Объектно-ориентированное программирование. Основные понятия.

Объектно-ориентированное программирование (ООП) — методология программирования, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования.

Абстракция данных: Абстрагирование означает выделение значимой информации и исключение из рассмотрения незначимой. В ООП рассматривают лишь абстракцию данных (нередко называя её просто «абстракцией»), подразумевая набор значимых характеристик объекта, доступный остальной программе.

Инкапсуляция — свойство языка программирования, позволяющее пользователю не задумываться о сложности реализации используемого программного компонента, а взаимодействовать с ним посредством предоставляемого интерфейса.

Наследование — свойство системы, позволяющее описать новый класс на основе существующего с частично или полностью заимствующейся функциональностью. Класс, от которого производится наследование, называется базовым, родительским или суперклассом. Новый класс — потомком, наследником, дочерним или производным классом.

Полиморфизм — это возможность применения одноименных методов с одинаковыми или различными наборами параметров в одном классе или в группе классов, связанных отношением наследования.

Например:

- создавать "одноименные методы" в одном классе ("перегрузка методов")
- или изменить поведение методов родительского класса ("переопределение методов").

Класс — универсальный, комплексный тип данных, состоящий из тематически единого набора «полей» (переменных более элементарных типов) и «методов» (функций для работы с этими полями), то есть он является моделью информационной сущности с внутренним и внешним интерфейсами для оперирования своим содержимым (значениями полей). В частности, в классах широко используются специальные блоки из одного или чаще двух спаренных методов, отвечающих за элементарные операции с определённым полем (интерфейс присваивания и считывания значения), которые имитируют непосредственный доступ к полю. Эти блоки называются «свойствами» и почти совпадают по конкретному имени со своим полем (например, имя поля может начинаться со строчной, а имя свойства — с заглавной буквы). Другим проявлением интерфейсной природы класса является то, что при копировании соответствующей переменной через присваивание, копируется только интерфейс, но не сами данные, то есть класс — ссылочный тип данных. Переменная-объект, относящаяся к заданному классом типу, называется экземпляром этого класса. При этом в некоторых исполняющих системах класс также может представляться некоторым объектом при выполнении программы посредством динамической идентификации типа данных. Обычно классы разрабатывают таким образом, чтобы обеспечить отвечающие природе объекта и решаемой задаче целостность данных объекта, а также удобный и простой интерфейс. В свою очередь, целостность предметной области объектов и их интерфейсов, а также удобство их проектирования, обеспечивается наследованием.

Объект - сущность в адресном пространстве вычислительной системы, появляющаяся при создании экземпляра класса (например, после запуска результатов компиляции и связывания исходного кода на выполнение).

Состав пакета java.lang. Класс Object и его методы.

Пакет *java.lang* является наиболее важным из всех пакетов, входящих в Java API, поскольку включает классы, составляющие основу для всех других классов. Каждый класс в Java неявным образом импортирует все классы данного пакета, поэтому данный пакет можно не импортировать.

В этой библиотеке включены:

- Object - базовый класс
- Class - объект описания класса (поля, методы)
- Math - набор статических методов для математических вычислений
- Классы-обёртки для примитивных типов:
    - Integer
    - Character
    - Boolean
    - Void
- Классы для работы с текстом
    - String
    - StringBuffer
- Системные классы:
    - ClassLoader
    - SecurityManager
    - System
    - Runtime
    - Process
- Потоки исполнения:
    - Runnable
    - Thread
    - ThreadGroup
- Классы работы с исключительными ситуациями:
    - Error
    - Exception
    - RuntimeException

Основу пакета составляет класс Object, который является корневым в иерархии классов. Если при описании класса родитель не указывается, то им считается класс Object. Все объекты, включая массивы, наследуются от этого класса. Класс Object включает методы, которые наследуются остальными классами Java.

Object clone() – функция создания нового объекта, копии существующего.

boolean equals(Object object) – функция определения равенства текущего объекта другому. В классах - наследниках, этот метод при необходимости может быть переопределен, чтобы отражать действительное равенство объектов.

void finalize() - процедура завершения работы объекта; вызывается перед удалением неиспользуемого объекта. Вызывается Java-машиной перед тем, как сборщик мусора (garbage collection) очистит память, занимаемую объектом. Объект «входит в сферу интересов» сборщика мусора, когда в выполняющейся программе не будет ни одной ссылки на объект. Перед очисткой занимаемой объектом памяти, будет вызван его метод *finalize()*.

