**Типи даних** в Джава: **ссилочн**і(класи, масиви, енуми... Все з чого можна створити екземпляр) і **примітивні** (byte(8-bit), short(16-bit), int (32-bit), long( 64-bit, 1 000 000 000 000L), float(32-bit, 23,4F - потрібно добавляти F), double(64-bit), boolean (в специфікації не визначено, може відрізнятися в різних JVM), char(16-bit/32 bit з JDK5)).

Можлива приставка букв для чисел з плаваючою точкою:

**double** x = 5.54f; \\можлива бо автоматично флоат розшириться

**float** x = 5.54d; \\неможлива бо дабл не вміститься в флоат.

При діленні на 0 отримаємо NPE. Крім float(“Infinity”), double(“Infinity”). Чи NaN.

При діленні цілого числа на 0 мала б вилітати ArichmeticException.

**1. int** x = 10;  
 **int** y = 0;  
 **int** z = x / y;  
 System.***out***.println(**"z = "** + z); //excpetion

**2. int** a = 4;  
 **try** {  
 System.***out***.println(a/0);  
 } **catch** (ArithmeticException e) {  
 System.***out***.println(**"Devision by zero"**);  
 } //Devision by zero

Елемент ссилочного типу має тільки одну ссилку.

Для символа можливі цілочисельні операції. Наприклад: charc=22; c++;

**char** x =**'2'**;  
x++;  
System.***out***.println(x); \\3

Можлива ситуація, коли в хіпі є 2 об’єкта які ссилаються одне на одного. Тоді, коли з стека пропадає посилання на ці 2 об’єкта – ГК не видалить їх, оскільки вони мають ссилку одне на одного.

Області видимості змінних бувають визначені методом і класом.

x<<1; // множення на 2. (2 - 8, 3 -16 - далі працює як взведення в степінь)

**byte** x = 2;  
x = (**byte**) (x<<3);  
System.***out***.println(x); \\16

x>>1; // ділення на 2. (далі працює як пониження степені)

**byte** x = 4;  
x = (**byte**) (x>>1);  
System.***out***.println(x); \\2

При діленні цілого непарного числа на парне - отримаємо ціле число без дробової частини:

**int** x = 15/2;  
System.***out***.println(x); \\7

Що б компілятор міг розрізнити float і doubleдо першого ставиться приставка f:

double doubleVariable = 2.6;  
float floatVariable = 2.6f;

Константи:

static final int INT\_CONST = 100; // static – одне для всіх, final – незмінне.

При приведенні типів з дробних до цилочисельних – відкидається дробна частина без округлення.

З цилочисельного в дробний – добавляється нуль після коми.

Для приведення ссилочних типів використовується клас Number і його метод, наприклад, intValue.

Float floatVar = 2.45f;[*\\змінна*](file:///\\змінна) типу float

Integer intVar =15;[*\\змінна*](file:///\\змінна) типу int

Number numb = floatVar; \\ створюємо оболонку типу Number в яку поміщаємо тип, який хочемо привести до іншого.(у нас Float доInteger)

Integer floatVar = numb.intValue(); \\ використовуємо приведення типу з Float в Integer за допомогою метода intValue();

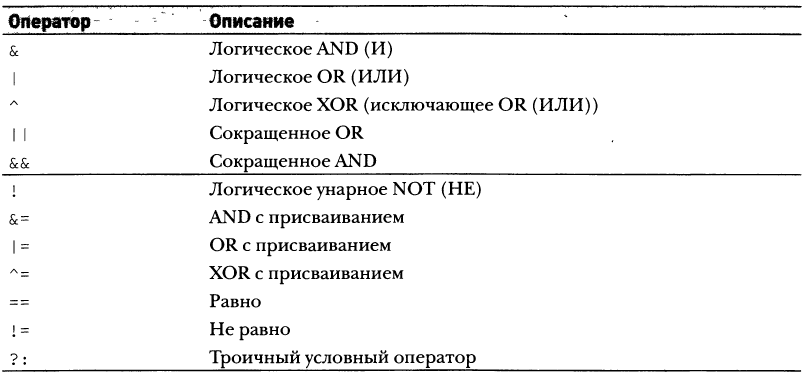
Або простіше:

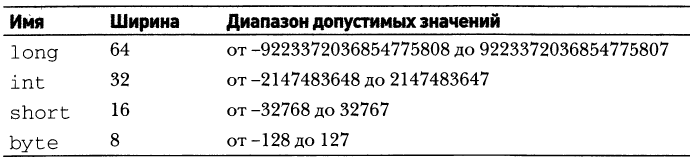
Float floatVar =45f;  
Integer intVar = floatVar.intValue();﻿

ВМ складається з GC, gitcompiler і Runtime(assembler\dissasembler, class loader, stack, thread managemant, reflection, bootstrap, exception, error report jni, і багато іншого.)

В різних реалізація ВМ може бути не один стек, а 2. Один в якому вираховуються вирази (expressions), а інший є стеком викликів.

Інлайнінг методів (inlining) – коли один метод викликає інший через точку.





Для того аби запустити джава програму з командної строки – потрібна перейти в розташування збереженого Example.java файлу і прописати **javac Example.java**. (Якщо при виконанні виникла помилка, потрібно системі вказати шлях до змінної середовища path, де оболочка ОС шукає виконавчі файли – пишемо в консолі **set PATH= C:\Program Files\Java\jdk1.8.0\_25\bin**) Після цього в папці появиться файл Example.class – це означає, що програма скомпілювалася в байт-код. Далі просто запускаємо **java Example.**

**Приведення типів** – коли ми маємо змінну одного типу і її значення хочемо присвоїти змінній іншого типу. Деякі типи між собою мають автоматиче приведення типів. Це відбувається тоді, коли точності змінної, яка приймає значення вистачає для того, щоб прийняти значення іншої (наприклад, коли short\byte зберігають в змінній int – це називається роширенням типу, або підвищенням типу оскільки тип з меншою розрядністю підвищується до типу з більшою розрядністю). Коли ж нам треба привести зворотнє присвоєння (наприклад int to byte), то автоматичного приведення типу не відбудеться. Нам потрібно вказати явно (byte) intVar – це називається звуженням типу. Якщо б при такій операції число int виходило за діапазон чисел byte, то результатом звуження був би остаток від ділення по модулю на допустимий діапазон типу byte.

Приклад:

int а = 100;  
 byteb = (byte) а;

Для проміжкових обчислень (наприклад byte \* byte) підвищення типу всіх частин виразу відбувається автоматично до int оскільки byte \* byte може вийти за межі діапазону самого byte.

byte а = 40;   
 byte b = 50;   
 byte с = 100;   
 int d = a\* b / с; // a, b, c будуть розширені до int

Іноді при автоматичному підвищенні може виникнути помилка навіть при збереженні діапазону типу. Наприклад:

byte b = 50;  
 b = b\* 2:   
 ^Incompatible type for =. Explicit cast needed to convert int to byte.  
 (Несовместимый тип для =. Необходимо явное преобразованиеint вbyte)

Исправленный код:   
 byteb = 50;  
 b = (byte) (b\* 2); // тобто потрібне явне приведення типу, якщо змінна b не є типом int

Якщо в виразах використовуються змінні типу byte, short, int, то для запобіганню переповнення всі члени виразу підвищуються до int. Якщо ж в виразі є хоча б один long, то всі члени виразу підвищуються до long. Але потрібно пам’ятати, що всі цифри в кінці яких не стоїть L є int, а не long (1 000 000 000L). Якщо ж в виразі є хоча б один float, то всі члени виразу підвищуються до float. Якщо ж в виразі є хоча б один double, то всі члени виразу підвищуються до double. По замовчуванню ВМ всі числа з точкою розглядає як double.

**Автоматичне присвоєння значення при ініціалізації:**

byte, short, int = 0

long = 0L

float = 0.0f

double = 0.0d

char='\u0000';

boolean = false

String = null

Інтервал char - U+0000 - U+FFFF.

З JDK 5 клас Character почав підтримку 32 бітного представлення символів Юнікод. (раніше було 16 бітне представлення, що є рівним розміру типу char). Раніше символи лежали в діапазоні від 0 до FFFF, а в 32 представленні від 0 до 10FFFF. Це зумовлює проблеми, оскільки 32 бітний елемент не поміщається в 16 бітний char. Вирішується ця проблема або використанням двох змінних char, одна з яких називається старшою сигнатурою, а інша молодшою сигнатурою. Вводиться поняття кодової точки. Або використовуються перегружені методи Character які замість char приймають int.

float погано підходить для змінних, які треба порівнювати за рахунок точності запису. Наприклад, ми записали в 2 змінні значення 1, а зберігтися вони могл по різному: 1,000000000000000010000 і 1,0000000000000000000001111111. Ціни, наприклад, не варто записувати з плаваючою точкою. Варто заисувати їх в цілочисельному вигляді. В банківській справі взагалі не викоритовуються операцій ділення грошуй - використовується зсув.

**Таблиця істинності** - таблиця, яка описує логічну функцію і результат взаємодії логічних змінних в ній. (функції: and, or, xor, not)

**Унарні операції в Джава**: +, -, ++, --. Приклад: j= -i; // j = -3, тобто поміняно знак на протилежний.

**Бінарні операції**: +, - (не путати + і - з унарними які просто міняють знак), /, \*, %.

**Строкові класи в Джава**: String (створює незмінні строки), StringBuffer (створює змінну строку. Добавлений в першій Джаві. Безпечний при багатопоточності), StringBuilder (створює змінні строки, більш новий. Джава 1.5. Ефективнійший при однопоточності бо не має синхронізації).

Метод **main** також може кидати виключення. Його оброблятиме jvm.

Від класів **String**, **StringBuffer**, **StringBuilder** неможливо унаслідуватися, оскільки вони оголошені, як **final**.

Порівняння строк String і StringBuffer можна виконати шляхом виклику у об’єкта StringBuffer метода toString, а потім вже порівняння двох строк методом equals().

**Конкатенація** **строк** – це об’єднаня строк. Виконується за допомогою оператор +.

Строку в число можне перетворити за допомогою метода valueOf() який є у кожного класу обгортки.

Для порівняння двух строк використовується метод equals(), який виконує посимвольне порівняння строк. Оператор == в цьому випадку не підійде, оскільки він порівнює ссилки на об’єкти.

Заміна частини строки – метод **replace**(). Заміна символа в строці – **setCharAt**(). Отримати частину строки – **substring**(), розбивання строки на частини – **split**(), перетворення змінної в строку – **toString()**, перевіряє чи є символ\послідовність символів в строці – **contains**(),

**Пул строк (множина строк)** – це область пам’яті Джава в хіпі, в якому зберігається фізичний вміст строк. Використовується для оптимізації збереження строк шляхом використання підходу Flyweigth (приспособленец), який при створенні декількох одинакових строк зберігає в пулі тільки одну і використовує її там, де мають бути такі самі строки (перекидає ссилки на себе). Саме тому строки є незмінними, оскільки зміна одного об’єкта міняла би всі об’єкти звязані з пулом строки.

String str1 = "abc";

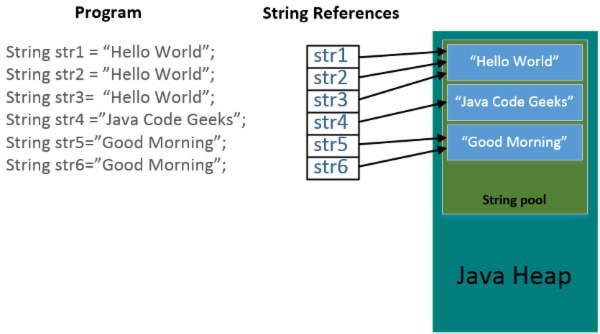
String str2 = "abc";

System.out.println(str1 == str2); //true

System.out.println(str1 == "abc"); //true

Тобто ми бачимо, що str1і str2 ссилаються на один і той же об’єкт. Це виглядає так:

Метод public String intern() - добавляє строку в пул строк, якщо її там нема і повертає строку в її канонічному представленні з пула строк щоб можна було використовувати "==", а не equals(). (зрівняння ссилок відбувається швидше ніж посимвольне порівняння строк.)



До Джава 7 пул строк знаходився в PermGen.

Інший варіант:

String str1 = "abc";

String str2 = "abc";

String str3 = new String("abc");

System.out.println(str1 == str2);//true, строки ссилаються на один строку в пулі

System.out.println(str1 == str3);//false, бо ми в ручну створили новий об’єкт в пам’яті ВМ

Ще варіант:

String str1 = "abc";

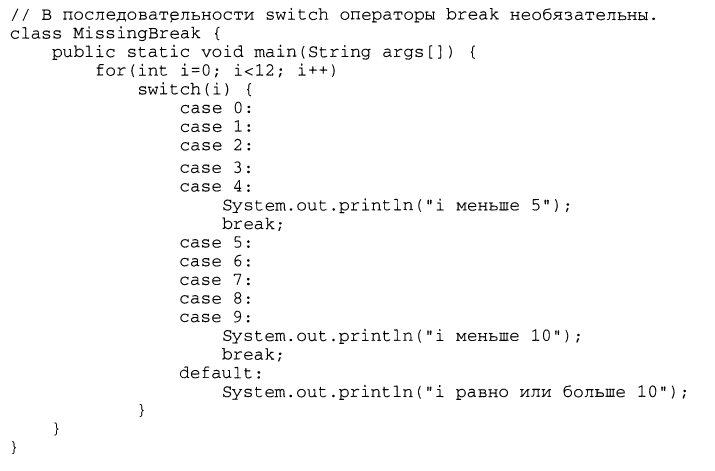
String str2 = "ab";

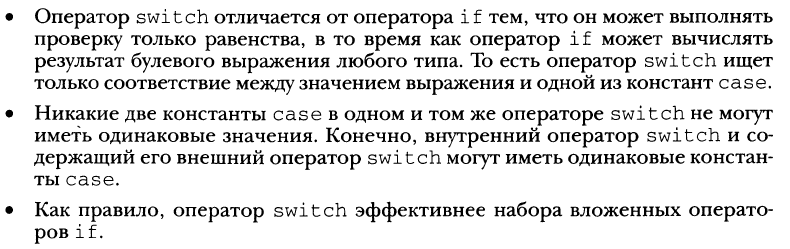
str2 = str2+"c";

System.out.println(str1 == str2);//false, юо строка є незмінною і при конкатенації створюється новий об’єкт. Тому це різні об’єкти.

Для ручного поміщення строки в пул використовується метод intern():

String s1 = "abc";   
String s2= "abc";   
  
String s3 = new String("abc");   
  
System.out.println(s1 == s2);  //true  
System.out.println(s1 == s3);  //false  
  
s3 = s3.intern(); // ручне введення строки в пул  
System.out.println(s1==s3); // true

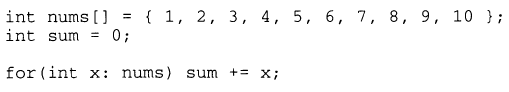
Приклад використання switch. Такожможливі вкладені switch.



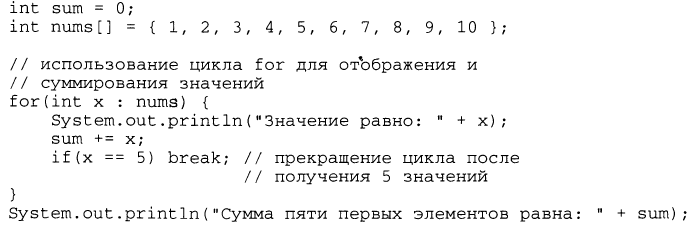
Оператори цикла: for, while, do-while, for-each.

for( ; ; ) – безкінечний цикл.

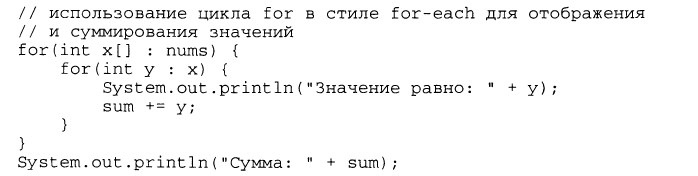
for-each: (на відміну від for не потрібна ручна індексація елементів і немає ризику виходу за межі масиву. Тільки читає дані з колекції і не міняє значення в ній)



Щоб перервати цикл при потрібному значенні – виикористовують break;



Для багатовимірних масивів: (int[] x – бо результатом буде ссилка на масив. Кожна ітерація цику for це нступний масив в масиві nums)



Оператири перехода: break (в switch, для переривання циклів і як форма оператора goto), continue, return. (обробка виключень також є механізмом переходу і виконання нелінійних операцій)

Оператор continue слугує для переходу виконання інструкцій в початок до ітеративної частини без виконання іншого коду цикла. (але не виходить з цикла на відміну від breake)

Оператор return виконує вихід з метода і передає керування методу, який викликав дану конструкцію. (код після return буде недосяжним unreachable, якщо немає певної логічної конструкції)

Метод println() спочатку друкує, а потім робить перенос строки:

public void println(String x) {  
synchronized (this) {  
 print(x);  
newLine();  
}  
}

Сатичну змінну неможна оголошувати в статичному мутоді:

**public static void** main(String[] args) {  
 **static char** i = **'\u1111'**; \\помилка  
 System.***out***.println(i);  
}

Якщо при приведенні типів змінну char вивести як тип int, то отримаємо юнікод сиволу.

**char** i = **'X'**;  
**int** y = (**int**) i;  
System.***out***.println(y); \\88

В конструкції switch(…) не потрібно забувати про case(…)… **break;**

& - Оператор логічного І. && - Оператор логічного І швидкої оцінки. При використанні && завжди обчислюється тільки ліва умова і якщо результат всього виразу ще не можливо передбачити, то обчислюється праве умова. Це дозволяє прискорити розрахунки. При використанні & завжди обчислюються обидві умови і права і ліва.

Енуми можуть мати свої методи. Від енумів неможливо наслідуватися.

Ми не можемо створити екземпляр інтерфейса, а тільки обєкт який наслідкє його, тому його некоректно називати ссилочним типом.

Ділення(\) реалізовано по різному в залежності від типу даних, які діляться (int, double… etc).

Строка в Джаві є незмінною. (immutable). Всі операції з нею приводять до створення нової строки.

String імплементують інтерфейси Serializable, Comparable, CharSequence.

Масив в джава фактично збільшити чи зменшити неможливо, оскільки на нього виділяється суцільний сегмент пам’яті. Всі дії по збільшенню чи зменшенню кількості елементів масива зводяться до створення нового масиву зміненої величини.

При пробіганні масиву з кінця в початок індекс k встановлюється рівним arr.lenght-1, оскільки індексація масивів починається з нуля.

Встроєні засоби виводу на консоль некоректно виводять в неї масиви – виводиться хеш код, а не самі елементи. Для коректного відображення масиву в консолі використовують Arrays.toString(массив).

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

Пакет **java.lang** імпортується автоматично.

Змінна System.in– об’єкт класу InputStream, а змінні System.out i System.err–об’єкти класу PrintWriter.

**Statement** – оператор. Все, після чого можна ставити «;». int x = 1;

**Expression** – вираз. Наприклад, double d = Math.sin(x); Boolean b = (int x == int y);

**System.arraycopy();** - нейтівний метод.

**System.arraycopy(**звідкиМасив, зЯкоїПозиції, кудиМасив, вЯкуПозицію, скількиЕлементів**);**

In-place алгоритми – алгоритми, які не використовують додаткову пам'ять для виконання своїх дій, а працюють з однією структурою даних в одному сегменті памяті.

Алгоритми сортування на співбесіді. (які, чим відрізняються)

Алгоритми сортування діляться на алгоритми стійкого сортування (не міняє порядку елементів які мають одинакові ключі) та нестійкого сортування.

