “A == B - false” між двома об’єктами А і В у випадку серіалізації. Тому-що після «клонування» в heap це різні об’єкти, тому “A == B - false”, але в них одинакові поля, тому “A.equals(B) - true”. (це при deep copy).

**DeepCopy** – коли повністю копіюється клас, його поля, поля його полів і т.д. (Якщо один об’єкт має 1 Мб, то копія об’єкту і його полів теж матиме 1 Мб)

**Shallow Copy** – коли копіюється клас, але його поля і поля його полів тощо… зсилаються на поля в пам’яті першого об’єкта. Тобто повного копіювання не відбувається, а в heap створюються ссилки на поля класу якого ми хочемо копіювати. Його проблема полягає в тому, що якщо створити декілька об’єктів і ними робити зміни в полях на які вони посилаються, то зміни будуть видні для всіх об’єктів оскільки це одна сутність до якої просто ведуть посилання в heap.

При поверхневому копіюванні є можливість того, що ми з копії з ссилки першого рівня перейдемо в ссилку другого рівня, а при поверненні назад на перший рівень попадемо не в перший рівень копії, а в перший рівень оригінала.

Клас Object має метод **clone**() який реалізовує поверхневе копіювання. Але об’єкт до якого застосовується clone() має **implements** **Cloneable**.

Коли ми працюємо з БД і нам потрібно вивести тільки деякі поля, то поля які нам не потрібні «обрізаються» - заповняються nullами.

В Джава найбільш популярними моделями є **об’єктна** і **реляційна**.

**Об’єктна модель** - модель з сукупністю принципів які лежать в основі ООП, яке основе на абстрагуванню, інкапсуляції, модульності, ієрархічності, типізації, паралелізму і персистентності.

**Реляційна** **модель** - модель яка представляє собою фіксовану структуру математичних понять, які описують те, як буду представлені данні. Базовою одиницею даних в реляціній моделі і в БД є таблиця.

**Серіалізація** – це процес який перетворює по одній вершині довільний граф об’єкта в масив байтів і може передати його в вихідний потік щоб зберегти в БД чи файлі. Для того, аби об’єкт можна було серіалізувати, потрібно аби він **implements** **Serializable** інакшеотримаємовиключення **NoSerializableException**.

При серіалізації записуються всі транзитивні замикання. І, наприклад, є поля, які ми не хочемо серіалізувати. (наприклад пароль, який можуть декомпілювати) Тоді використовується – **transient** (засіб, який ігнорує серіалізацію і не дає копіювати поле. Його значення буде null.). Записується як модифікатор доступу, наприклад, public. Також **не зберігаються при серіалізації статичні поля**.

При **глибокому** **копіюванні** використовується термін "**транзитвного** **замикання**".

При переході від моделі ООП до моделі Реляціної можливе тільки глибоке копіювання оскільки ми не можемо зсилатися з однієї моделі на об’єкт в іншій моделі (реалізовувати поверхневе копіювання) в силу різності їх організацій.

**інтерфейси ObjectInput/ObjectOutput** **( extends DataInput/Dataoutput, implements AutoCloseable, Flushable** ) - працює з Object.( **readObject**(obj), **writeObject** (obj). Може читати і записувати довільні об’єкти. Тобто дозволяє працювати з ссилочним типом на відміну від DataInput/DataOutput. (**ObjectOutput** ->**ObjectOutputStream**(адаптер) -> **OutputStream**).

В Джава можна довільний об’єкт перетворити в потік байтів і навпаки. Приклад: ObjectOutput(obj) -> ObjectOutputStream -> ByteArraOutputStream -> byte[]. Навпаки: ObjectInput(obj)<- ObjectInputStream <- ByteArraInputStream <- byte[].

new Object неможливо серіалізувати (вилітає ексепшн), а масив об’єктів можна. (Objectob = newObject[0])

Масив об’єктів являється об’єктом. Object[] obj = newObjct[1]; (це те ж сааме, що й Object[] obj = { newObject } ).

**Клас ObjectInputStream extends InputStream implements ObjectInput** - відповідає за читання об’єктів з потоку. Методи: **availiable**(), **close**(), **read**(), **readBoolean()**, **readByte()**, **readInt()** etc... **final** **void** **Object readObject**()...

ObjectInputStream(InputStream inStream) throws IOE.

**Клас ObjectOutputStream extends OutputStream implements ObjectOutput** - підтримує серіалізацію об’єктів. Для серіалізації об’єкта викликає метод **writeObject**(). Методи: **close**(), **flush**(), **write**(), **writeByte**(), **writeInt**() etc... **final** **void** **writeObject**().