Реализация метода *finalize* в классе Object не включает кода, т. е. не выполняет каких-либо действий. В классах-наследниках этот метод может быть переопределен для проведения всех необходимых действий по освобождению различных занимаемых ресурсов - закрытия сетевых соединений, файлов и т. д.

Class<?> getClass() – возвращает объект типа Class, соответствующий классу объекта.

int hashcode() - возвращает хеш (hash code) объекта. Хеш-код представляет собой целое число, которое с большой вероятностью является уникальным для объекта. Обычно метод hashCode не переопределяется. Но, если же, придется это сделать, то необходимо убедиться, чтобы метод возвращал одно и то же значение для равных между собой объектов. То есть, если x.equals(y) возвращает true, то зачения хеш-кодов x и y должны совпадать, то есть вызовы x.hashCode() и y.hashCode() должны возвращать одно и то же значение.

String toString() - возвращает строковое представление объекта.

void notify() - процедура возобновления выполнения потока, который ожидает вызывающего объекта.

void notifyAll() - процедура возобновления выполнения всех потоков, которые ожидают вызывающего объекта.

void wait() - остановка текущего потока и освобождение монитора до тех пор, пока другой поток не вызовет notify() или notifyAll метод для этого объекта.

void wait(long ms) - остановка текущего потока на время или до тех пор, пока другой поток не вызовет notify() или notifyAll метод для этого объекта.

void wait(long ms, int nano) - остановка текущего потока на время или до тех пор, пока другой поток не вызовет notify() или notifyAll метод для этого объекта.

Класс Number. Классы-оболочки. Автоупаковка и автораспаковка.

Абстрактный класс Number является суперклассом для классов BigDecimal, BigInteger, Byte, Double, Float, Integer, Long и Short. Подклассы Number должны обеспечить методы, чтобы преобразовывать представленную числовую величину в byte, double, float, int, long и short.

byte byteValue() - Возвращает величину определенного числа как byte

abstract double doubleValue() - Возвращает величину определенного числа как double

abstract float floatValue() - Возвращает величину определенного числа как float

abstract int intValue() - Возвращает величину определенного числа как int

abstract long longValue() - Возвращает величину определенного числа как long

short shortValue() - Возвращает величину определенного числа как short

Классы-оболочки Java являются Объектным представлением восьми примитивных типов в Java (byte, int, long, short, double, float, boolean, char). Все классы-оболочки в Java являются неизменными и final.

Есть два вида переменных:

- примитивные типы java, хранят непосредственно значение байтов данных;
- ссылочный тип, хранит байты адреса объекта в Heap, то есть через эти переменные мы получаем доступ непосредственно к самому объекту (такой себе пульт от объекта).

Примитивы имеют свои классы-обертки, чтобы можно было работать с ними как с объектами. Эти классы дают возможность сохранять внутри объекта примитив, а сам объект будет вести себя как Object (ну как любой другой объект). При всём этом мы получаем большое количество разношерстных, полезных статических методов, как например — сравнение чисел, перевод символа в регистр, определение того, является ли символ буквой или числом, поиск минимального числа и т. п. Предоставляемый набор функционала зависит лишь от самой обертки. То есть, для каждого примитивного типа существует, соответствующий ему ссылочный тип. Java автоматически делает преобразование между примитивными типами и их обёртками.

Автоупаковка: Преобразование примитивного типа данных в объект соответствующего класса-оболочки.
Автораспаковка: Присваивание объекта класса-оболочки переменной примитивного типа.

Математические функции. Класс Math. Пакет java.lang.math

Класс Math состоит из набора статических методов, выполняющих наиболее популярные математические вычисления, и двух констант, имеющих особое значение в математике — это число Пи и экспонента. Константы определены следующим образом:

- public static final double Math.PI - задает число Пи
- public static final double Math.E - число e

Часто класс Math называют классом-утилитой (Utility class), т. к. все методы класса статические и нет необходимости создавать экземпляр этого класса - поэтому он и не имеет открытого конструктора. Этот класс нельзя также и унаследовать, поскольку он объявлен с атрибутом final.

Математические операции:

● double abs(double) возвращает абсолютное значение числа, есть также варианты для float -- float abs(float), int, и long.

● double ceil(double) - возвращает округленное до большего число

● double floor(double) -- до меньшего

● double min(double, double) -- наименьшее из 2 чисел, переопределен для float, int, long

● double max(double, double) -- наибольшее, также переопределен для float, int, long

● double pow(double, double) -- возводит первое число в степень, переданную как второе

По такому же принципу реализованы тригонометрические cos, sin, tan, acos, asin, atan (аргументы в радианах). Из градусов можно получить радианы с помощью double toRadians(double).