**Сортування вибором. O(n2)**

Два цикла. Внутрішній знаходить позицію мінімального чи максимального елемента масива, зовнішній здійснює обмін його на першу чи останню позицію з невідсортованим елементом, сортування хвоста виключивши вже переглянуті (записані) позиції. Використовуються додаткові 2 змінні.

**import** java.util.Arrays;

**protected void** selectSort(**int**[] arrayToSort) {  
 **int** minimal;  
 **int** temp;  
  
 **for** (**int** index = 0; index < arrayToSort.**length** - 1; index++) {  
 minimal = index;  
 **for** (**int** scan = index + 1; scan < arrayToSort.**length**; scan++) {  
 **if** (arrayToSort[scan] < arrayToSort[minimal]) minimal = scan;  
 }  
 temp = arrayToSort[minimal];  
 arrayToSort[minimal] = arrayToSort[index];  
 arrayToSort[index] = temp;  
 }  
 System.***out***.println(Arrays.*toString*(arrayToSort));

**Сортування пузирком. (самий великий в кінці). O(n²)**

Внутрішній цикл біжить вперед міняє 2 сусідніх елемента місцями (за допомогою тимчасової змінної), а потім наступні сусідні по черзі так аби в кінці циклу найбільший елемент масиву стояв в кінці масиву. А зовнішній цикл біжить з кінця, встановлюючи межу для внутрішнього циклу, який з кожним разом відсортовує найбільший елемент в кінець. Повільний на великій кількості значень але швидкий на малій, не використовується в бібліотеках.

**static void** boobleSort(**int**[] arr) {  
 **int** temp;  
 **for** (**int** barier = arr.**length** - 1; barier >= 0; barier--) {  
 **for** (**int** scan = 0; scan < barier; scan++) {  
 **if** (arr[scan] > arr[scan + 1]) {  
 temp = arr[scan];  
 arr[scan] = arr[scan + 1];  
 arr[scan + 1] = temp;  
 }  
 }  
 }  
 System.***out***.println(Arrays.*toString*(arr));  
 }  
}

**Сортування вставками.** (добавлення елемента в відсортований масив). **O(n²)**

Сортування починається з того, що ми беремо перший елемент. Один елемент вже по суті відсортовано. Потім порівнюємо цей перший елемент з другим. І запа’ятовуємо, що ми порівнюємо другий елемент. Якщо другий є меншим за перший, то ми міняємо їх місяцями. Тепер ми маємо відсортований масив з двох елементів. Далі ми беремо третій елемент, запам’ятовуємо, що ми порівнюєм третій елемент (ніби встановлюємо уявну межу, яка кількість елементів у нас вже є відсортованою аби не пробігати масив заново). Якщо третій елемент менший за другий, то далі цей елемент порвнюється з першим елементом і якщо він є меншим, то ми міняємо їх місцями інакше міняємо його місцями з другим. Тпер ми маємо відсортований масив з трьох елементів. Далі беремо четвертий і т.д.

**import** java.util.Arrays;  
  
**public class** InsertionSort {  
 **static void** insertionSort(**int**[] arr) {  
 **for** (**int** index = 1; index < arr.**length**; index++) {  
 **int** currentElement = arr[index];  
 **int** previousKey = index - 1;  
  
 **while** (previousKey >= 0 && arr[previousKey] > currentElement) {  
 arr[previousKey + 1] = arr[previousKey];  
 arr[previousKey] = currentElement;  
 previousKey--;  
 }  
 }  
 System.***out***.println(Arrays.*toString*(arr));  
 }  
}

**Сортування злиттям**. (рек.слияние). (розділяй і володарюй)

Задача зводиться до розбиття її на менші задачі. Масив розбивається пополовинно, сортується кожна половина окремо. Рекурсивно розбиваємо массив до сортованих частин(наприклад по 2 елементи), потім зливаються ці розділені впорядковані частини шляхом порівняння індекса першого елемента з першого масивадо кожного індекса з другого, потім наступного елемента з першого масива до елементів з другого масива. На основі цього сортується кожна частинка.

**private** **static** int[] sortMerge(int[] arr) {

int len = arr.length;

**if** (len < 2) **return** arr;

int middle = len / 2;

**return** merge(sortMerge(Arrays.copyOfRange(arr, 0, middle)),

sortMerge(Arrays.copyOfRange(arr, middle, len)));

**private** **static** int[] merge(int[] arr\_1, int[] arr\_2) {

int len\_1 = arr\_1.length, len\_2 = arr\_2.length;

int a = 0, b = 0, len = len\_1 + len\_2; *// a, b - счетчики в массивах*

int[] result = **new** int[len];

**for** (int i = 0; i < len; i++) {

**if** (b < len\_2 && a < len\_1) {

**if** (arr\_1[a] > arr\_2[b]) result[i] = arr\_2[b++];

**else** result[i] = arr\_1[a++];

} **else** **if** (b < len\_2) {

result[i] = arr\_2[b++];

} **else** {

result[i] = arr\_1[a++];

}

}

**return** result;

**Швидке сортування**. (рек) quick sort O(n log n)

В масиві вибирається опорний елемент (масив ділиться пополовині) ,а всі решта елементи перекидаються вліво від нього, якщо вони менші і вправо, якщовони рівні або більші за опорний елемент. Потім рекурсивно ця дія повторюється для всіх участків масива зліва і зправа. Детальніше, до прикладу ми маємо масив з 14 елементів, вибираємо опорний елемен поділивший масив на 2 - це буде 7-й лемент. Далі 7 елемент порівнюємо з 1, якщо він менший, то залишаємо його на місці і порівнюємо 7 елемент до 2, якщо другий також менший, то залишаємо його на місці, далі порівнюємо 7 елемент з 3 і якщо він більший за 7, то ми його запам’ятовуємо і порівнюємо його з останнім елементом другої половини (14). Якщо цей останній елемент є меншим чи рівним з 7 опорним елементом, то ми міняємо місцями цей 18 елемент з 3, який ми запам’ятали і продовжуємо вже порівнювати наш опорний елемент з 4 елементом і так далі. Таким чином ми отримуємо підмасиви, які також сортуються таким же чином і в кінечному результаті маємо повністю відсортований масив.

**import** java.util.Comparator;

**import** java.util.Random;

**public** **class** Quicksort {

**public** **static** **final** Random RND = **new** Random();

**private** **void** swap(Object[] array, **int** i, **int** j) {

Object tmp = array[i];

array[i] = array[j];

array[j] = tmp;

}

**private** **int** partition(Object[] array, **int** begin, **int** end, Comparator cmp) {

**int** index = begin + RND.nextInt(end - begin + 1);

Object pivot = array[index];

swap(array, index, end);

**for** (int i = index = begin; i < end; ++ i) {

**if** (cmp.compare(array[i], pivot) <= 0) {

swap(array, index++, i);

}

}

swap(array, index, end);

**return** (index);

}

**private** **void** qsort(Object[] array, **int** begin, **int** end, Comparator cmp) {

**if** (end > begin) {

**int** index = partition(array, begin, end, cmp);

qsort(array, begin, index - 1, cmp);

qsort(array, index + 1, end, cmp);

}

}

**public** **void** sort(Object[] array, Comparator cmp) {

qsort(array, 0, array.length - 1, cmp);

}

**Пошук бінарний.**

Також відомий як метод половинного ділення. Це алгоритм пошуку елемента в відсортованому масиві шляхом ділення його по половині. Визначається половина в інтервалі якої лежить шуканий елемент. Знову відбувається ділення масиву по половині і визначається половина в якій може лежати шуканий елемент. Наприклад, ми маємо відсортований масив на 11 елементів (1, 3, 4 , 6, 7, 9, 10, 15, 19, 22, 33). Відповідно ліва межа (L) - індекс масива 0 і права межа (R) - індекс 10. І ми хочемо знайти елемент що містить цифру 3. Ділимо масив пополовині, вибираємо опорний елемент посередині масива (з елементом 9) і порівнюємо його до шуканого елементу. Наш шуканий елемент менший за опорний елемент, тому ми будемо шукати його в лійвій частині масиву - отримуємо межі масиву дл пошуку L - 0, R - 4. Знову цей інтервал ділимо пополовині і порівнюємо опорний елемент (2) до шуканого нами. Бачимо, що наш шуканий елемент менший за опорний і його треба шукати в лівій частині - отримуємо масив для пошуку з межами L - 0, R - 1. Знову цей інтервал ділимо пополовині і порівнюємо опорний елемент (2) до шуканого нами. Бачимо, що він відповідає шуканому - всі щасливі.

Arrays.binarySearch()

Collections.binarySearch()

Приклад:

**public static int** binarySearch0(**int**[] arr, **int** key) {  
 **int** left = 0;  
 **int** right = arr.**length** - 1;  
  
 **while** (left <= right) {  
 **int** middle = (left + right) >>> 1; \\ діл на 2  
 **int** midValue = arr[middle];  
  
 **if** (midValue < key)  
 left = middle + 1;  
 **else if** (midValue > key)  
 right = middle - 1;  
 **else  
 return** middle; *// key found* }  
 **return** -1; *// key not found.* }  
}

**Числа Фібоначчі: (**з розгалуженням**)**

**public static void** fibonnachy(**int** number) {  
 **for** (**int** i = 0; i <= number; i++) {  
 System.***out***.println(**"fib("** + i + **") = "** + *fib*(i));  
 }  
}  
  
**public static int** fib(**int** number) {  
 */\*if (number == 0) {  
 return 1;  
 } else if (number == 1) {  
 return 1;  
 } else {  
 return fib(number - 2) + fib(number - 1);  
 }\*/* **return** number < 2 ? 1 : *fib*(number - 2) + *fib*(number - 1);  
  
}

**Факторіал числа: (**без розгалуження**)**

**static int** factorial(**int** number){  
 **if**(number == 0) **return** 0;  
 **if**(number == 1) **return** 1;  
 **return** number \* *factorial*(number-1);  
}

Рекурсія буває з розгалуженням (числа Фібоначі) і без розгалуження (факторіал).

Безкінечної рекурсії в Джава нема. Вилітає SОЕ.

Якщо при рекурсії не вказати умову виходу з неї, то вилітає переповнення стеку оскільки метод викликає сам себе безкінечно.

В Джаві нема множественного наслідування.

**LinkedList. Односвязного типу.(ссилки в одну сторону)** Коли перший елемент крім свого значення має ще й посилання на наступний елемент. На відміну від масивів не потрібно копіювати хвости масиву аби вставити\видалити значення в середині, потрібно просто перекинути посилання на елемент.

**public class** Node {  
 **private** Object **value**;  
 **private** Node **next**;  
  
 **public** Node() {  
 }  
  
 **public** Node(Object value, Node next) {  
 **this**.**value** = value;  
 **this**.**next** = next;  
 }  
  
 **public** Object getValue() {  
 **return value**;  
 }  
  
 **public void** setValue(Object value) {  
 **this**.**value** = value;  
 }  
  
 **public** Node getNext() {  
 **return next**;  
 }  
  
 **public void** setNext(Node next) {  
 **this**.**next** = next;  
 }  
   
}

В LinkedList можна зробити циклічне співвідношення елементів в списку. Наприклад, останній елемент може зсилатися на перший утворюючи таким чином замкнутий ланцюг. На таких конструкціях ітерація і рекурсія ламаються.

Один з варіантів покращення супроводження коду в майбутньому, це робити клас і його методи абстрактними, а потім, коли нам треба буде добавляти новий функціонал – не переписувати старий код, а унаслідуватися від предка і переоприділяти конкретний метод(функціонал).

**Області пам’яті Джава** (stack, heap, permanent generation)

**Stack** зберігає локальні змінні.

**Рermanent generation** ініціалізує та верифікує класи. Його поля і на що вони посилаються. Методи (зокрема main).

**Heap**. Область пам’яті як зберігає об’єкти з їх технічними полями на які посилаються з стека.

Наприклад массив. Приклад - **пам’ять в heap** виіділяється при застосуванні конструкції **new**.

**Аргументи метода** можна називати **локальними змінними** бо вони належать до області видимості конкретного методу. {}

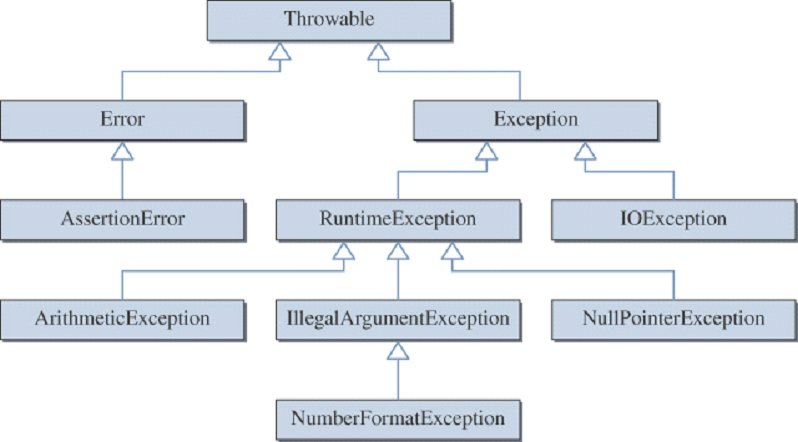
**Передача** за **посиланням** і за **значенням**. Коли зовнішня змінна в методі який повертає void не змінюється, а масив, наприклад, мініяє своє значення бо його значення знаходиться в heap і передається по посиланню.

Коли ми виходимо з методу, то пам’ять в стеку звільняється і локальнізмінні методу стають недоступні.

Завжди відбувається обрахування правої частини, а потім присвоєння до лівої.

В InteliJIDEA потрібно слідукувати який саме клас стоїть на запуск виконання, інакше отримаємо неочікуваний результат.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Class **Throwable** (**checked** перевіряєме, флаг незмінний. impl Serialazible) | | | | | | |
| **Error**(**unchecked** неперевіряєме) | | **Exception**  (**checked** перевіряєме) | | | | |
| OOME | SOE | **IOException** (**checked**) | | **RuntimeException** (**unchecked** неперевіряєме) | | |
| FileNotFoundException | EndOfFileException | Arichmet | NullPointer | Illegal  Arg |

****

**Приклад комерційних експшенів**: користувач хоче зареєструватися, а в базі вже є людина з таким логіном чи поштою, або в нього мінусовий баланс, а він хоче щось купити. В такому випадку створюються власні ексепшни які унаслідуються від нащадків по ієрархії ексепшенів.

Що б створити класс виключень потрібно або наслідуватися від Throwable, або наслідуватися від Exception.

Комбінації, які пропустить компілятор: **try** - **catch**, **try** – **finally**, **try** – **catch** – **finally**.

При **встроюванні в tryдекількох catchі в** потрібно дотримуватися **ієрархії** наслідування класів ексепшенів, оскільки компілятор не пропустить перехоплення ексепшена нащадком після перехоплення ексепшена предком. (недосяжний код). Приклад catch Exception, catch RE.

**try** {  
   
} **catch** (Exception e) {  
} **catch** (RuntimeException e) {  
}

Компілятор підкреслить виключення в catch якщо вище нього стоїть його предок.

Якщо в тілі try з одним throw є декілька catchів, то виповнується тільки один – той, який перший перехопив виключення.

Блоки catch не взаємодіють між собою. Якщо один catch кидає виключення, то сусідній кетч його перехопити не може, навіть якщо виключення співпадає по імені.

Блок **finally** виконується **завжди**, не залежно від того перехопив catch виключення чи ні.

Всі поля після оператоів **return** і **break** також будуть недосяжним кодом.

**if** (**true**) {  
 **return**;  
 **int** x = 0; //unreachable code  
}

**return** в **finally** перекриває **return** з **try**. Також return в finally перекриває екепшн, який вилітає з try. (не бажано в finally ставити return або throw бо можна випадково перебити ексепшн летячий з try). Також throw в finally перекриває return в try.

**1. public static void** main(String[] args) **throws** IOException {  
 **try**{  
 **throw new** Exception();  
 } **catch** (Exception e) {  
 System.***out***.println(**"From catch"**);  
 } **finally** {  
 System.***out***.println(**"From finally"**);  
 **throw new** IOException();  
 }  
}

**2. public static void** main(String[] args) {  
 **try**{  
 **throw new** Exception();  
 } **catch** (Exception e) {  
 System.***out***.println(**"From catch"**);  
 **throw new** IOException();  
 } **finally** {  
 System.***out***.println(**"From finally"**);  
 **return**;  
 }   
}

**IOException** не вилетіло бо **return** з **finally** перебив його.

// **From catch**

**// From finally**

Якщо з try летить ексепшн, а в finally ми зробили throw new і його ж перехопили, тоді статус летячого з try ексепшена відновлюється і воно летить далі, хоч в конструкторі finally є throw.

**try**{  
 **throw new** ArithmeticException();  
} **finally** {  
 **try** {  
 **throw new** IOException();  
 } **catch** (IOException e) {  
 }  
} \\ ArithmeticException

Якщо робити system.out.println() в блоці finally(), а потім штучно кидати виключення throw new… То інколи вивід стектрейсу може перегнати вивід println() хоч той і був вище в коді. Справа в організації друку system.out і system.err. Результат може мінятися і бути нестабільним. Наприклад:

**try**{  
 **throw new** ArithmeticException();  
} **finally** {  
 System.***out***.println(**"Hello"**);  
}

// В консолі буде надруковано Hello і стектрейс виключення але постійно в різному порядку.

Якщо у нас є декілька потоків, то ексепшн «вбиває» не цілу програму, а тільки поток, в якому він вилетів.

Компілятор реагує на checked ексепшни тільки якщо вони були ініційовані оператором throw. Якщо ж checked ексепшн просто створити (Object o = new Exception(); ), то компілятор його не підкреслюватиме.

Якщо один метод викликає інший, який кидає ексепшн (throws), то перший метод в своїй сигнатурі повинен також попередити про можливий ексепшн або взяти його в try – catch.

Якщо в сигнатурі метода вказати throws (відповідно, throws ставиться і в сигнатурі методів які викликали даний метод), то ексепшн(який є checked) можна передати по всьому шляху до кінечного метода по стеку виклику методів який перехоплює цей ексепшн оператором try-catch.

**ArichmmethicExecption** – ділення на нуль.

**NullPointerException** – при ссилочних типах.

//**NOP** – коментарій “no operation”.

//**TODO** – коментарій, що зробити.

**DAO** – data acces sobject рівень проекту який працює з БД.

**-xms$Mbclassname.class** – керування глибини стеку при запуску ВМ.

**-xmx$Mbclassname.class**– керування глибини heap при запуску ВМ.

Якщо немає умови виходу з методу який використовує рекурсію – отримаємо SOE.

**IllegalArgumentException** – рекомендується використовувати для перевірки введених даних в блоці if, наприклад.

Не рекомендується робити нащадка прямо від Throwable оскільки людина яка користуватиметься нашим модулем не зможе передбачити такого бо розраховує на перехоплення або Exception або Error.

Не рекомендується штучно кидати і робити нащадків від Error, OOME, SOE бо прийнято, що з ними працює тільки ВМ.

Рекомендується проекти розбивати на умовні блоки і для кожного з них створювати корінь ексепшенів з назвою яка містить назву блока. Так можна прослідкувати звідки вилітає ексепшн.

Якщо в конструкції **try – finally** в try кинути ексепшн, а в **finally** просто написати **return**, то **обробка** і "політ" **ексепшена** буде **перервано**.

Перегружені методи можуть збуджувати тільки ексепшени, які описані в версії класа предка.