**Маркед інтерфейс** – інтерфейс без методів. Служить для маркування. Приклад [Serializable](https://www.google.com/search?q=%D0%B8%D0%BD%D1%82%D0%B5%D1%80%D1%84%D0%B5%D0%B9%D1%81+Serializable&spell=1&sa=X&ei=k3xGU4C0B8zS4QSnoYGQCQ&ved=0CCgQvwUoAA).

Serializable, Cloneable - marked інтерфейси.

------------------------------------------------------------------------------------------------------------------------

Кожен клас може мати тільки одного предка. (неявний java.lang.Object). Тобто в Джава немає множинного наслідування і конструкція class X extends Y, Z, F неможлива. При чому один з класів неявно буде наслідником java.lang.Object. (class X extends Y. Де Y буде неявним наслідником java.lang.Object).

Якщо за допомогою рефлексії запитати хто предок Object – отримаємо null:

SOP(Object.class.getSuperclass() ); //null

В одному файлі не може бути 2 публічних класа:

public class X0 {}

public class X1 {} // error

В Джава не може бути циклічної залежності предків. Тобто наслідник не може бути предком предка:

public class X0 extends X1{}

public class X1 extends X0 {} // error

Кожен клас має мати хоча б один конструктор. Якщо ж ми не напишемо конструктор явно, то компілятор неявно створить пустий конструктор класу.

Якщо ми класу створюємо якийсь конструктор, то обов’язково треба вручну створити конструкто по замовчуванню, оскільки компілятор це не зробить автоматично в даному випадку. (йому головне дотримуватися правила, що має бути хоча б один конструктор) Приклад:

private class C{

// private C() {}

privete C(int x) {} // компілятор пропустить, але коли ми захочемо //зробити new C() – компліятор підкреслить строку оскільки такий //конструктор не буде знайдено ( private C() )

}

Інтерфейси слугують для того аби «документувати обіцянки». Тобто, імплементуючи інтерфейс ми обіцяємо, що використаємо і перевизначимо його методи.

У інтерфейсів не може бути полів. Ні статичних, ні фінальних. Також у них не може бути реалізації методів.

Наприклад, ми маємо клас С, який наслійдується від класа В, а той наслідується від класа А (A <- B <- C). То для того аби зробити екземпляр класа С, компілятор створює внутрішній екземпляр класа А, потом внутрішній екземпляр класа В, а потом вже клас С. При чому ВМ викличе по порядку конструктор класу А, потом В, потім С. Клас С, в такому випадку буде мати всі поля, які є в класах А і В.

Важливо!!! Від предків наслідуються методи, але не наслідуються конструкори.

public class A{

public A(int x){}

x();

y();

}

class B extends A{

z();

f();

new B.x(); //метод предка

new B.y(); // метод предка

new A(7);

new B(7); // помилка, бо В немає такого конструктора, а від А не наслідує

}

this (ссилка на інстанс того об’єкта в якому ми зараз знаходимся) не можна використовувати в статичних методах.

Через this можна викликати інші конструктори класу. Приклад:

public class Recruit){

int age;

public Recruit(int x) { this.age = x; SOP(age); }

public Recruit() { this(18); SOP(age); } // викликали конструктор Recruit(int x) чрез this

}

При чому виклик іншого конструктора має бути на першому місці:

public Recruit() {

int y = 8; // compile error

this(18);

SOP(age);

}

Ще один приклад це клас ArrayList<> який в конструкторі по замовчуванніпросто ссилається на інший конструктор ,який вказує величину ліста і по дефолту встановлює її на 10:

new ArrayList<>(); //створюємо екземпляр

-------------------------------

public ArrayList(){ this(10) } // пустий к-р, який викликає інший к-р

-------------------------------

public ArrayList(int capacity){ …this.elementData = new Object[capacity] }

super повертає ссилку на екземпляр предка. Можна викликати поле чи метод предка через крапку: super.init(); Або викликати конструктор предка: super(18);

Заповнення масиву цілих чисел нулями відбувається для того, або ми не мали доступу до старих байтів в сегменті пам’яті, який був виділений під масив. Наприклад, там можуть бути байти якими оперувала ОС до цього.

Є різниця між ініціалізацією полів і змінних. Неініційоване в ручну поле можна використовувати, а неініційовану змінну – ні, компілятор видасть попередження про те, що вона неініційована.

public class A{

private int i;

public A(){

int x;

SOP(i); // видасть 0

SOP(x); // помилка компілятора, що х неініційована.

}

}

Тобто поля, які ініціюються до виклику конструктора – забиваються дефолтними значеннями, змінні записані після виклику конструктора – не заповняються дефолтними значиннями. При чому, поля, які вказані до конструктора ініціюються дефолтними значеннями автоматично ще до виклику конструктора. Тобто, при створенні екземпляра класа, якщо якесь його поле до конструктора буде ініційоване методом, то спочатку виконається цей метод, а потім вже виконуватиметься конструктор (але так робити неприйнято).