Работа со строками. Классы String, StringBuilder, StringBuffer.

Класс String используется в Java для хранения и представления не модифицируемых строк. После того как создан экземпляр этого класса, строка уже не может быть модифицирована.

String a = “abc”; String b = “abc” -- ссылаются на один объект в памяти.

String c = new String(“abc”) -- ссылается на новый объект.

Строки иммутабельны (константы), операции изменения (конкатенация и др.) создают новые объекты. Для сравнения необходимо использовать equals, который сравнивает строки, == сравнивает ссылки. == вернет true, если обе строки были созданы как литералы (s = “string”), то есть указаны в коде до компиляции. Ввод от пользователя будет новым объектом, == вернет false.

Объекты String являются неизменяемыми, поэтому все операции, которые изменяют строки, фактически приводят к созданию новой строки, что сказывается на производительности приложения. Для решения этой проблемы, чтобы работа со строками проходила с меньшими издержками в Java были добавлены классы **StringBuffer** и **StringBuilder**. По сути, они напоминает расширяемую строку, которую можно изменять без ущерба для производительности.

Эти классы похожи, практически двойники, они имеют одинаковые конструкторы, одни и те же методы, которые одинаково используются. Единственное их различие состоит в том, что класс StringBuffer синхронизированный и потокобезопасный. То есть класс StringBuffer удобнее использовать в многопоточных приложениях, где объект данного класса может меняться в различных потоках. Если же речь о многопоточных приложениях не идет, то лучше использовать класс StringBuilder, который **не потокобезопасный**, но при этом работает быстрее, чем StringBuffer в однопоточных приложениях.

У них есть 4 конструктора:

StringBuffer()

StringBuffer(int capacity)

StringBuffer(String str)

StringBuffer(CharSequence chars)

Метод append() добавляет подстроку в конец строки.

Обработка исключительных ситуаций. Классы Error, Exception, RuntimeException.

Базовым классом для всех исключений является класс Throwable. От него уже наследуются два класса: Error и Exception.

Иногда метод, в котором может генерироваться исключение, сам не обрабатывает это исключение. В этом случае в объявлении метода используется оператор throws, который надо обработать при вызове этого метода.

Все классы ошибок и исключений, которые могут использоваться в throw, наследуются от Throwable. Они включают в себя информацию об ошибке и stack trace. Основные классы:

Error -- критические ошибки, которые не должны перехватываться в try-catch. Класс Error описывает внутренние ошибки в исполняющей среде Java. Примеры: OutOfMemoryError (нехватка памяти), StackOverflowError (слишком глубокая рекурсия)

RuntimeException -- наследуется от Exception, unchecked исключение, которое не обязательно обрабатывать в try-catch, но можно. RuntimeException является базовым классом для так называемой группы непроверяемых исключений (unchecked exceptions) - компилятор не проверяет факт обработки таких исключений и их можно не указывать вместе с оператором throws в объявлении метода. Все остальные классы, образованные от класса Exception, называются проверяемыми исключениями (checked exceptions). Такие исключения являются следствием ошибок разработчика, например, неверное преобразование типов или выход за пределы массива. Пример: NullPointerException

Exception -- исключения, которые должны или обрабатываться в try-catch, или быть указанными в сигнатуре метода (String readFile() throws IOException).

Классы System и Runtime. Класс java.io.Console

Класс System содержит в себе статические поля потоки ввода (stdin -- System.in), вывода (stdout -- System.out) и ошибок (stderr -- System.err), а также статические методы работы с системой, например, String getenv(String) для получения переменной окружения и void exit(int) для выхода из программы с определенным кодом возврата.

У класса Runtime есть один экземпляр на всю программу (Runtime.getInstance()). Он позволяет добавлять потоки, который будут выполнены при завершении работы JVM (через addShutdownHook(Thread)).

Текущую Console можно получить, вызвав System.console(). Метод вернет null, если консоли нет, что зависит от ОС и от того, как запущена программа. Если она запущена из интерактивного терминала, то консоль будет доступна. С ее помощью можно считывать пароли без вывода символов (byte[] readPassword()), остальной функционал можно реализовать через System.in и System.out.

Коллекции. Виды коллекции. Интерфейсы Set, List, Queue.

Для хранения наборов данных в Java предназначены массивы. Однако их не всегда удобно использовать, прежде всего потому, что они имеют фиксированную длину. Эту проблему в Java решают коллекции. Однако суть не только в гибких по размеру наборах объектов, но в и том, что классы коллекций реализуют различные алгоритмы и структуры данных, например, такие как стек, очередь, дерево и ряд других.