Коли конструктор не видає ексепшенів рекомендується ідеома з принципом, що одразу ж після створення об’єкта який потребує завершення (приклад file.close в роботі з файлами) починати конструкцію try – finally. (file.close виконується в finally). Це гарантує, що file.close виконається тільки в тому випадку, якщо створення файлу не викидає ексепшенів в try верхнього рівня.

**Параметризовані типи – generic.**Параметризація дозволяє створювати класи, інтерфейси і методи в яких **тип** оброблюванних **даних задається як параметр**.

Якщо в блоці **try** оголосити якийсь **ресурс** (приклад **FileInputStream**), а в блоці **finally** викликати **метод close**() для закриття файлу, то виключення яке кинуто в try буде подавлено і в стек вилетить виключення з finally якщо воно там матиме місце.

Для **автоматичного закриття ресурсів** використають **try-з-параметрами**. (приклад try(FileInputStreaminput = …) ), який автоматично закриває ресурси при виході з блоку try і викидає виключення яке було кинуте в try ігноруючи виключення finally.«FileInputStream реализует **интерфейс** **java.lang.AutoCloseable**. Всі класи якиі реалізують цей інтерфейс можуть використовуватися в операторі try з ресурсами.

Файли закриваються (**close**) в порядку зворотньому до їх оголошення.

Рекомендують ексепшени «обгортати» своїми ексепшенами з додаванням деталей про місце й причину виліту ексепшена.

Потрібно давати значення змінній суми, якщо ми хочемо сумувати по ній в циклі.

**int** summ;*// потрібно присвоїти 0.*summ += 0; *// summ буде підкреслено компілятором оскільки вона немає значення.*

Облочки типів потрібні, наприклад, якщо треба передати значення по ссилці**.**

**Клас Float (**ext Numbers, impl Comparable**) –** оболочка для об’єктного представленя примітивого типу float. Наприклад, для роботи примітивного типу флоат з колекціями, які працюють тільки з об’єктами. Котструктори: Float(float a), Float(double a), Float(String str) //throwsNFE.

**Клас Douoble–** оболочка для об’єктного представленя примітивого типу double. Наприклад, для роботи примітивного типу double з колекціями, які працюють тільки з об’єктами. Котструктори: Double(double a), Double(String str) – throwsNFE.

Методи всіх класів оболочок:

**byte byteValue();** - повертає значення об’єкта в байтовому представленні.

**double doubleValue();** - повертає знченя викликаючого об’єкта як тип double.

**float floatValue();** - повертає знченя викликаючого об’єкта як тип float.

**int intValue();** - повертає знченя викликаючого об’єкта як тип int.

**long longValue();** - повертає знченя викликаючого об’єкта як тип long.

**short shortValue();** - повертає знченя викликаючого об’єкта як тип short.

Integer integer = **new** Integer(150);  
**long** lv = integer.longValue();  
System.***out***.println(lv);

**boolean booleanValue();** - повертає знченя викликаючого об’єкта як тип boolean.

**static int compare(float 1, float f2);**- порівнює 2 числа. Повертає -1 - якщо перше менше, 0 – якщо числа рівні, 1 – якщо перше більше. Можна вказувати також числа типу double, float, short, int, long, boolean.

**int compareTo(Float f);**- порівнює викликаюче число відносно даного в конструкторі. Наприклад: varFloat.compareTo(varFloat2); Можна вказувати також числа типу double, float, short, int, long, boolean.

Integer x = 12;  
Integer y = 15;  
**int** z = x.compareTo(y);  
 System.***out***.println(z);

**boolean equals(Object FloatObj);** - повертає істину, якщо об’єкти ідентичні. В протилежному випадку false.

**int hashCode();** - викликає хеш-код об’єкта. Можна вказувати також числа типу double, boolean.

Integer x = 12;  
System.***out***.println(x.hashCode()); //12

**boolean isInfinity();** - повертає true якщо число має безкінечне значення.

Float y = **new** Float(**"12.25"**);  
**boolean** x = y.isInfinite();  
System.***out***.println(x); //false

**boolean isNaN();** - повертає true якщо значення е нє числом.

**static boolean isInfinity(float f);** - повертає істину, якщо число має безкінечне значення. Можна вказувати також числа типу double.

**static boolean isNan(float f);** - повертає true якщо значення не є числом. Можна вказувати також числа типу double.

**static float parseFloat(String str);** //throwsNFE – повертає значення типу флоат з строки. (основа десятична).

String toInt = **"1000"**;  
**int** x = Integer.*parseInt*(toInt);  
System.***out***.println(x);

**static double parseDouble(String str);** //throwsNFE – повертає значення типу double з строки. (основа десятична).

**static byte parseByte(String str);** //throwsNFE – повертає значення типу byte з строки. (основа десятична).

**static short parseShort(String str);** //throwsNFE – повертає значення типу short з строки. (основа десятична).

**static long parseLong(String str);** //throwsNFE – повертає значення типу long з строки. (основа десятична).

**static int parseInt(String str);** //throwsNFE – повертає значення типу int з строки. (основа десятична).

**static boolean parseBoolean(String str);** - істиа, якщо в строці true.

**static Float valueOf(float f); -** повертає об’єктне представлення числа f.

Integer integer = **new** Integer(10);  
String string = **"125"**;  
System.***out***.println(**"Value: "** + integer.*valueOf*(string));

**static Double valueOf(double d); -** повертає об’єктне представлення числа d.

**static Byte valueOf(byte b); -** повертає об’єктне представлення числа b.

**static Short valueOf(short s); -** повертає об’єктне представлення числа s.

**static Integer valueOf(int i); -** повертає об’єктне представлення числа i.

**static Long valueOf(long l); -** повертає об’єктне представлення числа яке знаходиться в строці str.

**static Float valueOf(String str); -** повертає об’єктне представлення числа яке знаходиться в строці str.

**static Integer valueOf(String str); -** повертає об’єктне представлення числа яке знаходиться в строці str.

**static Byte valueOf(String str); -** повертає об’єктне представлення числа яке знаходиться в строці str.

**static Short valueOf(String str); -** повертає об’єктне представлення числа яке знаходиться в строці str.

**static Long valueOf(String str); -** повертає об’єктне представлення числа яке знаходиться в строці str.

**String toString()** – викликає строкове представлення об’єкта.

**static String toString(float f);** //throwsNFE- викликає строкове представлення числа f. Можна вказувати також числа типу double, float, short, int, long.

**static Byte decode(String str);**// throwsNFE – повертає об’єкт Byte який має значення вказане в строці str.

**static Long decode(String str);**// throwsNFE – повертає об’єкт Long який має значення вказане в строці str.

**static Short decode(String str);**// throwsNFE – повертає об’єкт Short який має значення вказане в строці str.

**static int rotateLeft(int a, int n), static int rotateRigth(int a, int n);** - зміщення вліво\вправо числа а на n позицій. Застосовується і до long.

**static int signum(int a); -** вертає -1 – якщо а від’ємне, 0 – якщо а нуль і 1 - якщо а більше нуля.

**Клас Character (impl Serializable, Comparable)**– оболочка для типу char. Конструктор: Character(char c); Щоб отримати значення, яке поміщено в оболочку Character потрібно викликати метод char charValue(); який поверне символ.

Деякі методи класа Characer:

**static boolean isDigit(char c);** - повертає істину, якщо символ десяткове число.

**static boolean isLetter(char c);** - істина, якшо с – буква.

**static boolean isLetterOrDigit(char c);** - істина, якщо с – буква або десяткове число.

**static boolean isLowerCase(char c);** - істина, якщо с – символ нижнього регістру.

**static boolean isUpperCase(char c);** - істина, якщо с – символ верхнього регістру.

**static boolean isSpaceChar(char c);** - істина, якщо с – пробіл кодування Юнікод.

**static boolean isWhiteSpace(char c);** - істина, якщо с – символ пробілу.

**static chartoUpperCase(char c); -** символ с до верхнього регістру.

**static char toLowerCase(char c);** - символ с до нижнього регістру.

**int compareTo(Character c);** // -1, 0, 1.

З JDK 5 клас Character почав підтримку 32 бітного представлення символів Юнікод. (раніше було 16 бітне представлення, що є рівним розміру типу char). Раніше символи лежали в діапазоні від 0 до FFFF, а в 32 представленні від 0 до 10FFFF. Це зумовлює проблеми, оскільки 32 бітний елемент не поміщається в 16 бітний char. Вирішується ця проблема або використанням двох змінних char, одна з яких називається старшою сигнатурою, а інша молодшою сигнатурою. Вводиться поняття кодової точки. Або використовуються перегружені методи Character які замість char приймають int.

Ще деякі методи класу Boolean:

**static boolean getBoolean(String ім’яОпції);** - повертає істину, якщо системна опція в строці ім’яОпції є true.

**Клас Void; -** має тільки одне поле TYPE. Екземпляри даного класу не створюються.

**void ensureCapasity(int size), void setLength(int size)** – задають об’єм\довжину об’єкта СБ. Якщо вказаний параметр буде менший за існуючий в об’єкті, то зайва частина строки буде відсікатися.

**char charAt(intindex), void setCharAt(intindex, charChar)** – повертає \ установлює символ в вказаній позиції.

**void getChars(int startSubstr, int endSubstr, char[] arrTo, int positionInArr)** – копіювая символів з об’єкта СБ в масив. Потрібно впевнитися, що масив достатньої довжини. Строка буде дожиною від startSubstr до endSubstr-1.

**StringBuffer append()** – об’єднує об’єкт СБ з строковим представленям любого типу (найчастіше його отримують методом valueOf). Просто String не має метода append() бо він є незмінним.

StringBuffer sb = **new** StringBuffer(20);  
**int** a = 17;  
String str = sb.append(**"hello - "**).append(a).append(**";"**).toString();  
System.***out***.println(str); // hello - 17;

**StringBuffer insert(int index, String str)** – дозволяє вставити строку в іншу строку в задану позицію. Перегружений для всіх типів даних.

StringBuffer sb = **new** StringBuffer(**"I Java"**);  
sb.insert(2, **"like "**);

System.***out***.println(sb); *// “I like Java”*

**StringBuffer reverse(); -**повертає строку у зворотньому порядку.

StringBuffer sb = **new** StringBuffer(**"abc"**);  
sb.reverse();   
System.***out***.println(sb); *// “cba”*

**StringBuffer delete(int indexStart, int indexEnd), StringBuffer deleteCharAt(int index)** – видалення підстроки\символа в заданій позиції.

StringBuffer sb = new StringBuffer(“abc”);

sb.delete(2); // “ab”

**StringBuffer replace(int indexStart, int indexEnd, String str)** – заміна підстроки в строці.

**StringBuffer substring(int index), substring (int indexStart, int indexEnd) –** повертає підстроку від початкового індекса або в заданому інтервалі.

**void trimToSize()** – обрізає виділений буфер до розмірів реального об’єкту всередині.

**Класс StringBuilder** не є синхронізованим на відміну від StringBuffer. Тому при роботі з потоками краще використовувати StringBuffer. А якщо програма однопоточна, то ефективнішим буде StringBuilder.

**Класс Number** – абстрактний суперклас який реалізують класи оболочки для примітивних типів. Містить в собі абстрактні методи типу double doubleValue(), byte byteValue(), int intValue() etc… Зачення які повертають ці методи можуть бути округлені або обрізані.

**Клас Runtime** – клас в який інкапсульовано середовище виконання. **Створити об’єкт даного класу не можливо.** Можна тільки отримати посилання на його об’єкт – staticRuntime.getRuntime(); Методи класа:

**void addShutdownHook(Thread thr);** - реєструє потік, який запускається при зупинці віртуальної машини.

**Process exec(String progName);** //throws IO – виповняє програму progName як окремий процес.

**Process exec(String[] commandLineArray);** //throwsIO – виповняє командну строку передану в масиві commandLineArray.

Runtime rnt = Runtime.*getRuntime*();

Process prc = **null**;

**try**{

prc = rnt.exec("notepad");

prc.waitFor();

}**catch**(Exception e){

//NOP

}

**void exit(int exitCode)** – перериває виконаня і передає exitCode процесу предку. **0** – прийнято кодом нормального завершення. Інші значення символізують різні типи помилок.

**long freeMemory();** - приблизне значення вільних байтів середи виконаня.

**long totalMemory();** - повертає загальну кількість байт яка є вільною для програми.

**void gc();** - ініціює збір мусора.

**static Runtime getRuntime();** - повертає поточний об’єкт класа Runtime.

**void halt(int code);** - негайно зупиняє ВМ і завершає всі поток включно фінальні. Повертає код викликаючому процесу.

**void load(String libaryPath);** - загружає динамічну бібліотеку libaryName де повинен бути вказаний повний шлях до неї.

**void loadLibary(String libaryName); -** загружає динамічну бібліотеку ім’я якої асоційовано з libaryName.

**boolean removeShutdownHook(Thread thr);** - видаляє потік thr зі списку потоків які мають запускатися після зупинки ВМ. Повертає істину, якщо видалення пройшло успішно.

**void runFinalization();** - ініціює методи finalize() для об’єктів які не були використані, але не повернули значення.

**Перечислення** – списки йменованих констант. Кожна константа є екземпляром даного класу-перечислення. В Джава реалізовані як клас. Можуть мати свої конструктори, методи, змінні екземпляра і реалізації інтерфейсів. Створюються за допомогою ключового слова **enum**. І описуються по конвеншену в класі після методів. В методі main, наприклад їх описати не вийде. До прикладу перечислення сортів яблук:

**public class** ExceptionPractice {  
 **public static void** main(String[] args) {  
 }  
 **enum** Apple { ***Johnatan***, ***GlobalRed***, ***WhiteDel*** } // в дужках константи перечислень //кожна об’явлена як public static final  
}

**Неможливо** створити екземпляр перечислень за допомогою конструкції **new**. Перечислення використовуються як примітивні типи. Значення змінної типу перечислення (Apple) можуть бути рівними тільки константам які є в перечисленні. Приклад:

Apple app;  
app = Apple.***Johnatan***; *//правильно*app = Apple.AnotherApple; *//неправильно*

Методи перечислень:

**public static<enumtype>[] values();** - повертає масив значень які є в перечисленні.

Apple allApples[] = Apple.*values*();

**public static<enumtype> valueOf(String str)** – повертає константу перечислення яка еквівалентна значення в строці.

Apple app = Apple.*valueOf*(**"Johnatan"**);

// [Johnatan, GlobalRed, WhiteDel]

Приклади конструкторів і методів перечислень:

**enum** Apple {  
 ***Johnatan***(10), ***GlobalRed***(5), ***WhiteDel***(7);  
 **private int price**;  
  
 Apple(**int** p) {  
 **price** = p;  
 }  
  
 **int** getPrice() {  
 System.***out***.print(**"price: "**);  
 **return price**;  
 }  
}

При чому, можна викликати методи перечисленняя для будь-якої з констант:

**int** x = Apple.***Johnatan***.getPrice();

**Перечислення не можуть наслідувати інші класи.** По замовчуванню вони вже наслідують клас **java.lang.Enum.**

Перечислення не можуть бути суперкласом.

**final int ordinal();** - повертає номер викликаючої константи в списку перечислень. Нумерація починається з нуля.

**Автоупаковка** – процес автоматично інкапсулювання примітивного типу в його відповідну об’єктну оболонку.

Integer iob = 100;

int i = iob;

Також можливе автоматичне приведення типів при автоупаковці:

Integer int = 100;

Double dou = 200.2;

dou = int + dou; // 300.2

**Аннотація (метаданні)** – засіб який дозволяє документувати код. Створюються на основі механізму основаного на інтерфейсі. Аннотація не може використовувати розширення (extends) проте, автоматично розширює інтерфейс Annotation (пакет java.lang.annotation) в якім перевизначенометоди toString(), equals(), hashCode(), annotationType() –повертає тип аннотації як об’єкт Class. Аннотувати можна класи, методи, поля, константи перечислень і навіть самі аннотації.

@MyAnno (str = “This is anno”, val = 115)

public static void myMath();

Правило утримання аннотацій – визначають момент, коли аннотація відкидається. Визначено 3 таких правила, які інкапсульовані в перечислення java.lang.annotation. **RetentionPolicy** – це SOURCE, CLASS, RUNTIME. **SOURCE** – аннотації зберігаються в вихідному файлі і відкидаються при компіляції. **CLASS** – аннотації зберігаються в файлі .class під час компіляції, але не доступні для ВМ. **RUNTIME** - аннотації зберігаються в файлі .class під час компіляції і є доступними для ВМ, тобто установлюють найбільший рівень зберігання для аннотацій. RUNTIME також дозволяє опитувати аннотації в програмі за допомогою рефлексії.

@Retention(RetentionPolicy.CLASS)

@interface MyAnno{ String str(); int val(); }

**Аннотації локальних** змінних, на відміну від об’єктів**, не зберігаються** в файлі .class.

Одна аннотація не може наслідувати іншу. Всі методи які оголошені в аннотації повинні бути без параметрів і в цих методах не можна використовувати throws.

**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, 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(), interrupt(), interrupted().

**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.

|  |  |
| --- | --- |
| 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!"**);  
 }  
  
}

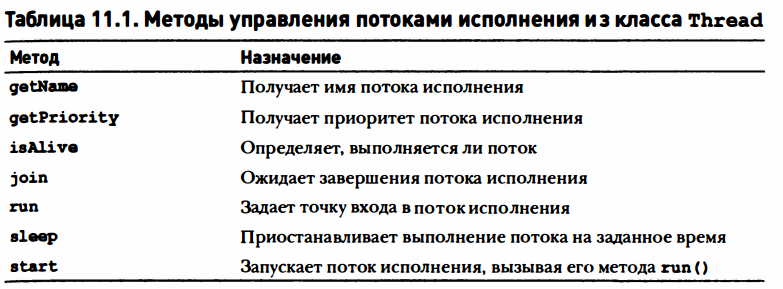
**2. public class** HelloThread **extends** Thread {  
  
 **public static void** main(String args[]) {  
 HelloThread thread = **new** HelloThread();  
 thread.start();

*// 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.

Вибір способу створення потоку залежить від ситуації. Якшо створювати екземпляр класу Thread (extendsThread) то варто подумати чи перевизначаємо ми в коді ще якісь методи крім run(). Програмісти вважають, що наслідування варто використовувати тільки в тих випадках, коли йдеться мова про модифікацію класу предка. Тому, в випадку, коли при наслідування класа Thread перевизначається тільки run(), то краще реалізувати інтерфейс Runnable. Крім того, при реалізації інтерфейсу Runnable наш клас не повинен наслідувати Thread аби звільнитися від наслідування інших класів.

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 ThreadDeathя ке 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 Threadз 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) // …

**java.util.concurrent** – функціональні можливості як альтернатива встроєним методам синхронізації і міжпотокових комунікацій.Ключов іелементи: **синхронізатори, виконавці, паралельні колекції, Fork/JoinFramework**.

**-Синхронізатори**– пропонують високорівневі способи синхронізації і взаємодії між декількома потоками.(Класисинхроніізаторів: **Semaphore**, **CountDownLatch**, **CyclicBarrier**, **Exchanger**, **Phaser** )

Клас**Semaphore –** керує доступом до загального ресурсу за допомогою лічильника. Якщо лічильник більше нуля – доступ дозволяється, якщо лічильник рівний нулю – доступ забороняється. Відповідно, потокові який хоче отримати доступ до об’єкта потрібно отримати доступ у семафора. Якщо потік отримує доступ у семафора, то лічильник семафора зменшується на одиницю. Інакше, потік буде заблокований доти, доки не зможе отримати дозвіл у семафора. Якщо доступ до об’єкта потокові більше не потрібен, він звільняє дозвіл, що збільшує лічильник семафора на одиницю, в цей час, якщо якийсь інший потік чекає в черзі дозволу на роботу з об’єктом – він його отримує.