--------------------------------------------------------------------------------------------------------------------------

Щоб отримати немодифіковану колекцію (доступну тільки для читання) використовують методи: Colections.unmodifiabeCollection(), Colections.unmodifiabeList(), Colections.unmodifiabeSet(),Colections.unmodifiabeMap().

Майже всі колекції не синхронізовані. Крім застарівших Vector i HashTable.Щоб отримати синхронізовану колекції з несинхронізованої використовують статичні методи synchronizedCollection(),synchronizedMap(),synchronizedList() etc.

Різниця між Iterator і Enumeration: енумерейшн в 2 раза швидші і займають менше пам’яті, але ітератор є безпечнішим для багатопоточності.

ListIterator наслідується від terator. ListIterator використовується тільки для передобру List , а Ітератор добре працює зколекціями Set. ЛістІтератор може перебирати колекцію в двох напрямках (дляпереборувикористовують interator або цикл foreach).

fail-safe ітератори на відміну від fail-fast не викликають ніяких виключень при зміні структури тому-що працюють з клоном колекції замість оригінала.

ConcurrentModificationException може виникнути під час модифікації колекції яка в даний момент ітерується.

Різниця між Comparable i Comparator: Comparable задає природній порядок порівняння об’єктів і має тільки один метод compareTo() який порівнює вхідний об’єкт з викликаючим, а Comparator задає додатковий порядок порівнянняоб’єктів, має метод compare(), який порівнюэ 2 вхідних параметри.

в Integerpool попадают значенияот -128 до 127. Клас ItegerCash має в собі масив всіх інтів від -128 до 127, і коли ми створюємо цілочисельну змінну, відбувається пошук такої ж рівної змінної в цьому масиві. Якщо така зміннв там є, то не створюється новий об’єкт, а нашій змінній присвоюється хеш код на такй самий елемент в цім масиві. Зроблено це за для того, щоби не наплоджувати нових об’єктів з одинаковим значенням.

Integer a = 1000;   
Integer b = 1000;   
Integer c = -100;   
Integer d = -100;   
System.out.println(a == b);  //flse  
System.out.println(c == d);  //true

Дане присвоєння (автобоксинг) Integer a = 1000;  відбувається неявно як Integer a =  Integer.valueOf(1000);.  Метод valueOf() класа ItegerCash порівнює параметр який йому дають і якщо він вписується в межі -128 до 127, то відбувається присвоєння з масиву хеш коду змінній, яка була створена. Зроблено аби зекономити ресурси, оскільки для інта цей інтервал найчастіший.

Масив в Джава однорідний (однотипний). Але якщо створити масив через клас, то можно в ячейки добавляти не тільки екземпляри даного класу, а й його наслідників. Це зумовлено поліморфізмом.

**Персистентність** - стосовно коду - властивість, наприклад, модуля, зберігати свою працездатність і функціонал не залежно від змін, які відбуваються довкола нього з іншими модулями. Стосовно БД і файлів - це можливість зберігати свій стан і інформацію, навіть якщо, наприклад, буде виключено сервер і все з ОЗУ пропаде. Самі ж файли і БД будуть збережені на жорсткому диску. Тобто це властивість зберігати свій стан довше ніж процес, який тебе породив.

------------------------------------------------------------------------------------------------------------------------------------

Разница между интерфейсом и абстрактным классом.

Интерфейс — это договорные отношения с классами, которые этот интерфейс реализуют, о том, что реализация происходит путём, обозначенным интерфейсом. Это пустая оболочка с объявленными методами.

Абстрактный класс определяет некоторое общее поведение и просит свои подклассы определить нетипичное или конкретное поведение для своего класса.

Методы и члены абстрактного класса могут быть обозначены любым модификатором доступа, в свою очередь все методы интерфейса обязаны быть открытыми (public).

Когда происходит наследование абстрактного класса, класс-наследник должен определить абстрактные методы, в то время как интерфейс может наследовать другой интерфейс и при этом не обязательно определять его методы.

Класс-наследник может расширять только один абстрактный класс, а интерфейс может расширять или класс может реализовывать множество других интерфейсов.

Класс-наследник может определять абстрактные методы с тем же или менее ограниченным модификатором доступа, при этом класс, реализующий интерфейс, должен определять методы с тем же уровнем видимости.

Интерфейс не содержит конструкторы, в том время, как они есть в абстрактном классе.

Переменные, объявленные в Java-интерфейсе по умолчанию являются static final. Абстрактный класс может содержать переменные, которые не являются final.

Все участники Java-интерфейса по умолчанию являются public. Участники абстрактного класса могут позволить себе быть public, protected и др

-------------------------------------------------------------------------------------------------------------------------------