Классы коллекций располагаются в пакете java.util, поэтому перед применением коллекций следует подключить данный пакет.

Хотя в Java существует множество коллекций, но все они образуют стройную и логичную систему. Во-первых, в основе всех коллекций лежит применение того или иного интерфейса, который определяет базовый функционал.

Среди этих интерфейсов можно выделить следующие:

- **Collection**: базовый интерфейс для всех коллекций и других интерфейсов коллекций
- **Queue**: наследует интерфейс Collection и представляет функционал для структур данных в виде очереди
- **Deque**: наследует интерфейс Queue и представляет функционал для двунаправленных очередей
- **List**: наследует интерфейс Collection и представляет функциональность простых списков
- **Set**: также расширяет интерфейс Collection и используется для хранения множеств уникальных объектов
- **SortedSet**: расширяет интерфейс Set для создания сортированных коллекций
- **NavigableSet**: расширяет интерфейс SortedSet для создания коллекций, в которых можно осуществлять поиск по соответствию
- **Map**: предназначен для созданий структур данных в виде словаря, где каждый элемент имеет определенный ключ и значение. В отличие от других интерфейсов коллекций не наследуется от интерфейса Collection

С помощью применения вышеописанных интерфейсов и абстрактных классов в Java реализуется широкая палитра классов коллекций - списки, множества, очереди, отображения и другие, среди которых можно выделить следующие:

- **ArrayList**: простой список объектов
- **LinkedList**: представляет связанный список
- **ArrayDeque**: класс двунаправленной очереди, в которой мы можем произвести вставку и удаление как в начале коллекции, так и в ее конце
- **HashSet**: набор объектов или хеш-множество, где каждый элемент имеет ключ - уникальный хеш-код

- **TreeSet**: набор отсортированных объектов в виде дерева
- **LinkedHashSet**: связанное хеш-множество
- **PriorityQueue**: очередь приоритетов
- **HashMap**: структура данных в виде словаря, в котором каждый объект имеет уникальный ключ и некоторое значение
- **TreeMap**: структура данных в виде дерева, где каждый элемент имеет уникальный ключ и некоторое значение

Интерфейс Collection является базовым для всех коллекций, определяя основной функционал.

Интерфейс Collection является обобщенным и расширяет интерфейс Iterable, поэтому все объекты коллекций можно перебирать в цикле по типу for-each.

Среди методов интерфейса Collection можно выделить следующие:

- boolean add (E item): добавляет в коллекцию объект item. При удачном добавлении возвращает true, при неудачном - false
- boolean addAll (Collection<? extends E> col): добавляет в коллекцию все элементы из коллекции col. При удачном добавлении возвращает true, при неудачном - false
- void clear (): удаляет все элементы из коллекции
- boolean contains (Object item): возвращает true, если объект item содержится в коллекции, иначе возвращает false
- boolean isEmpty (): возвращает true, если коллекция пуста, иначе возвращает false
- Iterator<E> iterator (): возвращает объект Iterator для обхода элементов коллекции
- boolean remove (Object item): возвращает true, если объект item удачно удален из коллекции, иначе возвращается false
- boolean removeAll (Collection<?> col): удаляет все объекты коллекции col из текущей коллекции. Если текущая коллекция изменилась, возвращает true, иначе возвращается false
- boolean retainAll (Collection<?> col): удаляет все объекты из текущей коллекции, кроме тех, которые содержатся в коллекции col. Если текущая коллекция после удаления изменилась, возвращает true, иначе возвращается false
- int size (): возвращает число элементов в коллекции
- Object[] toArray (): возвращает массив, содержащий все элементы коллекции

Для создания простых списков применяется интерфейс List, который расширяет функциональность интерфейса Collection. Некоторые наиболее часто используемые методы интерфейса List:

- void add(int index, E obj): добавляет в список по индексу index объект obj
- boolean addAll(int index, Collection<? extends E> col): добавляет в список по индексу index все элементы коллекции col. Если в результате добавления список был изменен, то возвращается true, иначе возвращается false
- E get(int index): возвращает объект из списка по индексу index
- int indexOf(Object obj): возвращает индекс первого вхождения объекта obj в список. Если объект не найден, то возвращается -1
- int lastIndexOf(Object obj): возвращает индекс последнего вхождения объекта obj в список. Если объект не найден, то возвращается -1

- ListIterator<E> listIterator(): возвращает объект ListIterator для обхода элементов списка
- static <E> List<E> of(элементы): создает из набора элементов объект List
- E remove(int index): удаляет объект из списка по индексу index, возвращая при этом удаленный объект
- E set(int index, E obj): присваивает значение объекта obj элементу, который находится по индексу index
- void sort(Comparator<? super E> comp): сортирует список с помощью компаратора comp
- List<E> subList(int start, int end): получает набор элементов, которые находятся в списке между индексами start и end

Очереди представляют структуру данных, работающую по принципу FIFO (first in - first out). То есть чем раньше был добавлен элемент в коллекцию, тем раньше он из нее удаляется. Это стандартная модель однонаправленной очереди.