Конструктори: **Semaphore**(**int** число**) –** задає значення лічильника потоків які можуть одночасно працювати з об’єктом. Тобто можна задати «0» і об’єкт з цим семафором не виконуватиметься першим якщо є об’єкт з семафором «1». Тобто є можливість задавати початкові умови синхронізації.**Semaphore**(**int** число, **booleаn** як) – установка параметра «**true**» дозволить потокам заходити в об’єкт в тому порядку, в якому вони ставали в чергу.

Щоб отримати дозвіл викликається метод voidacquire(). Щоб звільнити дозвіл після виконаних дій – метод release().

Клас**CountDownLatch (защолка) –** заставляє чекати потік поки не виповниться певна кількість подій передана в конструкторі класу CountDownLatch. Кожен раз, коли відбувається подія, значення лічильника CountDownLatch зменшується на одиницю. Коли значення лічильника стане рівним нулю защолки буде знята.

Конструктор **CountDownLatch(int число).** Число – кількість допустимих подій до зняття защолки. Для виклику защолки використовується метод **void await(),** тоді очікування відбувається доти, доки лічильник пов’язаний з об’єктом класу CountDownLatch, який викликав метод await(), не стане рівним нулю. Щоб сповістити лічильник про подію викликається метод countDown(), який зменшує лічильник на одиницю.

Клас**CyclicBarrier**–задає кількість потоків, які мають виповнитися до певної точки (await() ) і на ній призупинитися доки весь набір потоків не дійде до цієї точки.CyclicBarrier визначає об’єкт синхронізації який призупиняється доки всі зазначені потоки не дійдуть до бар’єрної точки.Конструктори: **new CyclicBarrier(int кількість\_потоків);** - задає кількість\_потоків, які мають ввійти в синхронізований об’єкт і достигнути бар’єру. **new CyclicBarrier(int кількість\_потоків, newRunnable дія);** - задає кількість потоків і Runnableдію, яка виконається після проходження потоками бар’єру. Приклад: CyclicBarrier cbr = new CyclicBarrier(3, new BarAction()); - після досягнення синхронізованим об’єктом (cbr.await() ) бар’єру виконується клас Runnable.

Клас**Exchanger**– призначений для синхронізації обміну даних між двома потоками.Exchanger очікує поки обидва потоки які його використовують виключуть метод exchnge(), тоді відбудеться обмін даних між потоками. Конструктор:Exchanger<тип\_даних>. Приклад Exchanger<String> exch = new Exchanger<String>();, потім класу передається ця змінна - new Потік(exch);, а в самому класі для обміну викликається метод exch.exchange(str); Метод exchange() заключається в тім, що його робота не буде продовжена до тих пір, поки два потоки не викличуть його для одного й того ж об’єкту класа Exchanger.

Клас**Phaser**–забезпечує синхронізацію між потоками які мають декілька фаз виконання. Наприклад: ми маємо 3 фази. Перша фаза перевіряє дані клієнта в базі і суму товару. Коли ці потоки відпрацюють починається фаза 2: іншими потоками вираховується вартість доставки і ПДВ. Фаза 3: виконується замовлення і вираховуються строки доставки. Фази в конструкторі іменуються з 0 (тобто перша фаза – 0, друга – 1, третя - 2… getPhase() повертає номер фази). Клас Phaser прцює подібно класу CyclicBarrier за виключенням того, що підтримує декілька фаз (Якщо використати Phaser для синхронізації тільки однієї фази, то він діє як CyclicBarrier). Конструктори: **Phaser(),Phaser(int число) –** число(кількисть зареєстрованих сторін). Використання: Спочатку створюється новий об’єкт класа Phaser. (Phaserphaser = newPhaser();), потім на фазері реєструється одна чи декілька сторін методом intregister(); (phaser.register();- повертає номер зареєстрованої фази) або при вказанні необхідної кількості сторін в конструкторі. Для кожної зареєстрованої сторони є фазер, який чекає закінчення фази всіма зареєстрованими сторонами (сторона сповіщає про закінчення фази методами arrive() – повертає поточний номер фази. Якщо робота фазера закінчена, то повертає від’ємне значення. Сповіщає про те, що сторона(потік) завершили завдання. Не призупиняє викликаючого виконання поточного потоку – тобто, викликаючий потік не чекає завершення фази, arriveAndAwaitAdvance() – повертає номер наступної фази, відємне значення, коли фазер завершив роботу. Вказує на завершення фази і чекає поки фазу завершать всі зареєстровані сторони (в тому числі очікує й головний потік)). Коли кількість завершених сторін рівна кількості зареєстрованих сторін – фаза рахується завершеною. Тобто, як тільки всі сторони готові фаза рахується закінченою і фазер переходить до наступної фази чи завершає роботу.arriveAndDeregister – відміняє реєстрацію сторони на фазер. Повертає поточний номер фази. Якщо робота фазера закінчена, то повертає від’ємне значення.awaitAdvance(intфаза) – очікує певної фази, номер якої вказано в «фаза». forceTermination() – переводить фазер в стан негайного завершення.

**TimeUnit**– параметр, який було введено для вирішення проблеми задання таймауту.Наприклад, в Джава цей параметр задається в мілісекундах, а в бібліотеці Tоmcat – в секундах. Тому ввели TimeUnit(long скільки, enumTimeUnit чого). Приклад: (100, MILLISECONDS, DAYS, HOURS…). Перечислення TimeUnit мають також 3 метода: voidsleep(long мілісек),voidtimedJoin(Threadпотік, long мілісек), voidtimedWait(Object обєкт, long мілісек)

Фабрика потоків – клас,з одним методом, якому даєш Runnable, а той запускає new Thread.

**-Виконавці (executor, альтернатива Thread) –** керують виконанням потоку. Перший в ієрархії є **інтерфейс Executor** який служить **для запуску потока**(потік запускає метод voidexecute(newRunnable потік);)**.** Також з виконавцями пов’язані інтерфейси **Future<V>**(має значення яке повертає потік після виконання. Таким чином його значення визначиться «в майбутньому», коли потік завершить роботу– так званаспилка на майбутній результат). Це механізм, який обходить асинхронність і дозволяє перевіряти стан дій переданих потоку. (ніби реєструючи квитанцію на потік з таском) Асинхронність – коли ми передаємо таск потоку, він йде на виконання і ми втрачаємо зв’язок з тим, на якій стадії виконання знаходиться потік і виконуємося собі далі не очікуючи закінчення потоку з таском. Приклад: Future<String>futureResult = pool.submit(newCallable<String>). Тоді ми можемо, наприклад в циклах, дізнаватися про стан потоку Callable() використовуючи метод futureResult.isDone() і витягнути результат методом futureResult.get(); i **Інтерфейс Callable<V> -** те ж саме, що й Runnable тільки параметризований, а не void.(визначає потік, який може вертати значення. Приклад використання: паралельні математичні обрахунки). Callable визначає тільки один метод call(). В середиді метода call() визначається задача яку потрібно виконати, коли вона виконана метод повертає результат, або ексепшн при збої виконання задачі. Задача Callаble запускається на виконання методом submit() визначеним в інтерфейсі ExecutorService. Результат задачі з Callаble буде вертатися через об’єкт інтерфейса Future (приклад: : Future<String>futureResult = pool.submit(newCallable<String>)). Щоб отримати результат задачі newCallable<String> потрібно викликати метод get() в обєкті Future<String>futureResult який повертатиме результат. (futureResult.get() )

**Інтерфейс ExecutorService** розширює інтерфейс Executor і надає додаткові методи для керування виконання потоку. Наприклад, метод void shutdown(), якийзупиняє всі потоки, що находяться в даний момент під керівництвом екземпляра об’єкта ExecutorService . (Реаліізації: класи **ThreadPoolExecutor**, **ScheduledThreadPoolExecutor**, **ForkJoinPool**).

**Клас ThreadPoolExecutor(пул потока)–**реалізує інтерфейси Executor і ExecutorService. Забезпечує підтримку керованого пула потоків.

**Пул потоків.** Якщо ми маємо сукупність потоків, то ми можемо об’єднати їх в один пул. Це робиться для того аби зекономити на ресурсах процесора при старті і завершенні потока. Тобто встановлюється якась кількість потоків (core), каприклад 10, потім стратується пул з 20 потоків і ці потоки по черзі завершають свою роботу, то коли кількість потоків дійде до 10 - процесор не буде викликати далі інструкції для завершення потоків й витрачати пам’ять, він буде вважати, що в нього «на ходу» все ще 10 потоків, хоч насправді працюючих потоків може бути й менше. Також можна вказати максимальну кількість потків для пула(max) і коли число потоків дійде до цієї точки всі решта потоки ставитимуться в блоковану чергу(BlockingQueue). А якщо не справляється з навантаженням й BlockingQueue, то викликається RejectedExecutionHandler**.**

**Інтерфейс RejectedExecutionHandler**– використовується, коли і пул потоків досягнув максимуму (max), і блокуючи черга (BlockingQueue) досягла максимуму. Тоді потік передається RejectedExecutionHandler в його метод rejectedExecution(Runnable r, ThreadPoolExecutor executor);. Приклад, коли веб-сервер перегружений запитами – вернути клієнту, який довго очікуватиме черги сторінку про загруженість сервера.

**Інтерфейс SheduledExecutorService –** розширює інтерфейс ExecutorService для підтримки планування потоків. (Executor/ExecutorService/ SheduledExecutorService).

**КласScheduledThreadPoolExecutor –** реалізує інтерфейс SheduledExecutorService для підтримки можливості планування пула потоків.

Хоча класи ThreadPoolExecutor і ScheduledThreadPoolExecutorможна використовувати напряму, частіше всього приходиться використовувати один з фабричних методів допоміжного класа Executors (staticExecutorService**newCachedThreadPool()** – створює пул потоків який не тільки добавляє потоки за необхідністю, а й по можливості їх повторно використовує.staticExecutorService**newFixedThreadPoll(int кількістьПотоків)** – створює пул потоків фіксованої кількості.staticSheduledExecutorService**newSheduledThreadPoll(int кількістьПотоків)**– створює пул потоків, в якім можна виконувати їх планування)

**-Паралельні колекції. –** колекції з паралельним виконання альтернативні встроєним колекціям Collections Framework. (**ConcurrentHashMap**, **ConcurrentLinkedQueue**, **CopyOnWriteArrayList**)

-**Fork/Join –** реалізує інтерфейси Executor і ExecutorService . Підтримує паралельне програмування. (Основні класи: **ForkJoinTask**, **ForkJoinPool**, **RecursiveTask**, **RecursiveAction**)

**Fork/Join (паралельне програмування)** – технологія яка в своїх потребах використовує багатоядерність процесорів. Спрощує створення декількох потоків, автоматизує використання декількох процесорів. Тобто з’являється можливість автоматично масштабувати свій додаток в залежності від кількості доступних процесорів. Ядро Fork**/**Join складають класи: ForkJoinTask<V>, ForkJoinPool, RecursiveAction, RecursiveTask<V>. Як правило, Fork**/**Join використовує рекурсивну методологію «разделяй и властвуй».

**ForkJoinTask<V>** - абстрактний клас - визначає задачу, якою може керувати ForkJoinPool (такий механізм дозволяє керувати великою кількістю задач невеликою фактичною кількістю потоків – перевага над Thread). Відрізняється від Thread тим, що представляє собою не потік виконання, а полегшену абстракцію задачі;Основні методи: final**ForkJoinTask<V>fork()** – передає задачу для асинхронного виконання. Тобто потік, який викликав fоrk() продовжує виконуватися. Після того як задача запланована методом fork() він повертає this; final<V>**join()**–очікує завершення задачі для якої він викликаний. Повертається результат задачі. Таким чином, за допомогою методів fork() i join() можна запустити декілька задач і очікувати їх завершення;final<V>**invoke() –** об’єднує операції розгалуження і поєднання в єдиний виклик оскільки запускає задачу, а потім чекає на її завершення.Повертає результат задачі; staticvoid**invokeAll()** – можна викликати декілька задач одразу. Конструктор: staticvoidinvokeAll(ForkJoinTask<?> задача1, ForkJoinTask<?> задача2), або staticvoidinvokeAll(ForkJoinTask<?>…список задач);**cancel(**booleanуспішноПерервано**)**–відміна задачі. **isCanceled()**– перевірка на перерваність задачі.**shutdown()** –поточні дії ще виконуються але інші задачі запустити вже неможливо. **shutdownNow()** –зупиняє поточні дії (поточний пул).

**RecursiveAction**–клас задач, похідний від класу ForkJoinTask<V>, не повертає значення (інкапсулює задачу). Як правило треба розширити клас RecursiveAction щоби створити задачу яка повертає void.Метод: protectedabstractvoidcompute() – метод в який поміщається код задачі. Як правило використовується для реалізації рекурсивного рішення задачі, яка не повертає результат.

**RecursiveTask<V>** - клас задач, похідний від класу ForkJoinTask<V>, інкапсулює задачу і повертає значення.Метод: protectedabstractVcompute() – метод в який поміщається код задачі. (protected означає, що він може бути викликаний тільки методами даного класу або класів наслідуваних від нього).

**КласForkJoinPool –** виконання об’єктів класу ForkJoinTask відбувається в рамках об’єкта класа ForkJoinPool який керує виконанням задач. Тому, щоби запустити об’єкт ForkJoinTask, спочатку треба створити об’єкт ForkJoinPool. Конструктори: ForkJoinPool() – створює стандартний пул який забезпечує рівень паралелізму рівний кількості процесорів в системі; ForkJoinPool(int рівеньПаралл) - дозволяє задати рівень паралелизму. Тобто кількість одночасно виконуваних потоків (кількість одночасно заданих задач не повинно перевищувати кількості процесорів); Після створення екземпляра класа ForkJoinPool задачу можна запустити декількома способами. Один з них – це запуск основної задачі, яка нерідко породжує інші задачі якими також керує пул. Найпоширенішим способом запуску основної задачі є виклик метода invoke() класа ForkJoinPool. (<T>Tinvoke(ForkJoinTask<V> задача); - в такому випадку потік, який викликав задачу буде очікувати до завершення методу invoke()). Щоб запустити задачу і не очікувати потоком до завершення методу invoke() – використовують метод execute(). (voidexecute(ForkJoinTask<?>задача)). Клас ForkJoinPool керує своїми потоками реалізуючи ідею «захват задачі – work-stealing». Кожний робочий поток підтримує чергу задач, якщо черга одного робочого потока зробиться вільною, то цей потік візьме задачу від іншого робочого потока. Це сприяє покращенню продуктивності і підтримці балансу навантаження.ForkJoinPool використовує потоки демони, таким чином немає необхідності явно завершати роботу класа ForkJoinPool (хоча це можливо методом shutdown() ).

Різниця між традиційноюбагатопоточністюі паралельним програмуванням полягає в тому, що багатопоточність проектувалася під використання одного процесора. Тобто один потік очікував завершення іншого або ж використовувалося квантування часу процесора, коли була потрібна паралельна робота декількох потоків. Паралельне ж програмування заточене під багатоядерні процесори і здатне розділяти роботу потоків між ядрами процесора. Тобто два і більше потоки можуть виконуватися одночасно на різних ядрах процесора не використовуючи механізм квантування.

**java.util.concurent.atomic** – пакетдлявикористаннязміннихвпараллельномувиконанні. Пропонує засоби зміни значень змінної без використання блокіровок. (Класи: **AtomicInteger**, **AtomicLong**…Методи:**compareAndSet**(), **decrementAndGet**(), **getAndSet**()/ Ці методи працюють в режимі однієї безперервної операції )

**java.util.concurrent.atomic** – пропонує додаткові функціональні можливості при синхронізації читання чи запису деяких типів змінних (AtomicInteger…etc). В пакеті доступні методи, які отримують, записують чи порівнюють значення змінних під час однієї неперервної (тобто атомарної) операції. Тобто тепер не потрібні ні синхронізатори, ні блокування. Приклад: створюється класSharedInt з статичноію змінною з якою будуть атомарно працювати потоки (staticAtomicInteger*varAtomInt* = newAtomicInteger(0);), а потім в тілі потока для запису змін в змінну викликається один з методів, наприклад, SharedInt.*varAtomInt*.getAndSet(значення).

**java.util.concurent.locks –** пропонує альтернативний варіант роботи з синхронізованими методами (synchronized). В його основі лежить інтерфейс Lock, який визначає механізм доступу до об’єкта чи забороні доступу до об’єкта.Ключові методи: **lock**(), **tryLock**(), **unlock**().

Методи інтерфейса Lock: voidlock() - блокує, voidunlock() – знімає блокування, tryLock() – дізнається чи блокування вільне (повертає true) чи зайняте (повертає false), також має конструктор tryLock(long скількиЧекати, ТimeUnit чого).

**ReentrantLock**. (класс інтерфейсу Lock) Метод unlock() в реалізації ReentrantLockможна використовувати декілька раз на відміну від synchronized() який використовує облать фігурних скобок. Приклад: f(); lock();тіло методу f(); -> виклик із методу інший метод g(); lock(); тіло методу g(); unlock(); -> вихід в метод f(); unlock(); Тобто метод вкладений в інший метод і обдива мають свої конструкції lock-unlock. При чому, можливо зробити lock() в методі f(), а unlock() в методі g() і все буде коректно працювати. Приклад використання: добавлення даних в деревовидну структуру. Спочатку захоплюється блокування вершини, потім відбувається спуск по дереву, коли ми знаходимо наступне звено дерева, ми захоплюємо його блокування, а тобі відпускаємо блокування в вершині. (тобто рух відбувається за принципом захопив наступний, потім відпустив попередній)

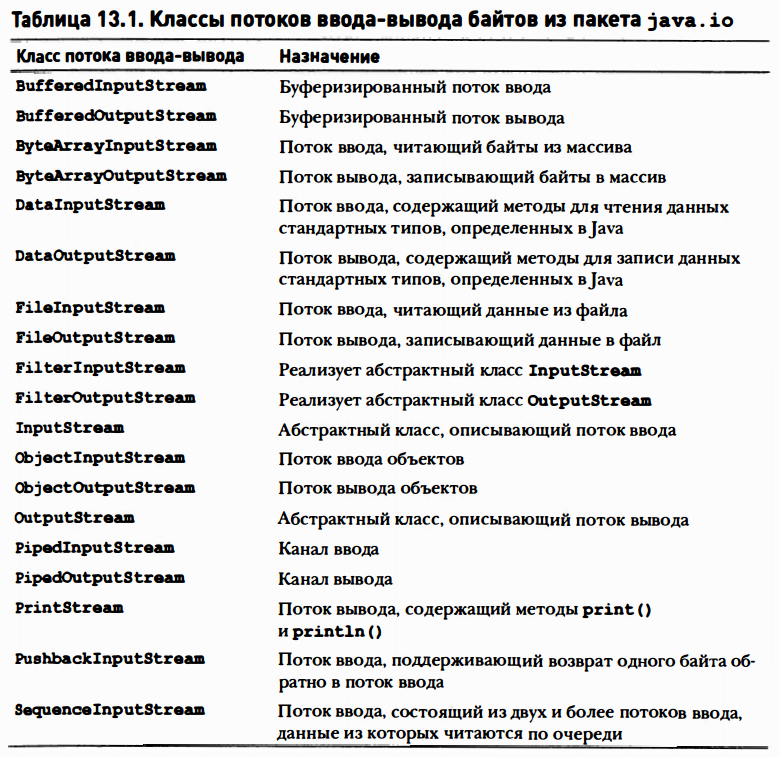
ReentrantLock має декілька wait-set і один block-set.Також він (і багато інших методів j.u.c) можуть забезпечити «чесне пробудження» потоків з wait-set. (на відміну від метода notify() який вибирав будь-який потік, що очікує чергу не звертаючи уваги котрий з них довше чекає на виконання.)

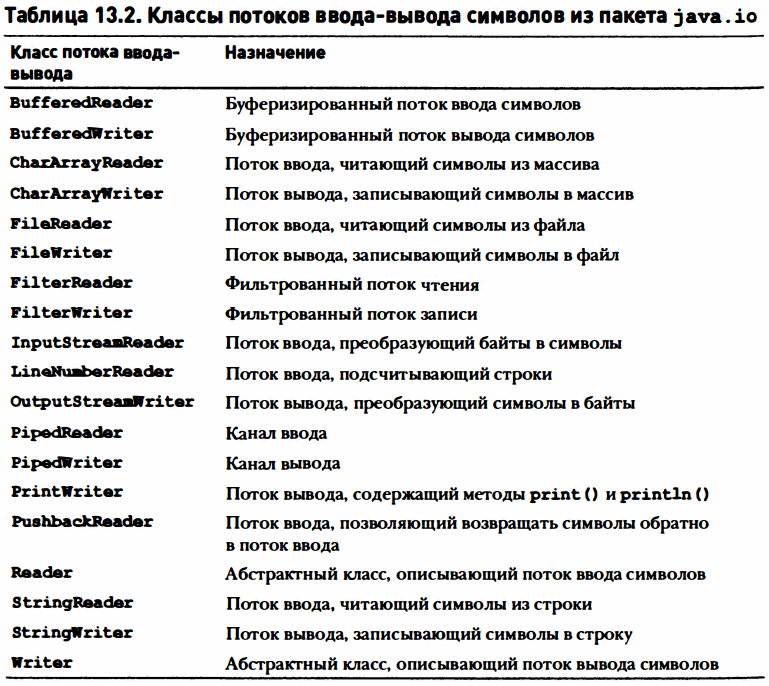
**ReentrantReadWriteLock.** Має методи readlock() (shared), writelock() (exsclusive). Ідея полягає в тому, що якщо взяти exsclusive «лочку», то більше ніяку іншу взяти не можна. Якщо ж взяти shared «лочку», то exsclusive вже взяти не можна, но ще shared – можна. Суть полягає в тому, що якщо викликати метод який не міняє структуру даних (наприклад ми працюємо з масивом і викликаєм метод contains() ), то, наприклад, немає сенсу блокувати цей масив для іншого потока, який теж прцює з цією структурою даних не змінюючи її (наприклад, теж викликає метод contains() для визначення наявності елемента в масиві). Тобто, таке блокування дозволяє іншим методам «читати» дані, при цьому блокує доступ методам, які «пишуть» дані аби все працювало коректно.

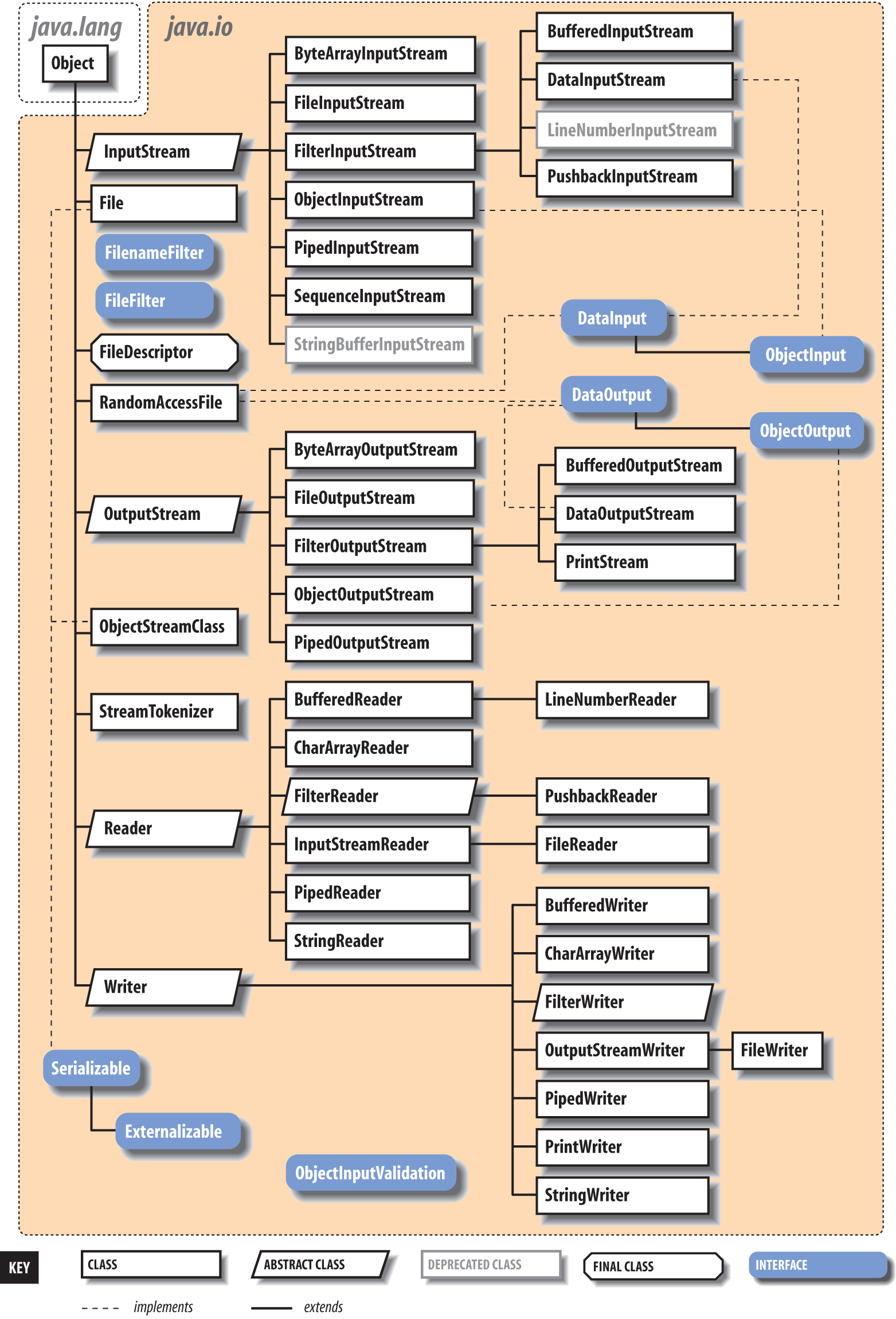
Одна з моделей: якщо є послідовність readlock – writelock - readlock –readlock, то writelock буде чекати, поки не «відпустяться» всі readlock і тоді почне виконуватися. А, можливо, все буде виконуватися попорядку -readlock – writelock - readlock –readlock. (це запитання для роздумів.) Приклад: програми для дефрагментації. Колишні блокували систему, сучасні роблять де фрагментацію в фоновому режимі.

Інтерфейс **ConcurentMap** – насілідується від **Map** але добавляє ще деякі методи.

Коли вичитується буфер даних в масив байтів, то при кількості отриманих даних меншій за розмір буфера (наприклад, файл дочитався до кінця) в кінці масиву байт (після точки -1) залишаються байти які були записані з попереднього читання. (н еобнуляються спеціально, а масив є immutable). Також в масив байтів може копіюватися кількість даних не рівна розміру буфера за причини влаштування ОС та носіїв даних (ХДД, інтернет тощо).







**SLAB** – **single layer of abstraction**. На одному рівні абстракції повинні бути конструкції такого ж рівня.

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.

BufferedInputStreamбуферизує 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.

**import** java.io.FileInputStream;  
**import** java.io.IOException;  
**import** java.io.InputStream;  
  
**public class** Template {  
 **public static void** main(String[] args) {  
 InputStream inputStream = **null**;  
 **try** {  
 inputStream = **new** FileInputStream(**"C://note.txt"**);  
 **int** counter;  
 **while** ((counter = inputStream.read()) != -1) {  
 System.***out***.print((**char**) counter);  
 }  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 } **finally** {  
 **if** (inputStream != **null**) {  
 **try** {  
 inputStream.close();  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
 }  
}

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);

**import** java.io.FileInputStream;  
**import** java.io.FileNotFoundException;  
**import** java.io.IOException;  
  
**public class** Template {  
 **public static void** main(String[] args) {  
 String filePath = **"C://note.txt"**;  
 **int** i;  
 FileInputStream fileInputStream = **null**;  
  
 **try** {  
 fileInputStream = **new** FileInputStream(filePath);  
 **do** {  
 i = fileInputStream.read();  
 **if** (i != -1) System.***out***.print((**char**) i);  
 } **while** (i != -1);  
  
 } **catch** (FileNotFoundException e) {  
 e.printStackTrace();  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 } **finally** {  
 **try**{  
 **if**(fileInputStream != **null**) fileInputStream.close();  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}

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);

**public class** Template {  
 **public static void** main(String[] args) {  
 String string = **"qwertyuiop"**;  
 **byte**[] strBytes = string.getBytes();  
 **int** symbol;  
  
 ByteArrayInputStream inputStream = **new** ByteArrayInputStream(strBytes);  
  
 **while**((symbol = inputStream.read()) != -1){  
 System.***out***.print((**char**) symbol);  
 }  
 }  
}

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);

**import** java.io.BufferedInputStream;  
**import** java.io.ByteArrayInputStream;  
**import** java.io.IOException;  
  
**public class** Template {  
 **public static void** main(String[] args) {  
 String string = **"abcdefgh"**;  
 **byte**[] byteArr = string.getBytes();  
 ByteArrayInputStream inputStream = **new** ByteArrayInputStream(byteArr);  
 BufferedInputStream bufferIS = **null**;  
 **int** symvol;  
  
 **try** {  
 bufferIS = **new** BufferedInputStream(inputStream);  
 **while** ((symvol = bufferIS.read()) != -1) {  
 System.***out***.print((**char**) symvol);  
 }  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 } **finally** {  
 **try** {  
 **if** (bufferIS != **null**)  
 bufferIS.close();  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
 }  
 }  
}

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. Приклад:

String s = **"if (a == b) x = y"**;  
**byte**[] stringByte = s.getBytes();  
ByteArrayInputStream byteArrInputStream = **new** ByteArrayInputStream(stringByte);  
**int** c;  
  
**try** (PushbackInputStream pushBack = **new** PushbackInputStream(byteArrInputStream)) {  
 **while** ((c = pushBack.read()) != -1) {  
 **switch** (c) {  
 **case '='**:  
 **if** ((c = pushBack.read()) == **'='**) {  
 System.***out***.print(**".eq"**);  
 } **else** {  
 System.***out***.print(**"<-"**);  
 pushBack.unread(c);  
 }  
 **break**;  
 **default**:  
 System.***out***.print((**char**) c);  
 **break**;  
 }  
 }  
} **catch** (IOException e) {  
 System.***out***.println(**"I/O Exceotion"**);  
}

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:

**import** java.io.FileInputStream;  
**import** java.io.FileNotFoundException;  
**import** java.io.FileOutputStream;  
**import** java.io.IOException;  
  
**public class** Template {  
 **public static void** main(String[] args) {  
 **int** symbol;  
 String inputFilePath = **"C://note.txt"**;  
 String outputFilePath = **"C://noteCopy.txt"**;  
 FileInputStream fileInputStream = **null**;  
 FileOutputStream fileOutputStream = **null**;  
  
 **try** {  
 fileInputStream = **new** FileInputStream(inputFilePath);  
 fileOutputStream = **new** FileOutputStream(outputFilePath);  
  
 **do** {  
 symbol = fileInputStream.read();  
 **if** (symbol != -1) fileOutputStream.write(symbol);  
 } **while** (symbol != -1);  
 } **catch** (FileNotFoundException e) {  
 e.printStackTrace();  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 } **finally** {  
   
 **try** {  
 **if** (fileInputStream != **null**)  
 fileInputStream.close();  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
  
 **try** {  
 **if** (fileOutputStream != **null**)  
 fileOutputStream.close();  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
  
 }  
 }  
}

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 **DataOutputStrem** (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.

**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();  
 }  
   
 }  
}

**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.)

Приклад:

**import** java.io.BufferedReader;  
**import** java.io.IOException;  
**import** java.io.InputStreamReader;  
  
**public class** Template {  
 **public static void** main(String[] args) **throws** IOException {  
 **char** c = 0;  
 BufferedReader bufferedReader = **new** BufferedReader(**new** InputStreamReader(System.***in***));  
  
 System.***out***.print(**"Enter \"q\" for exit."**);  
  
 **do** {  
 c = (**char**) bufferedReader.read();  
 System.***out***.println(c);  
 } **while** (c != **'q'**);  
 }  
}

Приклад примітивного текстового блокнота:

**import** java.io.BufferedReader;  
**import** java.io.IOException;  
**import** java.io.InputStreamReader;  
  
**public class** Template {  
 **public static void** main(String[] args) **throws** IOException {  
 BufferedReader bufferedReader = **new** BufferedReader(**new** InputStreamReader(System.***in***));  
  
 String[] strings = **new** String[100];  
 System.***out***.println(**"Type the lines. \"stop\" for finish."**);  
  
 **for** (**int** i = 0; i < 100; i++) {  
 strings[i] = bufferedReader.readLine();  
 **if** (strings[i].equals(**"stop"**)) **break**;  
 }  
  
 System.***out***.println(**"Your lines."**);  
 **for** (**int** i = 0; i < 100; i++) {  
 **if** (strings[i].equals(**"stop"**)) **break**;  
 System.***out***.println(strings[i]);  
 }  
 }  
}

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.

**import** java.io.FileNotFoundException;  
**import** java.io.FileReader;  
**import** java.io.IOException;  
  
**public class** Template {  
 **public static void** main(String[] args) {  
 **try** (FileReader fileReader = **new** FileReader(**"C://note.txt"**)) {  
 **int** symbol;  
  
 **while** ((symbol = fileReader.read()) != -1) {  
 System.***out***.print((**char**) symbol);  
 }  
 } **catch** (FileNotFoundException e) {  
 e.printStackTrace();  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
  
 }  
}

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

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

**import** java.io.FileWriter;  
**import** java.io.IOException;  
  
**public class** Template {  
 **public static void** main(String[] args) {  
 String string = **"Lorem ipsum dolor sit amet, consectetur adipiscing elit, "** +  
 **"sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. "** +  
 **"Ut enim ad minim veniam, "** +  
 **"quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat."**;  
  
 **char**[] buffer = **new char**[string.length()];  
 string.getChars(0, string.length(), buffer, 0);  
  
 **try** (FileWriter writer = **new** FileWriter(**"c://test.txt"**)) {  
 **for** (**int** i = 0; i < buffer.**length**; i++) {  
 writer.write(buffer[i]);  
 }  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
  
 }  
}

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

**DataInput**/**DataOutput** використовують для читання\запису даних формату Джава. (примітивних типів + String)

**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).

**FileInputStream**/**FileOutputStream** – створюють байтові потоки пов’язані з файлами. Throws FNFE.

Віртуальна машина може повноцінно працювати тільки з тими об’єктами, які знаходяться в її пам’яті. Коли мова заходить про файли чи доступ до інтернету, то ВМ відсилає запити до ОС. Тобто, якщо ми створили файл і пізніше прийшов GarbageCollector, видалив інстанс файлу в ВМ, то з файлом в файловій системі, який ми створили нічого не станеться.

**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.

Метод рекурсивного видалення папок і файлів з папки:

**public static void** delete(File root)**throws** IOE{  
 **if**(root.isDirectory()){  
 File files=root.listFiles();  
 **for**(File fileOne:files){  
 delete(fileOne);  
 }  
 }**else** delete(root);  
 }

**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.

**Абсолютний шлях файлу** – коли шлях до файлу записується повністю. Наприклад, для ссилки потрібно вказати протокол **http://www.uamedwed.com.**

**Відносний шлях файлу** – коли шлях вказується відносно файла чи директорії. Наприклад, якщо ми маємо каталог images в одній папці з сторінкою index.html,то для того аби вставити в html сторінку картинку з папки images не потрібно прописувати всю ієрархію каталогів від початку, достатньо вказати «images/pic.png» - це і є відносний шлях. Коли ж ми хочемо з каталога в якому перебуваємо вийти на один рівень ієрархії каталогів вверх, то пишемо «../someDir/images/pic.png»

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)

**Що б читати файл** з FileInputStream використовується метод **int read();** Кожного разу, коли викликається даний метод, він читає один байт і повертає його як ціле число. (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). Метод повертає -1, коли досягає закінчення файла. Метод throwsIOE.

**Для запису в файл** використовується метод **void write(int значенняБайта);** який визначено в класі FileOutputStream. Метод пише байт (з конструктора) в файл.

Видалення папок з використанням File (та й в загальному впринципі) є рекурсивним. Для видалення папки потрібно зайти в папку і перевірити на наявність в ній інших файлів і папок. Після цього видаляються файли, а в інші папки відбувається рекурсивне входження за таким же принципом.

При використанні InputStream не потрібно викликати метод close() в кінці, оскільки об’єкт, яким оперує InputStream (BAIS etc...) лежить в heap і з ним самостійно розбереться GC.

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 з 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**().

Приклад серіалізації і десеріалізації:

**import** java.io.\*;  
  
**public class** SerializationDemo {  
 **public static void** main(String[] args) {  
 *//writing an object* **try** (ObjectOutputStream outObject = **new** ObjectOutputStream(**new** FileOutputStream(**"c://serial"**))) {  
 MyClass objectToOut = **new** MyClass(**"Test"**, 7, 12.5);  
 System.***out***.println(**"object 1: "** + objectToOut);  
  
 outObject.writeObject(objectToOut);  
  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 }  
  
 *//reading an object* **try** (ObjectInputStream inObject = **new** ObjectInputStream(**new** FileInputStream(**"c://serial"**))) {  
 MyClass objectToIn = (MyClass) inObject.readObject();  
 System.***out***.println(**"object 2: "** + objectToIn);  
 } **catch** (FileNotFoundException e) {  
 e.printStackTrace();  
 } **catch** (IOException e) {  
 e.printStackTrace();  
 } **catch** (ClassNotFoundException e) {  
 e.printStackTrace();  
 }  
  
 }  
}

**import** java.io.Serializable;  
  
**public class** MyClass **implements** Serializable {  
 String **string**;  
 **int anInt**;  
 **double aDouble**;  
  
 **public** MyClass(String s, **int** i, **double** d) {  
 **string** = s;  
 **anInt** = i;  
 **aDouble** = d;  
 }  
  
 **public** String toString() {  
 **return "s = "** + **string** + **"; i = "** + **anInt** + **"; d = "** + **aDouble** + **";"**;  
 }  
}

**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.

**RMI** (**remote** **method** **invocation**). Загальна назва **RPC**. (**remote** **procedure** **call**). Можливість викликати методи в іншому адресному просторі. Як правило на інших ПК. Тобто з методів на одній машині викликати методи на іншій машині.

**Тонкий клієнт** – клієнт який за кожним запитом користувача посилає запити на сервер для отримання потрібної інформації.

**Товстий клієнт** – клієнт, який отримує достатню кількість даних аби працювати самому не посилаючи кожного разу запити на сервер.

**instanceof** – оператор, який дозволяє перевірити чи є заданий об’єкт типом чи наслідником заданого типу.Повертає істину або фолс.

class A{

int I, j;

}

psvm(String[] args){

A a = new A();

if( a instanceof A ) {SOP(“yes”) };

}

**assert** – застосовується на стадії розробки як правило. Допомагає визначити чи виконується якась умова під час виконання. Наприклад, чи повертає якесь значення невід’ємний результат. Якщо умова істинна – не відбувається нічого. Якщо значення хибне – отримуємо AssertionError.

assert n> 0; // AErr якщо n рівна 0 чи менше.