Обобщенный интерфейс Queue<E> расширяет базовый интерфейс Collection и определяет поведение класса в качестве однонаправленной очереди. Свою функциональность он раскрывает через следующие методы:

- **E element()**: возвращает, но не удаляет, элемент из начала очереди. Если очередь пуста, генерирует исключение NoSuchElementException
- **boolean offer(E obj)**: добавляет элемент obj в конец очереди. Если элемент удачно добавлен, возвращает true, иначе - false
- **E peek()**: возвращает без удаления элемент из начала очереди. Если очередь пуста, возвращает значение null
- **E poll()**: возвращает с удалением элемент из начала очереди. Если очередь пуста, возвращает значение null
- **E remove()**: возвращает с удалением элемент из начала очереди. Если очередь пуста, генерирует исключение NoSuchElementException

Интерфейс **Set** расширяет интерфейс Collection и представляет набор уникальных элементов. Set не добавляет новых методов, только вносит изменения унаследованные. В частности, метод **add()** добавляет элемент в коллекцию и возвращает true, если в коллекции еще нет такого элемента.

Обход элементов коллекции. Интерфейс Iterator.

Итераторы придумали практически тогда, когда и коллекции. Основная задача коллекций была – хранить элементы, а основная задача итератора – выдавать эти элементы по одному. Итератор – это специальный внутренний объект в коллекции, который с одной стороны имеет доступ ко всем ее private данным и знает ее внутреннюю структуру, с другой – реализует общедоступный интерфейс Iterator, благодаря чему все знают, как с ним работать.

Некоторые итераторы имеют внутри себя массив, куда копируются все элементы коллекции во время создания итератора. Это гарантирует, что последующее изменение коллекции не повлияет на порядок и количество элементов.

Методы интерфейса Iterator<E>	Описание
boolean hasNext()	Проверяет, есть ли еще элементы
E next()	Возвращает текущий элемент и переключается на следующий.
void remove()	Удаляет текущий элемент

Кроме итератора есть еще интерфейс Iterable – его должны реализовывать все коллекции, которые поддерживают итератор. У него есть единственный метод:

Методы interface Iterable<T>	Описание
Iterator<T>iterator()	Возвращает объект-итератор

Цикл for-each можно использовать для любых объектов, которые поддерживают итератор. Т.е. ты можешь написать свой класс, добавить ему метод iterator() и сможешь использовать его объекты в правой части конструкции for-each.

Сортировка элементов коллекции. Интерфейсы Comparable и Comparator.

Для того чтобы объекты можно было сравнивать и сортировать необходимо, чтобы они применяли интерфейс Comparable<V>. Интерфейс Comparable содержит один единственный метод int compareTo(E item), который сравнивает текущий объект с объектом, переданным в качестве параметра. Если этот метод возвращает отрицательное число, то текущий объект будет располагаться перед тем, который передается через параметр. Если метод вернет положительное число, то, наоборот, после второго объекта. Если метод возвратит ноль, значит, оба объекта равны.

Однако перед нами может возникнуть проблема, что если разработчик не реализовал в своем классе, который мы хотим использовать, интерфейс Comparable, либо реализовал, но нас не устраивает его функциональность, и мы хотим ее переопределить? На этот случай есть еще более гибкий способ, предполагающий применение интерфейса Comparator<E>. Интерфейс Comparator<E> содержит ряд методов, ключевым из которых является метод compare(T a, T b).

Метод compare также возвращает числовое значение - если оно отрицательное, то объект a предшествует объекту b, иначе - наоборот. А если метод возвращает ноль, то объекты равны.