Щоб ввімкнути перевірку за допомогою аасерта потрібно вказати при запуску класа ключ **-ea:**

java –ea AssertDemo //enable assertion

java –da AssertDemo // disable aasertion

java –ea:MyPack… //включає для всього пакета

**Статичний імпорт** (staticimport) використовується для імпорта статичних членів класів чи інтерфейсів. При статичному імпорті з’являється можливість звертатися безпосередньо до самих статичних членів без необхідності кваліфікувати їх імена класів. Приклад:

hypot = Math.sqrt(Math.pow(sideOne, sideTwo));

import static java.lang.Math.sqrt;

import static java.lang.Math.pow;

hypot = sqrt(pow(sideOne, sideTwo));

**Статичний імпорт** – засіб за допомогою якого можна імпортувати методи класу чи окремий метод аби скоротити розмір коду і для зручності написання коду:

//до статичного імпорту

Math.sqrt(x);

System.out.println(x);

//після статичного імпорту

staticimportjava.lang.Math.sqrt;

static import java.lang.System.out.\*;

sqrt(x);

out.println(x);

В одному конструкторі не можна використовувати super() ithis() – оскільки кожен з них має бути першим в конструкторі.

**Поліморфізми в Джава 3**: методів(**overriding** через наслідування, **overloading** через об’явлення багатьох методів з однаковим іменем але різними типами аргументів), поліморфізм типів даних (Generics)

**Узагальнення (параметризовані типи).**Ще називають **параметричним поліморфізмом**. Дозволяють утворити класи, інтерфейси і методи де типи даних якими вони оперують вказуються в вигляді параметру. Працюють тільки з ссилочними типами.

publicclassGen<T> {

Tob;

Gen(To){

ob = o;

}

T getOb(){

returnob;

publicclass GenTest {

publicstaticvoid main(String[] args) {

Gen<Integer>iOB = new Gen<Integer>(25);

intvar = iOB.getOb();

System.*out*.println(var); //25

Також можливі декілька параметрів в узагальненнях:

publicclassGen<T, V> {

Tob;

V ob2;

Gen(T o, V o2){

ob = o;

ob2=o2

}

T getOb(){

returnob, ob2;

Можливі обмежені узагальнення у випадку, коли нам потрібно працювати з конкретними типами даних. Наприклад, з числовими виключаючи строки. Типи узагальнення можуть вказувати які класи вони розширюють чи інтерфейси наслідують. Вказується так звана верхня межа. Це дає доступ до методів класу предку і його наслідників з типу узагальнення.

public class Stats<T extends Number> {

T[] nums;

Stats(T[] array){

nums = array;

}

double average(){

double sum = 0.0;

for(int i = 0; i<nums.length; i++)

sum += nums[i].doubleValue();

return sum / nums.length;

}

}

public class NumberDemo {

public static void main(String[] args) {

Integer[] intArr = {1, 5, 6, 78};

Stats<Integer> intNums = new Stats<Integer>(intArr);

double result = intNums.average();

System.out.println(result + " : result.");

}

}

**Метасимвольний аргумент <?> -** представляє собою невідомий тип в узагальненнях, який співпадає з будь-яким типом класу яким він викликаний.

boolean sameAvg(Stats<?> ob){

if(average() == ob.average() )

return true;

else

return false;

}

}

Stats<?> співпадає з будь-яким об’єктом класу Stats.

По аналогії можливі метасимвольні аргументи з вказанням межі: Stats<?extendsNumbers>

Можна створювати **узагальнені методи** в неузагальнених класах.

static <T extends Comparable<T>, V extends T> boolean isIn(T x, V[] y){

for(int i=0; i<y.length; i++)

if(x.equals(y[i])) return true;

return false;

}

**if**(GenMethDemo.isIn(78, nums)) System.out.println("true");

Також можа явно задати типи узагальнення:

GenMethDemo.<Integer, Integer>isIn(78, nums)

При чому тип узагальнення в даному випадку оголошується до оголошення типу який повертає сам метод – це притаманне узагальненим методам. Тут Vобмежений верхньою межою типу Т. Це означає, що Vмає бути таким самим типом, шо й Т, або його нащадком. Такий взаємозвязок гарантує те, що метод isIn() буде викликаний з сумісними параметрами.Прам методи можуть бути як статичні так і нестатичні.

Можна створювати узагальнені конструктори:

public class GenCons {

double doubleVal;

<T extends Number>GenCons(T num){

doubleVal = num.doubleValue();

}

При чому параметри узагальнення оголошуються до імені конструктора.

При присвоєнні в конструкторі важливо першою ставити змінну даного класу і тоді її прирівнювати до змінної отриманої в конструкторі. Не навпаки. Інакше отримаємо помилку.

class MyClass<T extends Comparable<T>> implements MaxMin<T> {

T[] values;

MyClass(T[] val){

values = val;

}

Можна створити **узагальнені інтерфейси**за таким же принципопм, як і узагальнені класи:

interface MaxMin<T extends Comparable<T>> {

T max();

T min();

}

Потім реалізувати його в класі:

class MyClass<T extends Comparable<T>> implements MaxMin<T> {

T[] values;

MyClass(T[] val){

values = val;

}

@Override

public T max() {

T v = values[0];

for(int i = 1; i< values.length; i++)

if(values[i].compareTo(v) > 0) v = values[i];

return v;

}

@Override

public T min() {

T v = values[0];

for(int i = 1; i<values.length; i++)

if(values[i].compareTo(v) < 0) v = values[i];

return v;

}

}

Знову ж таки, в даному випадку, тип Т має обмеження зверху Comparable. При оголошенні класу MуClassв параметр ми передаємо йому такий же тип, який має інтерфейс MinMaxі вказуємо, що наш MyClassбудереалізовуватиінферфейсMinMax. Потрібно звернути увагу, що в узагальнення інтерфейса передається просто тип Т оголошений в MyClass, а не ціла строка як в оголошенні інтерфейса, оскільки вони ідентичні.

//good

class MyClass<T extends Comparable<T>> implements MaxMin<T>

//bad

class MyClass<T extends Comparable<T>> implements MaxMin<T extends Comparable<T>>

Як правило, клас який реалізує узагальнений інтерфейс, повинен також бути узагальненим. Приклад помилки:

//bad

classMyClassimplementsMaxMin<T>

Тут тип MyClassє неоголошеним, тому ми не можемо передати його в інтерфейс як тип Т. Але, якщо ми реалізуємо конкретний тип, то така конструкція допустима:

//good

classMyClassimplementsMaxMin<Integer>

Перевага аузагальнених інтерфейсів полягає в тому, що вони можуть бути реалізовані для різних типів даних. І забезпечують типову безпеку. До прикладу типом даних переданих інтерфейсу MinMaxможуть бути тільки наслідники Comparableі він сам.

Базові типи використовуються для переносу старого коду, який не вмів працювати з узагальненнями. До JDK5.

Для роботи з циклом foreach потрібно щоб наш клас реалізував інтерфейс Iterable. Тоді ми зможемо перебирати в даному циклі екземпляри нашого класу.

**Інтерфейс Iterable** має тільки один метод iterator, який повертає тип **Iterator**:

import java.util.Iterator;

public interface Iterable<T>{

Iterator<T> iterator();

}

**Інтерфейс Iterator** орієнтований тільки на читання і не має random access(чим схожий на InputStream), має 3 методи:

public interface Iterator<E>{

**booleanhasNext();** // перевіряє чи є ще елементи для «взяття»

**E next();** // повертає наступний елемент в черзі, якщо такий є після перевірки методом hasNext()

**void remove();** // майже ніколи не реалізується.

}

Пркилад:

while( iter.hasNext() ){

return iter.next();

}

Приклад організації класу Company з об’єктами Employee для роботи зциклом foreachпри використанні інтерфейсів Iterableі Iterator:

public classExampleIterable{

psvm(String[] args){

Company company = new Company();

Employee[] employees = (new Employee(“Sarah”), new Employee(“Mike”), new Employee(“John”));

for(Employee empl : company){

SOUT(“Eployee: ” + empl);

}

}

}

// Company реалізує інтерфейс Iterable, який є нащадком інтерфейса Іterator.

import java.util.Iterator;

public class Company implements Iterable<Employee>{

private Employee[] employees = new Employee[0];

public void setEmployees(Employee[] employees){

this.employees = employees;

}

//@Override. ОголошенийвIterableі повинен бути реалізований в класі

public Iterator<Emlpoyees> iterator(){

return new ArrayIterator(employees);

}}

// ArrayIterator - по суті це адаптер масива до ітератора

import java.util.Iterator;

public class ArrayIterator<T> implements Iterator<T>{

private final T[] data;

privete int index = 0;

public ArrayIterator(T[] data){

this.data = data;

}

@Override

public boolean hasNext() { return index<data.length; }

@Override

public T next() { return data[index++]; }

@Override

public void remove() { //NOP }

}

В Java 8 було добавлено ще один ітератор **– ітератор-розділювач.** Він забезпечує паралельну ітерацію. Цей ітератор підтримується інтерфейсом **Spliterator**і підтримує примітивні типи в тому числі (інтерфейси PrimitiveIterator, PrimitiveIterator.OfDouble).

Якщо суперклас є узагальненим, то всі предки також мають працювати з узагальненим типом даних суперкласа. Також, крім узагальненого типу суперкласа, наслідники можуть добавляти свої узагальнені типи до типів суперкласу.

class Gen<T>{…}

class Gen2<T, V> extends Gen<T> {…}

Можливі і неузагальнені суперкласи наслідникик яких є узагальненими:

class Gen{…}

class Gen2<T > extends Gen{…}

З Джава 7 синтаксис дозволяє не писати ще раз параметри типів в скобках після new.

// було

MyClass<Integer, String> mcl = new MyClass<Integer, String>(88, “Hello”);

// стало

MyClass<Integer, String> mcl = new MyClass<>(88, “Hello”);

**Стирання** – засіб, який дозволяє старому коду працювати з узагальненим кодом. Принцип дії його заключається в тім, що під час виконання коду ніяких відомостей про типи узагальнень не існує – при компіляції всі відомості про типи стираються. Тобто, спочатку всі параметри типу заміняються на обмеження в вигляді типу Object, якщо ніякого явного обмеження не вказано. Потім вже виконується приведення типів яке потребує узагальнення.

Ми не можемо перегружати методи по типам T, Vпоодиноко, оскільки в силу стирання типів під час виконання вони обидва приводяться до Object.

publicclassGen<T, V> {

…

voidsetOb(Tob);

voidsetOb(Vob); //не скомпіляяється

Неможливо створити екземпляр самого узагальненого типу.

publicclassGen<T, V> {

Tob;

ob2 = newT(); // помилка. Тип T неможе бути визначено

Статичні члени не можуть використовувати параметри типів.

staticTob; // помилка

Неможливо створити екземпляр узагальненого масиву.

T[] values; // допустимо

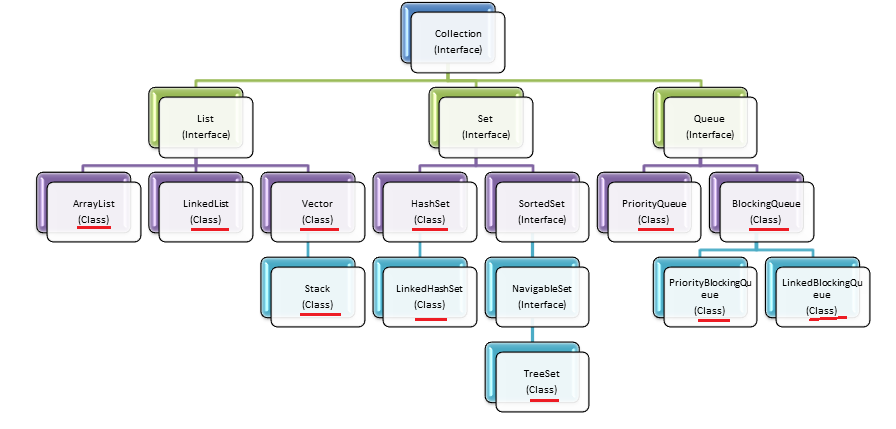
values = new T[10]; // неможна створити масив типу Т

Також неможливо оголосити масив ссилок конкретного узагальненоготипу. Потрібно використовувати метасимвол:

Gen<Integer> gen = new Gen<Integer>(10);

Gen<?> gen = new Gen<?>(10);

Узагальнення не можуть параметризувати виключення оскільки не розширюють клас Throwable.

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**.

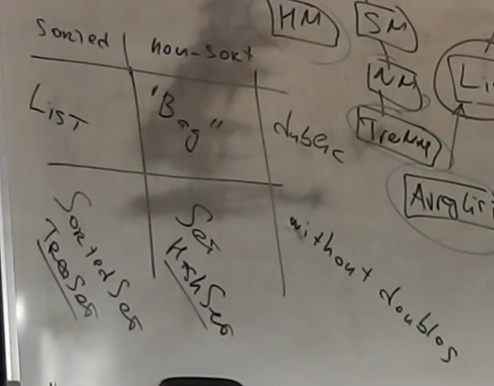
Виключення **CassCastException** виникає в тому випадку, коли ми хочемо добавити в колекцію об’єкт несумісного типу.

**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>extendsCollextion<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(Obkect obj)** – істина, якщо обєкт є в колекції.

**boolean containsAll(Collextion<?>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 неLinkedList. 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).

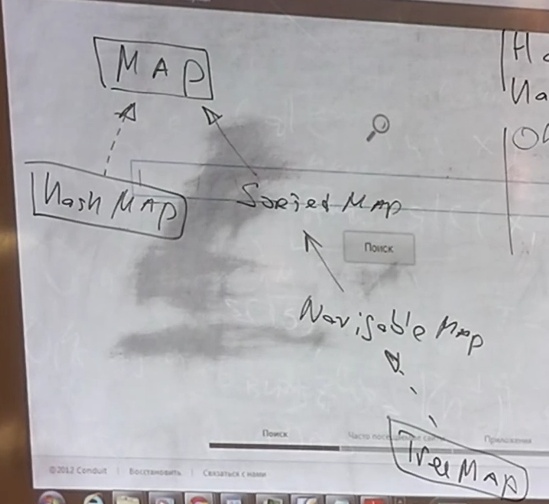
Якщо порівняти метод **remove()** для LinkedList і ArrayList, то отримаємо схожу картину – обидва відпрацюють відносно повільно. ArrayList швидко знайде потрібний індекс, видалить ячейку, але затратить час на копіювання і злиття частин результуючого масиву. В той час LinkedList повільно знайде індекс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

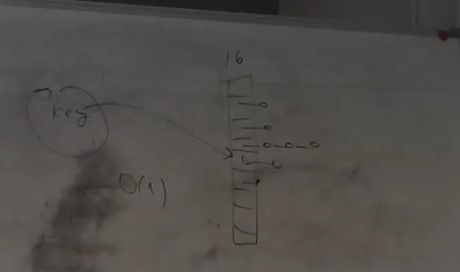
**get()**працює швидко в обох випадках на 0-му елементі. Одинаково на взятті з хвоста. При взятті елемента з середини LinkedList набагато повільніший, оскільки треба пробігтися по всім елементам до середини.

**contains()**працює в ArrayListнабагато 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 ArrayListробить новий масив і копіює значення, а LinkedList просто перекидає ссилку. В кінець (хвіст) – одинаково швидко. При вставці елемента в середину LinkedListповільніший, 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. 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.

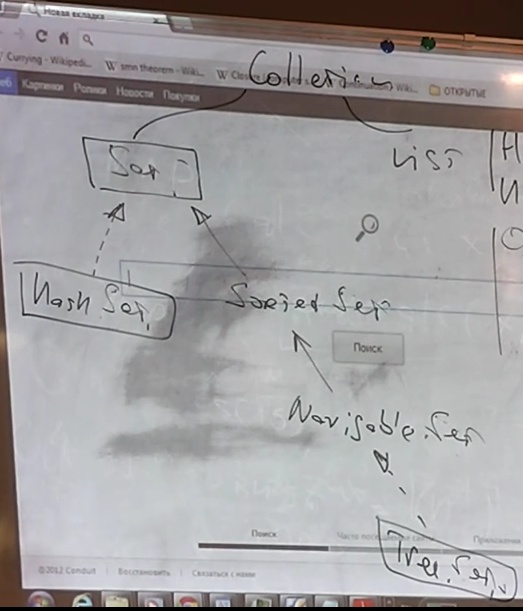
Методи Object:

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 ключі 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 ссилки, 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 Userз 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)? практичнояк і в TreeSetі 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)

**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 і сеттери, а не звертатися до полів напряму, оскільки ОРМ можуть використовувати lazy loading.

**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 колкцію 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 [ ]

**Клас PriorityQueue<E> -** розширяє клас AbstractQueue і реалізує інтерфейс Queue. Слугує для черги по пріоритетам влаштованій на заданому компараторі черги. Це динамічна черга, яка може мінятися при потребі. Рекомендовані методи для нього offer(), poll(). Конструктори:

PriorityQueue();// default size 11

PriorityQueue(int size);

PriorityQueue(int size, Comparator<? super E> comp);//якщо компаратор не //вказано, то береться дефолтний для типу даних які збкрігаються. //В голові буде найменший елемент.

PriorityQueue(Colection<? extends E> collection);

PriorityQueue(PriorityQueue<? extends E> collection);

PriorityQueue(SortedSet<? extends E> collection);

**Клас ArrayDeque<E>** - розширяє клас AbstractCollcetion і реалізує інтерфейс Deque. Створює динамічний масив, який не має обмежень по ємкості. Працює по принципу стекаю Конструктори:

ArrayDeque ();

ArrayDeque(int size);

ArrayDeque(Collection<? extends E> collection);

Приклад:

ArrayDeque<String> arrayDequeOfStrings = **new** ArrayDeque<String>();

arrayDequeOfStrings.push("First");

arrayDequeOfStrings.push("Third");

arrayDequeOfStrings.push("Second");

arrayDequeOfStrings.push("Fourth");

System.***out***.println("Taking from stack: ");

**while**(arrayDequeOfStrings.peek() != **null**){

System.***out***.println(arrayDequeOfStrings.pop() + " ");

}

**Клас EnumSet<E extends Enum<E>> -** розширяє клас AbstractSet і реалізує інтерфейс Set. Слугує для створення множини, яка розрахована для застосування разом з ключами типу перечислень enum. В даному класі конструктори не визначені. Замість них використовуються фабричні методи.

**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. Методи інтерфейсу:

**int characteristics()** - викликає характеристику ітератор яка описується цілим числом.

**long estimateSize()** - повертає кількість елементів, які ще треба перебрати. Якщо з деяких причин йому не вдається це дізнатися - повертає Long.MAX\_VALUE.

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

**default COmparator<? super E> getComparator();** - повертає компаратор який використову.ться ітератором-розділювачем. Повертає null якщо використовується звичайне впорядочення і кидає ISE якщо послідовність невпорядкована.

**default long getExactSizeIfKnown()** - повертає кількість елементів які потрібно перебрати, якщо попередньо було встановлено розмір викликаючого ітератора-розділювача. Інакше -1.

**default boolean hasCharacteristics()** - повертає істину, якщо у викликаючого ітератора-розділювача є характеристика яка задається в параметрі value.

**boolean tryAdvance(Consumer<? super E> action);** - виконує задану дію над наступним елементом в колекці. Повертає істину, якщо наступний елемент існує і фолс, якщо його немає.