Для применения интерфейса нам вначале надо создать класс компаратора, который реализует этот интерфейс:

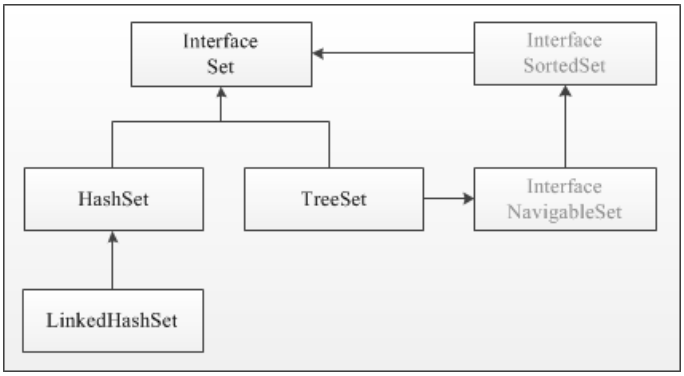
```
class PersonComparator implements Comparator<Person>{

    public int compare(Person a, Person b){

        return a.getName().compareTo(b.getName());
    }
}
```

Интерфейс Set, его варианты и реализации.

Реализация интерфейса Set представляет собой неупорядоченную коллекцию, которая не может содержать дублирующие данные.



Интерфейс **Set** включает следующие методы :

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

К семейству интерфейса Set относятся HashSet, TreeSet и LinkedHashSet. В множествах Set разные реализации используют разный порядок хранения элементов. В HashSet порядок элементов оптимизирован для быстрого поиска. В контейнере TreeSet объекты хранятся отсортированными по возрастанию. LinkedHashSet хранит элементы в порядке добавления.

Интерфейс List и его реализации. Особенности класса Vector.

Интерфейс List сохраняет последовательность добавления элементов и позволяет осуществлять доступ к элементу по индексу. Содержит методы, которые работают с индексами: get(int) для получения элемента по индексу, indexOf(T) для получения индекса по элементу.

Реализации: ArrayList и LinkedList. Для большинства задач (операций с индексом) ArrayList превосходит LinkedList по скорости, преимущество LinkedList -- относительно быстрая вставка и удаления элементов. Vector схож с ArrayList и отличается тем, что его операции синхронизированы, поэтому он работает несколько медленнее, но обладает потокобезопасностью.

По умолчанию в Java есть встроенная реализация этого интерфейса - класс ArrayList. Класс ArrayList представляет обобщенную коллекцию, которая наследует свою функциональность от класса AbstractList и применяет интерфейс List. Проще говоря, ArrayList представляет простой список, аналогичный массиву, за тем исключением, что количество элементов в нем не фиксировано.

Емкость в ArrayList представляет размер массива, который будет использоваться для хранения объектов. При добавлении элементов фактически происходит перераспределение памяти - создание нового массива и копирование в него элементов из старого массива. Изначальное задание емкости ArrayList позволяет снизить подобные перераспределения памяти, тем самым повышая производительность.

Обобщенный класс LinkedList<E> представляет структуру данных в виде связанного списка. Он наследуется от класса AbstractSequentialList и реализует интерфейсы List, Dequeue и Queue. То есть он соединяет функциональность работы со списком и фукциональность очереди.

LinkedList — реализует интерфейсы List, Dequeue, Queue и является представителем двунаправленного списка, где каждый элемент структуры содержит указатели на предыдущий и следующий элементы. Итератор поддерживает обход в обе стороны. LinkedList реализует методы получения, удаления и вставки в начало, середину и конец списка, а также позволяет добавлять любые элементы, в том числе и null.

LinkedList содержит все те методы, которые определены в интерфейсах List, Queue, Deque. Некоторые из них:

- **addFirst() / offerFirst()**: добавляет элемент в начало списка
- **addLast() / offerLast()**: добавляет элемент в конец списка
- **removeFirst() / pollFirst()**: удаляет первый элемент из начала списка
- **removeLast() / pollLast()**: удаляет последний элемент из конца списка
- **getFirst() / peekFirst()**: получает первый элемент
- **getLast() / peekLast()**: получает последний элемент

Класс Vector реализует динамический массив. Он похож на ArrayList, но с двумя отличиями:

- Vector синхронизирован.
- Vector содержит много устаревших методов, которые не являются частью структуры коллекций.

В Java класс Vector оказывается очень полезным, если вы заранее не знаете размер массива или вам нужен только тот, который может изменять размеры за время жизни программы.

It is recommended to use the Vector class in the thread-safe implementation only. If you don't need to use the thread-safe implementation, you should use the ArrayList, the ArrayList will perform better in such case.

The Iterators returned by the Vector class are *fail-fast*. In case of concurrent modification, it fails and throws the ConcurrentModificationException.

Класс Stack – это подкласс Vector, который реализует стандартный стек last-in, first-out.

В Java Stack только определяет стандартный конструктор, который создает пустой стек. Stack включает все методы, определённые Vector, и самостоятельно добавляет несколько своих собственных.

№	Метод и описание
1	**boolean empty()** Проверяет, является ли стек пустым. Возвращает true, если стек пустой. Возвращает false, если стек содержит элементы.
2	**Object peek()** Возвращает элемент, находящийся в верхней части стэка, но не удаляет его.
3	**Object pop()** Возвращает элемент, находящийся в верхней части стэка, удаляя его в процессе.
4	**Object push(Object element)** Вталкивает элемент в стек. Элемент также возвращается.
5	**int search(Object element)** Ищет элемент в стеке. Если найден, возвращается его смещение от вершины стека. В противном случае возвращается 1.

Интерфейс Map, его варианты и реализации

Интерфейс **Map<K, V>** представляет отображение или иначе говоря словарь, где каждый элемент представляет пару "ключ-значение". При этом все ключи уникальные в рамках объекта Map. Такие коллекции облегчают поиск элемента, если нам известен ключ - уникальный идентификатор объекта.

Следует отметить, что в отличие от других интерфейсов, которые представляют коллекции, интерфейс Map НЕ расширяет интерфейс Collection.

Среди методов интерфейса Map можно выделить следующие:

- `void clear()`: очищает коллекцию
- `boolean containsKey(Object k)`: возвращает true, если коллекция содержит ключ k
- `boolean containsValue(Object v)`: возвращает true, если коллекция содержит значение v
- `Set<Map.Entry<K, V>> entrySet()`: возвращает набор элементов коллекции. Все элементы представляют объект `Map.Entry`
- `boolean equals(Object obj)`: возвращает true, если коллекция идентична коллекции, передаваемой через параметр obj
- `boolean isEmpty`: возвращает true, если коллекция пуста
- `V get(Object k)`: возвращает значение объекта, ключ которого равен k. Если такого элемента не окажется, то возвращается значение `null`
- `V getOrDefault(Object k, V defaultValue)`: возвращает значение объекта, ключ которого равен k. Если такого элемента не окажется, то возвращается значение `defaultVlue`
- `V put(K k, V v)`: помещает в коллекцию новый объект с ключом k и значением v. Если в коллекции уже есть объект с подобным ключом, то он перезаписывается. После добавления возвращает предыдущее значение для ключа k, если он уже был в коллекции. Если же ключа еще не было в коллекции, то возвращается значение `null`
- `V putIfAbsent(K k, V v)`: помещает в коллекцию новый объект с ключом k и значением v, если в коллекции еще нет элемента с подобным ключом.

- `Set<K> keySet()`: возвращает набор всех ключей отображения
- `Collection<V> values()`: возвращает набор всех значений отображения
- `void putAll(Map<? extends K, ? extends V> map)`: добавляет в коллекцию все объекты из отображения map
- `V remove(Object k)`: удаляет объект с ключом k
- `int size()`: возвращает количество элементов коллекции

Класс HashMap использует хеш-таблицу для хранения карточки, обеспечивая быстрое время выполнения запросов get() и put() при больших наборах. Класс реализует интерфейс Map (хранение данных в виде пар ключ/значение). Ключи и значения могут быть любых типов, в том числе и null. При этом все ключи обязательно должны быть уникальны, а значения могут повторяться. Данная реализация не гарантирует порядка элементов.

Интерфейс **SortedMap** расширяет Map и создает отображение, в котором все элементы отсортированы в порядке возрастания их ключей. SortedMap добавляет ряд методов:

- K firstKey(): возвращает ключ первого элемента отображения
- K lastKey(): возвращает ключ последнего элемента отображения
- SortedMap<K, V> headMap(K end): возвращает отображение SortedMap, которые содержит все элементы оригинального SortedMap вплоть до элемента с ключом end
- SortedMap<K, V> tailMap(K start): возвращает отображение SortedMap, которые содержит все элементы оригинального SortedMap, начиная с элемента с ключом start
- SortedMap<K, V> subMap(K start, K end): возвращает отображение SortedMap, которые содержит все элементы оригинального SortedMap вплоть от элемента с ключом start до элемента с ключом end

Интерфейс **NavigableMap** расширяет интерфейс SortedMap и обеспечивает возможность получения элементов отображения относительно других элементов. Его основные методы:

- `Map.Entry<K, V> ceilingEntry(K obj)`: возвращает элемент с наименьшим ключом k, который больше или равен ключу obj (k >=obj). Если такого ключа нет, то возвращается null.
- `Map.Entry<K, V> floorEntry(K obj)`: возвращает элемент с наибольшим ключом k, который меньше или равен ключу obj (k <=obj). Если такого ключа нет, то возвращается null.
- `Map.Entry<K, V> higherEntry()`: возвращает элемент с наименьшим ключом k, который больше ключа obj (k >obj). Если такого ключа нет, то возвращается null.
- `Map.Entry<K, V> lowerEntry()`: возвращает элемент с наибольшим ключом k, который меньше ключа obj (k <obj). Если такого ключа нет, то возвращается null.