**Spliterator<T> trySplit()** - розділяє, якщо є можливість, викликаючий ітератор-розділювач для посідуючого розділення, інакше пусте значення null. При успішному розділенні початковий ітератор-розділювач перебере одну частину послідовності, а новий ітератор розділювач - іншу.

Де Consumer - це функціональний інтерфейс, якій визначає дію над об’єктом. Визначений в java.util.function. В даному інтерфейсі визначається єдиний абстрактний метод void accept(T linkToObject)

ArrayList<Double> listOfDoubles = **new** ArrayList<>();

listOfDoubles.add(1.0);

listOfDoubles.add(2.5);

listOfDoubles.add(5.6);

listOfDoubles.add(5.7);

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

Spliterator<Double> split = listOfDoubles.spliterator();

**while**(split.tryAdvance((n) -> System.***out***.println(n)));

System.***out***.println();

split = listOfDoubles.spliterator();

ArrayList<Double> sqrtList = **new** ArrayList<>();

**while**(split.tryAdvance((n) -> sqrtList.add(Math.*sqrt*(n))));

split = listOfDoubles.spliterator();

split.forEachRemaining((n) -> System.***out***.println(n));

Приклад збереження об’єктів в колекціях:

**public** **class** AddressForMailListDemo {

**private** String name;

**private** String city;

**private** String street;

**private** String state;

**private** String code;

AddressForMailListDemo(String n, String c, String s, String st, String cd){

name = n;

city = c;

street = s;

state = st;

code = cd;

}

**public** String toString(){

**return** name + "\n" + street + "\n" + city + " " + code;

}

}

==============================================================================

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

**public** **class** MailList {

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

LinkedList<AddressForMailListDemo> mailList = **new** LinkedList<AddressForMailListDemo>();

mailList.add(**new** AddressForMailListDemo("Mike","NewYork", "Bronx", "Washington", "58013"));

**for**(AddressForMailListDemo element : mailList){

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

}

}

}

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.

**Відображення (мапа)** - обєкт який зберігає звязок між парами ключ-значення. По заданому ключу можна знайти його значення. Ключі і значення - також є обєктами. Ключі не можуть дублюватися, а значення можуть дублюватися. Є відображення в яких допускаються пусті значення чи пцсті ключі, а є відображення які це не допускають. Вони не реалізують інтерфейс Iterable тому ми не зможемо для його елементів організувати цикл for each чи отримати ітератор відображення. Але можливо получити представлення відображенняв вигляді колекції, яке допускає отримання ітератора і перебирання в кциклі for each.

Інтерфейси відображень:

**inteface Map<K, V>** - представляє однозначні ключі і їх значення.

**Map.Entry -** внутрішній клас інтерфейса Мар. Описує пару "ключ-значення".

**SortedMap** - розширює інтерфейс Map атким чином що би ключі впорядковувалия по зростанню.

**NavigableMap** - розширяє інтерфейс SortedMap для отримання елементів з відображення по критерію пошуку найбільшого співпадіння.

**inteface Map<K, V> -** представляє однозначні ключі і їх значення. Ключ - обєкт, який використовується для подальшого витягнення даних з мапи. Якщо зберегти значення по ключу, то по цьому ж ключу ми можемо отримати значення. Методи:

**void clear()** - видаляє всі пари ключ-значення.

**boolean containsKey(Object k)** - повертає істину, якщо в колекції є такий ключ.

**boolean containsValue(Object v)** - повертає істину, якщо в колекції є таке значення.

**Set<Map.Entry<K, V>> entrySet()** - повертає колекцію типу Set, яка містить всі записи з викликаючої колекції в вигляді об’єктів типу Map.Entry.

**boolean equals(Object o)** - повертає істину, якщо заданий об’єкт є мапою і містить таке значення.

**V get(Object k)** - повертає значеня по ключу. null - якщо даний ключ не знайдено.

**int hashCde()** - повертає хеш-код мапи.

**boolean isEmpty()** - повертає істину, якщо мапа порожня.

**Set<K> keySet()** - повертає колекцію, яка містить в собі ключі викликаючої колекції.

**V put(K k, V v)** - вводить значення v по ключу к. Перезаписує старе значення, якщо воно там було. Повертає null якщо даний ключ не існує, інакше повертає попереднє значення.

**void putAll(Map<? extends K, ? extends V> m)** - вводить всі записи в множини m в викликаючу множину.

**V remove(Object k)** - видаляє значення по ключу.

**default V replace(K k, V v)** - якщо існує ключ к, то по цьому ключу записується нове значення. Інакше повертає null.

**default boolean remove(Object k, Object v)** - повертає істину, якщо пара ключ-значення існує і видаляє її.

**default boolean remove(K k, V oldValue, V newValue)** - якщо пара ключ-значення існує, то це значення замінюється на нове і повертається істина.

**int size()** - повертає кількість пар ключ-значення.

**Collection<V> values()** - повертає колекцію, яка містить значення з викликаючого відображення.

**iterface SortedMap<K, V>** - розширає інтерфейс Мар. Забезпечує запис елементів в мапі по зростанню ключів. Методи:

**Comparator<? super K> comparator()** - викликає компаратор відсортованого відображення. Якщо в відображенні застосовується природнє сортування елементів, то повертає null.

**K firstKey()** - повертає перший ключ в колекції.

**K lastKey()** -повертає останній ключ в клдекції.

**SortedMap(K, V) headMap(K end)** - повертає відсортоване відображення, яке містить ті елементи з викликаючого відображення, ключі якого менші за вказаний кінець end.

**SortedMap(K, V) tailMap(K start)** - повертає відсортоване відображення, яке містить ті елементи з викликаючого відображення, ключі якого більші за вказаний початок start.

**SortedMap(K, V) subMap(K start, K end)** - повертає відображення, яке містить елементи викликаючого відображення, де ключі більші\рівні за початок і менші за кінець.

**interface NavigableMap<K, V>** - розширяє інтерфейс SortedMap. Використовується для отримання елементів по найбільш точному співпадінню ключа чи набору ключів. Методи:

**Map.Entry<K, V> ceilingEntry(K obj)** - виконує пошук найменшого ключа по критерію k >= obj. Повертає запис по ключу ,якщо його знайдено, інакше null.

**K ceilingKey()** - виконує пошук найменшого ключа по критерію k >= obj. Якщо найдено такий ключ, то його й повертається, інакше null.

**NavigableSet<K> descendingKeySet()** - повертає множину типу NavigableSet яке містить ключі викликаючою множини у зворотньому порядку. Тобто метод повертає зворотнє представлення ключів відображення в вигляді множини.

**NavigableMap <K, V> descendingMap()** - повертає множину, зворотню викликаючому відображенню.

**Map.Entry<K, V> firstEntry()** - повертає перший запис в відображенні. Він є найменшим по суті.

**Map.Entry<K, V> floorEntry(K obj)** - виконує пошук найбільшого ключа по критерію k <= obj. Якщо такий ключ знайдено - повертається запис по ньому, якщо ні, то null.

**K floorKey(K obj)** - виконує пошук найбільшого ключа по критерію k <= obj. Якщо такий ключ знайдено - повертається сам ключ, якщо ні, то null.

**NavigableMap<K, V> headMap(K верхГраниця, boolean включно)** - повертає множину, яка містить всі записи викликаючої множини по ключам, які менші за задану границю. Включно з заданою границею або без.

**Map.Entry<K, V>higherEntry(K obj)** - виконує пошук найбільшого ключа по критерію k > obj. Якщо такий ключ знайдено - повертається запис по ньому, якщо ні, то null.

**K higherKey(k obj)** - виконує пошук найбільшого ключа по критерію k > obj. Якщо такий ключ знайдено - повертається сам ключ, якщо ні, то null.

**Map.Entry<K, V> lastEntry()** - повертає останній запис в відображенні. По суті, це запис з найбільшим ключем.

**Map.Entry<K, V> lowerEntry(K obj)** - виконує пошук найбільшого ключа по критерію k < obj. Якщо такий ключ знайдено - повертається запис по ньому, якщо ні, то null.

**K lowerKey(K obj)** - виконує пошук найбільшого ключа по критерію k < obj. Якщо такий ключ знайдено - повертається сам ключ, якщо ні, то null.

**NavigableSet<K> navigableKeySet()** - повертає множину, яка містить ключі з викликаючого відображення.

**Map.Entry<K, V> pollFirstEntry()** - повертає перший запис в відображенні видаляючи його. По суті, це запис по найменшому ключу, оскільки відображенні відсортовано. null - якщо відображення порожнє.

**Map.Entry<K, V> pollLastEntry()** - повертає останній запис в відображенні видаляючи його. По суті, це запис по найбільшому ключу, оскільки відображенні відсортовано. null - якщо відображення порожнє.

**NavigableMap<K, V> tailMap(К нижнГраниця, bollean включно)** - повертає множину, яка містить всі записи викликаючої множини по ключам, які більші за задану границю. Включно з заданою границею або без.

**NavigableMap<K, V> subMap(K нижнГраниця, boolean включно, K верхГраниця, boolean включно)** - повертає відображення яке містить всі записи з викликаючого відображення по ключам меншим за верхняГраниця і більшим за ключі нижняГраниця. Включно саму верхні і нижню границю, або не включно.

**interface Map.Entry<K, V>** - інтерфейс дозволяє працювати з окремими записами в відображенні. Метод entrySet() який об’явлено в інтерфейсі Мар, повертає множину типу Set яка містить записи з відображення; кожний елемент цієї множини представляє собою об’єкт типу Map.Entry. Методи:

**boolean equals(Object obj)** - повертає істину, якщо заданий об’єкт представляє запис з відображення Map.Entry, ключі і значення в якому так ж, як і у викликаючого об’єкта.

**K getKey()** - повертає ключ даного запису з відображення.

**V getValue()** - повертає значення даного запису з відображення.

**int hashCode()** - повертає хеш-код даного запису з представлення.

**V setValue(V val)** - установлює значення в даний запис відображення. Якщо воно не відноситься до типу відображення - отримаємо CCE. Якщо значення вказано некоректно - IAE. Якщо значення пусте, а в даному відображення неможна зберігати пусті значення - отримаємо - NPE. Якщо в відображення неможна вносити зміни - отримаємо UOE.

Класи відображень (класи, які реалізують інтерфейси відображень):

**Клас AbstractMap** - варто зауважити, що він є суперкласом для всіх реалізацій відображень.

**Клас WeekHashMap** - реалізує відображення в якому реалізрвуються так звані "слабкі ключі", які ГК видаляє, якщо вони більше не використовуються.

**Клас HashMap<K, V>** - розширяє клас AbstractMap і реалізує інтерфейс Мар. В нім використовується хеш-таблицядля збереження відображень, завдяки цьому відбувається постійне виконання методів get() і put() навіть при звертанні до великих відображень. Контсруктори:

HashMap()

HashMap(Map<? extends K, ? extends V> m)

HashMap(int size) //default 16

HashMap(int size, float overloadValue) // default 16, 0.75

Не доповняє предків своїми методами. Не гарантує порядок збереження елементів (хоча ключі сортовані в принципі), тому результати ітератора можуть бути хибними.

Щоб добавити значення в відображення потрібно врахувати те, що нове значення стре старе по ключу і якщо нам потрібно старе значення, наприклад, змінити з урахуванням нового, то потрібно спочатку зберігти старе значення в змінну, а потім оперувати з нею. Приклад:

//put 1000 to John Dou balance

**double** balance = ownerBalance.get("John Dou");

ownerBalance.put("John Dou", balance + 1000);

System.***out***.println("John's new balance: " + ownerBalance.get("John Dou"));

**Клас TreeMap<K, V>** - розширяє клас AbstractMap і реалізує інтерфейс NavigableMap. В нім створюється відображення яке зберігає пари ключ-значення в відсортованій по ключах деревовидній структурі. Завдяки цьому відбувається швидкий доступ до вісортованих елементів. Конструктори:

TreeMap() //порожнє, сортоване природньо

TreeMap(Comparator<? super K > comparator) //порожнє, компаратором

TreeMap(Map<? extends K, ? extends V> m) //заповняється елементами, природньо сортоване

TreeMap(SortedMap<K, ? extends V> sm) // заповняється елементами, відсортовано так само як і в sm

Не визначає ніяких додаткових методів.

**Клас LinkedHashMap<K, V>** - розширяє клас HashMap. Створює связний список елементів в відображені в тому порядку, в якому вони були введені. Це дозволяє організувати ітерацію з вводом елементів в відображення в заданому порядку. Конструктори:

LinkedHashMap()

LinkedHashMap(int size)

LinkedHashMap(int size, float overloadFactor)

LinkedHashMap(int size, float overloadFactor, boolean order) //order по вводу чи останньому доступу

LinkedHashMap(Map<? extends K, ? extends V> m) // заповняється з m

До методів Мар добавляє тільки один метод:

**protected boolean removeEldestEntry(Map.Entry<K, V> e)** - викликається з методів put() і putAll. По замовчуванню нічого не робить і повертає фолс, але якщо його перевизначити, то можна видаляти найстаріший запис з відображення типу LinkedHashMap. Для цього сметод повинен повернути істину, а щоб зберегти самий старий запис з відображення - метод повниен повертати фолс.

**Клас IdentityHashMap<K, V>** - розширяє AbstractMap і реалізує інтерфейс Map. Аналогічний класу HashMap за виключення того, що при перевірці на рівність елементів у ньому відбувається порівняння ссилок на об’єктів на рівність. Не призначений для загального вживання.

**Клас EnumMap** - розширяє AbstractMap і реалізує інтерфейс Map. Спеціально призначиний для роботи з ключами типу enum. Не визначає своїх методів. Конструктори:

EnumMap(Class<K> keyType)

EnumMap(Map<K, ? extends V> m) //заповняється з m

EnumMap(EnumMap<K, ? extends V> em) //заповняється з em

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.приділяти його неприйнято.

В Джава 8 з’явилися нові методи по замовчуванню і статичні методи Компаратора:

**default Comparator<T> reversed()** - повертає зворотній компаратор сортування. (А,Б,С - буде С,Б,А)

**static<T extends Comparable<? super T>> Comparator<T> reverseOrder()** - повертає з зворотнього копаратора природній.

**static<T extends Comparable<? super T>> Comparator<T> naturalOrder()** - також повертає природній компаратор.

**static <T> Comparator<T> nullFirst(Comparator<? super T> comp)** - компаратор який обробляє значення null і вважає їх найменшими - відповідно ставить в початок колекції. Якщо дати компаратору на порівняння 2 null - сприйме їх як рівнозначні.

**static <T> Comparator<T> nullLast(Comparator<? super T> comp)** - компаратор який обробляє значення null і вважає їх найбільшими - відповідно ставить в кінець колекції. Якщо дати компаратору на порівняння 2 null - сприйме їх як рівнозначні.

**default Comparator<T> thenComparing(Comparator<? super T> secondComp)** - компаратор, який виконує повторне порівняння за допомогою іншого компаратор secondComp, якщо перше порівняння привело до рівності об’єктів. Відповідно, можна порівняти спочатку по X, а потім по Y. Наприклад, спочатку порівняти назви міст, а потім їх області.

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

}

}

}

З JDK8 можна переписати даний функціонал з допомогою реверсивного метода **reversed**() для природнього 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;

//natural comparing this time

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

}

}

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

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

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

//create comparator

MyComp mc = **new** MyComp();

//reversing natural comparator with method reversed();

TreeSet<String> ts = **new** TreeSet<String>(mc.reversed());

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

}

}

}

Можна взагалі видалити клас MyComp якщо використовувати лямбди:

TreeSet<String> ts = new TreeSet<String>( (aStr, bStr) -> bStr.compareTo(aStr) );

Приклад:

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

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

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

TreeSet<String> ts = **new** TreeSet<String>( (aStr, bStr) -> bStr.compareTo(aStr) );

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

}

}

}

Приклад сортування вкладчиків не по імені, а по прізвищі:

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

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

@Override

**public** **int** compare(String firstStr, String secondStr) {

**int** i, j, k;

//find symbol ' ' before second name

i = firstStr.lastIndexOf(' ');

j = secondStr.lastIndexOf(' ');

k = firstStr.substring(i).compareTo(secondStr.substring(j));

**if**(k == 0){ //check for second names are equals

**return** firstStr.compareTo(secondStr);

}**else**{

**return** k;

}

}

}

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

**public** **class** TreeMapDemo2 {

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

TreeMap<String, Double> tm = **new** TreeMap<String, Double>(**new** TComp());

tm.put("John Dou", **new** Double(145.23));

tm.put("Alex Teer", **new** Double(754.58));

tm.put("Man Kukli", **new** Double(785.423));

tm.put("Abydu Akaka", **new** Double(145.23));

//get the set from TreeMap

Set<Map.Entry<String, Double>> set = tm.entrySet();

//show the set

**for**(Map.Entry<String, Double> m : set){

System.***out***.print(m.getKey() + " ");

System.***out***.println(m.getValue());

}

System.***out***.println();

//put 1000 to Man Kukli balance

**double** balance = tm.get("Man Kukli");

tm.put("Man Kukli", balance + 1000);

System.***out***.println("New Man Kukli's balance: " + tm.get("Man Kukli"));

}

}

**Клас Arrays** такождопомагає працювати з масивами і колекціями сумісно.

**static <T> List asList(T arr)** – повертає список з масиву. Іншими словами, масив і список будуть ссилатися на одне місце в оперативній пам’яті.

**static int binarySearch(byte/short/int…etc[] arr, byte/short/int…etc value)** – метод шукає значення value в масиві arr методом половинного ділення. Застосовується до відсортованих колекцій. Повертає індекс елемента, якщо не знайдено -1.

**static byte/short/int… copyOf(byte[] sourceArr, int size)** – копіює масив sourceArr і задає йому велечину size. Урізає копію, якщо size < sourceArr.length і присвоє автоматичні значення залишку копії, якщо size > sourceArr.length. Якщо вказати від’ємну величину масива – отримаємо ексепшн NegativeArraySizeException.

**static byte/short/int… copyOfRage(byte[] sourceArr, int startIndex, int endIndex) -** робить копію масива вказуючи інтервал (від початок до кінець-1) елементів які треба скопіювати. Якщо вказати "початок" ві’ємний чи більший за величину масива - отримаємо ArrayIndexOutOfBounsException.

**static boolean deepEquals(Object[] a, Object[] b)** - перевіряє на рівність масиви, які мають вкладені в собі масиви.

static void sort(Object[] arr) - впорядковує елементи масива по наростаючій.

static void parallelSort(byte/char/double... etc [] arr) - сортує паралельно частини масива і потім зливає ці відсортовані частини.

**ООП.**

**4 основних принципи ООП:** поліморфізм, наслідування, інкапсуляція і абстракція (якщо її враховують).

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, потом 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 – забиваютьсядефолтними 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).

**Поліморфізм** - властивість використовувати один функціонал до змінних різних типів. В Джаві є 3 поліморфізма (3 елемнти мови, які дають 3 різних поліморфізма). Overloading(не є чистим поліморфізмом, оскільки треба писати різний код для різних параметрів "одного" метода), Overriding (тільик він робиться через наслідування), Generics. Розрізняють поліморфізм методів (overloading, overriding) і поліморфізм даних (generics).

**Overloading** - перегрузка метода чи конструктора для використання його з різними параметрами. В Джава можна мати декілька методів з одинаковим іменем, якщо його параметри є достатньо різними. (int - int[] розрізняються, List<String> - List<Integer> - не розрізняються). Приклад перегрузки:

printAsBinary(100); //перший метод для int

printAsBinary(new int[]{100. 101. 102}); // перегружений метод для масива int

Перегрузка відбувається в момент компіляції на відміну від перевизначення, тому, якщо ми маємо перегружені методи для різних об’єктів і хочемо ці об’єкти, наприклад, засунусти в масив Object чи колекцію і пройтися по них for eachом, то компілятор не зможе визначити який з методів вибрати для конкретного типу елементу масива чи колекції, оскільки на стадії компіляції ВМ не можемо знати наперед, який з методів потрібно вибрати. Це стане відомо тільки в Runtime, але компілятор не дасть нам дойти до цієї стадії виконання за рахунок вищесказаного.

Ми не можемо мати 2 методи з одинаковою сигнатурою (іменем і параметрами). Навіть якщо атрибути різні:

public int print() { }

pulic boolean print() { } //помилка, бо метод з сигнатурою print() вже визначено і компілятор не знає який з них викликати при виклику метода.

Навіть якщо ми вкажемо змінні,компілятор всеодно не знатиме, який з методів портіюно запускати:

public void print() { }

pulic boolean print() { }

….

int x = print();

boolean y = print();

При перегрузці методів, яки приймають переметри де може бути предок і нащадок компілятором вивбрається за рахунок динамічної диспетчеризації метод де тип є найближчий по ієрархії вверх до заданого типу-параметру. Тобто вибирає самий точний тип для себе, а не самий загальний для всіх. Наприклад:

public class OverloadTest{

public void print(Collection x) { }

public void print(List x) { }

public void print(Set x) { }

public void test(new ArrayList() ) { } // перегрузкою буде видбрано метод з List x, оскільки він є найближчим предком по ієрархії.

}

По замовчуванню літерали з плавоючою точкою в Джава вважаються double. x = 1,1.

По замовчуванню цілочисельні літерали в Джава вважаються int. x = 1. Тобто, якщо є 2 методи method(short x) і method(byte x), а в тілі програми викликати метод з літералом method(1), то програма не скомпілюється бо немає методу, який прийматиме літерал int який є по замовчуванню. А звузити цілочисельний тип в байтовий чи шорт автоматично ВМ не може. method(1), також може обробити метод method(Object obj), оскільки через автоупаковку int буде приведено до Integer, а той є нащадком Object.

int x = 5;

int y = 4;

int z = 5+4; // скомпілюється.

Integer x = 5;

Integer y = 4;

Integer z = x + y; // скомпілюється також за рахунок автоупаковки і авторозпаковки типів.

**Overriding** - перезапис метода класа предка від якого ми унаслідовалися або інтерфейса, який ми розширили. Тобто, якщо у нас є об’єкт, який працює з методом предка, то цей об’єкт може коректно працювати і з заоверрайдженим методом класа наслідника. Перевизначення відбувається в Runtime на відміну від перегрузки. Аннотація @Override.

Якщо у предка є метод f(), а у класа наслідника є метод g(), то по факту у метода наслідника є 2 метода f() і g().

Приклад:

public static void print(Object obj){

String str = (arg == null) ? "null" : arg.toString();

SOP(str);

}

Конструкція List lst = new ArrayList(); розбирається як: lst - це ссилка типу List на об’єкт типу ArrayList. Де ссилка List знаходиться в стеку, а об’єкт типу ArrayList знаходиться в хіпі. Відповідно об’єкт в хіпі є незмінним, а тип ссилки на об’єкт може змінюватися - Collection cc = lst; Тепер ми матимемо в стеку ссилку типу Collection на все той же ArrayList в хіпі. Тепер різниця між перегрузкою і перевизначенням (overloading vs overriding) - при оверлоадінгу ми працюємо з типом ссилки (List), а при оверрайдінгу ми працюємо з типом об’єкта в хіпі (ArrayList).

**Generics** - забезпечують поліморфізм типів данних. Приклад: ArrayList<String> чи ArrayList<Integer> etc.

Ідентифікатори доступу можуть бути:

**public** – видно в пакеті, видно в нащадку і в всіх решту місцях.

**private** – не видно в пакеті, не видно в нащадку.

**protected** – видно в пакеті, видно в нащадку.

**default** – коли ми явно не вказуємо ідентифіктор доступу. Являється package private. видно в пакеті, видно в нащадку.

Якщо ми унаслідуємося від предка, який лежить в іншому пакеті, то ми не можемо бачити його приватних полів.

**Длегірування** – коли ми не наслідуємося від предка, а просто робимо всій клас, в якому використовуємо функціонал предка напряму(робимо поле типу як у предка і через це поле звертаємся до предка). Тобто ми делегуємо запити через себе від класа який нас використовує, до класа, який має в собі потрібний функціонал.

**public** **class** DelegateList{

//створюєм приватне поле замість extends ArrayList.

**private** Collections<Integer> impl = **new** ArrayList<>();

// прийдеться вручну добавити і делегувати методи які нам потрібні з ArrayList. Сучасні IDE допомагають це автоматизувати.

**public** **void** add(**int** arg){

impl.add(arg);

}

// створюємо свій потрібний метод

**public** **int** delta(){

//**TODO**

}

}

Коли запитують про різницю між делегуванням і наслідуванням, то зачасту це методи вирішення однієї проблеми.

За допомогою статичних полів ми миожемо звертатися до них з інших класів через крапку.

class X{

static int yVal;

int zValue;

}

то ми можем звернутися до стаитчного поля напряму X.yVal; .щоб звернутися до нестатичного прийдеться створювати екземпляр класа - new X().zValue; . Якщо створювати інстанс класу ,як в другому не статичному випадку, то можна таким же чином звернутися і до статичного поля new X().yVal; .

Статичного поля завжди існує тільки один екземпляр. Це поле буде одним для всіх екземплярів класу де це поле знаходиться. Приклад:

**public** **class** StaticFieldDemo {

**public** **int** a;

**public** **static** **int** *b*;

**public** **int** c;

StaticFieldDemo(){

a++;

*b*++;

c = *b*;

}

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

StaticFieldDemo first = **new** StaticFieldDemo();

System.***out***.println("a=" + first.a + " " + "b=" + first.*b* + " " + "c=" + first.c);

StaticFieldDemo second = **new** StaticFieldDemo();

System.***out***.println("a=" + second.a + " " + "b=" + second.*b* + " " + "c=" + second.c);

StaticFieldDemo third = **new** StaticFieldDemo();

System.***out***.println("a=" + third.a + " " + "b=" + third.*b* + " " + "c=" + third.c);

StaticFieldDemo fourth = **new** StaticFieldDemo();

System.***out***.println("a=" + fourth.a + " " + "b=" + fourth.*b* + " " + "c=" + fourth.c);

}

}

Результат:  
a=1 b=1 c=1

a=1 b=2 c=2

a=1 b=3 c=3

a=1 b=4 c=4

Тобто, при створенні нового екземпляру, кожного разу змінна а обнулялася і їй присвоювалося дефолтне значення 0, оскільки вона є полем класу до конструктора. Статична змінна b в даному випадку є полем класу StaticFieldDemo, зберігається в нім і буде єдиною для всіх нових екземплярів в main оскільки її значення буде постійно зберігатися в класі, а не буде обнулятися, як у змінної а, яка не є статичною. Змінній с просто присвоюється значення статичної змінної b.

Статичні методи можуть читати тільки статичні поля і викликати тільки статичні методи. А з нестатичних методів можна викликати статичні і нестатичні методи і поля. Статичний метод не може викликати нестатичний меод, тому що статичний метод можна викликати навіть не створюючи екземпляр його класу ( X.main() ). Тобто можлива ситуація, коли у нас нема жодного інстансу класу, або цих інстансів декілька - в обох випадках компілятор не буде знати до якого інстансу чи поля звертатися. Теоретично це зумовлено тим, що створили статичні поля, які слугують спільними для всіх інстансів, а так, як інстансу класа може не бути, або інстансів може бути декілька, то ввели і статичні методи, які можуть викликиати тільки статичні методи і поля в силу вищесказаного.

**Процес** – програма ,яка виконується і має свій адресний простір.

**Потік** – послідовність команд, які виконуються процесором.

**Синхронізація** **потоків** – процес керування виконання потоків для створення багатопотокового ПЗ.

Як правило, одна прорама – це один або декілька процесів. В однім процесі може бути запущено багато птоків.

Якщо 2 потока будуть використовувати однй й тк ж несинроніховану секцію, то, наприклад, один з потоків або й два будуть мати в собі невірні дані з цього окду оскільки вони ж 2 несинхронізовано міняли поля.

**Головний** **потік** – потік з якого запускається програма. Наприклад, потік main.

**Стани** **потоку**: створений, запущений, блокований, зупинений.

**interrupt**() – встановлює значення потоку в «зупинений, перерваний». **interrupted**() – перевіряє чи потік перервано і скидає внутрішній флаг в фолс. **isInterrupted**() – перевіряє чи потік було перервано але не змінює внтурішній флаг.

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.

Потоки вводу і виводу бувають **байтові** і **симольні**.

Всі потоки є потомками суперкласів java.io.InputStream (байтовий), java.io.OutputStream (байтовий), java.io.Reader (символьний), java.io.Writer (символьний).

**Клас java.io.RandomAccessFile** – входить в склад API але не входить в ієрархію потоків, а наслідується прямо від Object. Викристовується для роботи з файлами які допускають довільний доступ до файлу. Можна задати режими доступу до файлу: r – read, rw – read/write.

Більшість пакетів лежать в java.io, класи для роботи з новою системою вводу\виводу лежать в java.nio, пакети для роботи з архівами лежать в java.util.

Надстройка – наділення існуючого потоку новими властивостями. Наприклад, BufferedOutputStream, FilterInputStream, DataInputStream.

Щоб вибрати файли по якомусь критерію, наприклад, розширенням – потрібно в методі listFiles() класа File прийняти об’єкт FileFilter в якому ми вкажемо файли якого типу ми виберемо по критерію.

Для архівування використовують класи GZIPOutputStream\GZIPInputStream, ZipOutputStream\ZipInputStream, DeflaterOutputStream\InflaterInputStream.

В Java 8 відмовились від Permanent Generation. Але, грубо кажучи, він є, тільки тепер називається Metaspace і теж може кидати OOME.

Колекції можуть зберігати будь-які ссилочнітипи даних.

Реалізації List: ArrayList, LinkedList, Vector(depreceted).

Ркалізації Set (множина без дублікатів): HashSet (хеш таблиця), TreeSet (бінарне дерево).

Якщо в Map записати 2 значення з одинаковим ключом, то буде записано тільки останнє значення. (відбувається перезапис)

Колекції дають змогу сортува, шукати, копіювати, отримувати синхронізовані колекції а також колекції достіпні тільки для читання.

Різниця HashMap i HashTable (і-с Мар): деякі методи HashTable синхронізовані, але цей клас є застарілим і нерекомендованим для застосування.

HashMap може тільки сказати об’єкт який нам дають є шуканим об’єктом. Більше ніяких бізнесових функій вона не виконує.

TreeMap має компаратор і може порівняти 2 об’єкти взаємодіючи з їх полями.

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.

Map не наслідується від Collections оскільки вони не сумісні бо були створені длярізних структур даних.

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.

Параметризовані класи ще називають шаблонизованими.

**Трьохтировий (трьохлеєрний) додаток** – загальноприйнятий принцип побудови програми для невеликого ентерпрайз додатку, який будується на 3 рівнях: **UI**\Interface Layer, **Domain**\Bussiness Logic Layer (дотримується POJO), **DAO**\DB Layer(Percistanse Layer). Проект додатку так і робивається на 3 основних пакети: UI, Domain, DAO.

**CRUD** – create, read, update, delete. 4 базових функції які виконуються в персистентних системах даних. Приклад на SQL: create (INSERT), read (SELECT), update (UPDATE), delete (DELETE).



java.lang.NullPointerException.

Слова Null і NULL компілятор не пропустить.

null - значення по замовчуванню будь-якого ссилочного типу. Не являється ні типом, ні об’єктом.

String myStr = (String) null; - скомпілюється нормально без ексепшенів.

int i = null; // type mismatch: cannot convert from null to int - помилка. Не можна приводити до примітивних типів. Можна присвоїти null типу обгортці, а потім привести її до примітива (null -> Integer -> int). Але в процесі виконання виникне NPE в наслідок автоупаковки.

Integer iAmNull = null;

int i = iAmNull; //компіляція пройде успішно, але тільки ми запустимо код - отримаємо NPE (). Таке часто трапляється при роботі з HashMap коли ключом є Integer.

import java.util.HashMap;

import java.util.Map;

public class Test {

    public static void main(String args[]) throws InterruptedException {

        Map numberAndCount = new HashMap<>();

        int[] numbers = {3, 5, 7, 9, 11, 13, 17, 19, 2, 3, 5, 33, 12, 5};

        for (int i : numbers) {

            int count = numberAndCount.get(i);

            numberAndCount.put(i, count++); // NullPointerException

        }

    }

}

Вивід:

|  |  |
| --- | --- |
|  | Exception in thread "main" java.lang.NullPointerException      at Test.main(Test.java:14)  Этот код выглядит простым и понятным. Мы ищем, сколько каждое число встречается в массиве, это классический способ поиска дубликатов в массиве в Java. Мы берем предыдущее значение количества, инкрементируем его и кладем обратно в HashMap. Мы полагаем, что Integer позаботится о том, чтобы вернуть значение по умолчанию дляint, однако если числа нет в HashMap, метод [get()](http://java67.blogspot.sg/2013/06/how-get-method-of-hashmap-or-hashtable-works-internally.html) вернет null, а не 0. И при оборачивании выбросит NullPoinerException. Представьте, что этот код завернут в условие и недостаточно протестирован. Как только вы его запустите на продакшен – УПС!  if (**null == null**) {//TODO} - поверне **true**. |
|  |  |

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

Для класу User при записі поля віку краще використовувати оболочку інта, оскільки він має додаткове значення null, що може вказувати на те, що вік не було введено.

Object obj = new ArrayList<>();

String str = obj.toString();

SOP(str); // [], бо викликається метод toString у ArrayList, а не Object. Оскільки при лверрайдинзі метод вибирається в рантаймі.

Колекції були добавлені в другій версії Джава. 1.2 До них використовувалися Dictionary, Srack, Vector і Properties. Але їм не вистачало узагальненості і достатньої взаємодії між собою.

В JDK 8 було добавлено новий вид ітератора - інтерфейс **Spliteator**. Він відрізняється від звичайного Iterator тим, що є вміє ітерувати паралельно. Його ще називають ітератором-розділителем.

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.

При використанні рефлексії можна викликати приватні і протектед методи. (method.setAccsessible(true) ) (как вызвать приватный метод в Джава)

Спочатку ініціалізується блок static, а потім керування передається блоку main.

**public** **class** Main {   
  {   
    System.out.print("1 ");   
  }   
   
  Main(){   
    System.out.print("2 ");   
  }   
   
  **public** **static** **void** main(String[] args) {   
    System.out.print("3 ");   
    Main m = **new** Main();   
    System.out.print("4 ");   
  }   
   
  **static** {   
    System.out.print("5 ");   
  }   
}

// 5 3 1 2 4

int в String конвертуватися не може. Потрібно використовувати valueOf();

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.

String testString = "This is a test string.";   
   String[] resultArray = testString.split(" ");   
   System.out.println(resultArray.length); // 5 спліт розріза в строку на підстроки різавшт по пробілу

Конструктори можуть бути тільки*public, protected, private.*

Публічний клас можна скомпілюватичерез командну строку навіть якщо він пустий чи закоментований.

При декомпозиції предметної області варто виконувати тільки ті задачі які поставив нам ПМ чи замовник. Не варто добавляти свого функціоналу який буде корисним на вашу думку. Тобто не виходити за scope.

**XML**. Способи читання мовами програмування: DOM(DocumenObjectModel. Зчитує документ повністю, а потім вже можна з ним працювати), SAX(SimpleAPItoXML. Зчитує документ поступово і ми можемо зупинити зчитування в будь-який момент), StAX(StreamingAPItoXML. Схожий до SAX лише зчитує файл дискретно порціями)

**XPath** – мова запитів XML. Не має формату.

**XSLT**, **xQuery** – мови, за допомогою яких з документа XML можна витягнути сегмент даних і зберегти його в новий документ XML. (або інший формат. По замовчуванню формат є XML)

**XMLSchema** – мова опису структури XML документа. Визначає правила поведінки документа. Розроблена для використання при роботі з XML документами при розробці ПО (шляхом створення моделі даних документа). Формат **.xsd**.

**JAXP** – загальна назва для всіх класів Джава, які працюють з XML. Включаэ в себе DOM, SAX, StAX

**ORM**– технологія програмування яка пов’язує бази даних з ОО моделлю програмування створюючи віртуальну об’єкту базу даних.(інструментом є Hibernate).

**JAXB–**дозволяє ставити у відповідність класи і XML представлення.Можна анотаціями в класі показати які теги мають бути згенеровані в XML документі.

Абстрактний клас і інтерфейс. Інтерфейс вирішує проблему кількісногонаслідування. Тобто якщо у нас є клас який реалізує потік засобом extendsThread і в подальшому є ймовірність, що цей клас треба буде розширити іншим, то виникне проблема адже наслідувати можна тільки один клас. А інтерфейсів можна наслідувативеликукількість, тому при потоках і реалізується інтерфейс Runnable, що звільняє місце для доповненя класу іншим класом.

**Поліморфізм** – властивість куска коду працювати з даними різного типу. 1-й тип поліморфізму - Поліморфізм який оснований на наслідуванні – якщо ми написали код який працює з класом, то ми можемо цим кодом працювати з будь-яким нащадком цього класу. 2-й тип поліморфізму – generic.

**POJO** ( **PlainOldJavaObject**) – простий джава об’єкт в старому стилі. Не унаслідуванийвід якихось специфічних класів і не реалізуючий ніяких службових інтерфейсів більше тих, які потрібні для бізнес логіки. Вони сконцентровані на своїй предметній області. Концептуально прості і зручні для тестування.

Слово abstract вживається в оголошенні методів без реалізації. Коли ми об’являємо інтерфейс ми можемо писати abstract, а можемо не писати – це одне й те саме.

**TDD - (test-driven development, пов’язане з KISS)** - техніка розробки програмного забезпечення, яка заключається в попередньому написанні автоматизованих модульних тестів до коду, а потім саого коду, який повинен дати потрібний результат, потім вже проводиться рефакторинг коду і так по мікроциклах. Темт містить умови, які або виконуються, або ні. Кент Бек, засновник методології, вважає її надійною і яка дає ясний результат.

**YAGNI (you ain't gonna need it - тобі це не знадобиться)** - принцип розробки ПО, який каже, що потрібно відмовлятися від надлишкової функціональності і речей, які в яких нема безпосередньої потреба.

**DRY (don repeat yourself або single sours of thruth)** - методологія розробки ПО, яка націлена на зниження дублювання коду особливо в багатослойовий асбтрактних рішеннях. Тобто кожна функція повинна мати свою одну дію і не робити нічого зайвого, що в неї не мало би входити. Це допомагає не робити змін по всьому проекту, коли ми щось міняємо в одному модулі. Порушення принципу DRY називають **WET (write everything twice)**.

**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.

**Metaprogramming** - вид програмування, пов’язаний з створенням програм, які породжують (генерують) нові програми в ході своєї роботи (наприклад, на стадії компіляції вихідного коду), або програм, які міняють себе по ходу виконання (наприклад ВМ). Це дозволяє зменшити затрати по розробці нових програм і покращує швидкодію і розмір коду, який буде на виході.

**Інтроспекція** - об’явлення внутрішніх структур мови в вигляді змінних встроєного типу з можливістю доступу до них з програми.

**Bootstrap Protocol (BOOTP)** - протокол мережевого рівня, призначений для роздавання ІР адреси пристроям по їх МАС адресам в момент їх загрузки.