Классы Collections и Arrays.

Каркас коллекций определяет несколько алгоритмов, которые могут быть применимы к коллекциям и картам. Эти алгоритмы определены как статические методы в классе Collections.

Collections содержит статические методы для работы с коллекциями, а именно sort(List) и sort(List, Comparator), reverse(List), binarySearch(), copy() и др.

Arrays работает с массивами (в том числе примитивов), позволяя их копировать, сортировать, создавать с ними Stream. Arrays.asList(array) возвращает List, созданный из массива, для его использования как коллекции. Обратную операцию выполняет Collection.toArray() (set.toArray(), list.toArray(), …). Arrays.sort([]a) – упорядочивание массива в порядке возрастания.

Обобщённые и параметризованные типы. Создание обобщённых классов.

Параметризованные типы позволяют объявлять классы, интерфейсы и методы, где тип данных, к оторыми они оперируют, указан в виде параметра. Используя дженерики, можно создать единственный класс, например, который будет автоматически работать с разными типами данных.

Классы, интерфейсы или методы, имеющие дело с параметризованными типами, называются параметризованными или обобщениями, параметризованными (обобщенными) классами или параметризованными (обобщёнными) методами.

Следующий пример демонстрирует использование параметризованного класса, который описывает матрицу:

```
public class Matrix<T> {
    private T[] array;

    public Matrix(T[] array) {
        this.array = array.clone();
    }

    public static void main(String[] args) {
        Matrix<Double> doubleMatrix = new Matrix<>(new Double[2]);
        Matrix<Integer> integerMatrix = new Matrix<>(new Integer[4]);
        Matrix<Byte> byteMatrix = new Matrix<>(new Byte[7]);
    }
}
```

Обобщенный класс может быть объявлен с любым количеством параметров типа. Например:

```
public class TwoGen<T, V> {
    private T obT;
    private V obV;

    public TwoGen(T obT, V obV) {
        this.obT = obT;
        this.obV = obV;
    }
```

Работа с параметризованными методами. Ограничение типа сверху или снизу.

Методы (включая статические) также могут иметь обобщенные типы:

```
public class Util

{

        public static <T> boolean compare(Box<T> b1, Box<T> b2)

        {

                return b1.get().equals(b2.get());

        }

}

Util.<Integer>compare(b1, b2); // вызов
```

Ограниченные типы. Указывая параметр типа, можно наложить ограничение сверху в виде верхней границы, где объявляется суперкласс, от которого должны быть унаследованы все аргументы типов. С этой целью вместе с параметром указывается ключевое слово extends:

```
class Gen <T extends Superclass>
```

Потоки ввода-вывода в Java. Байтовые и символьные потоки.

Отличительной чертой многих языков программирования является работа с файлами и потоками. В Java основной функционал работы с потоками сосредоточен в классах из пакета java.io.

абстракции, которая используется для чтения или записи информации (файлов, сокетов, текста консоли и т.д.).

В основе всех классов, управляющих потоками байтов, находятся два абстрактных класса: **InputStream** (представляющий потоки ввода) и **OutputStream** (представляющий потоки вывода).

Класс InputStream

Класс InputStream является базовым для всех классов, управляющих байтовыми потоками ввода. Рассмотрим его основные методы:

- `int available()`: возвращает количество байтов, доступных для чтения в потоке
- `void close()`: закрывает поток
- `int read()`: возвращает целочисленное представление следующего байта в потоке. Когда в потоке не останется доступных для чтения байтов, данный метод возвратит число -1
- `int read(byte[] buffer)`: считывает байты из потока в массив buffer. После чтения возвращает число считанных байтов. Если ни одного байта не было считано, то возвращается число -1
- `int read(byte[] buffer, int offset, int length)`: считывает некоторое количество байтов, равное length, из потока в массив buffer. При этом считанные байты помещаются в массиве, начиная со смещения offset, то есть с элемента `buffer[offset]`. Метод возвращает число успешно прочитанных байтов.
- `long skip(long number)`: пропускает в потоке при чтении некоторое количество байт, которое равно number

Класс OutputStream

Класс OutputStream является базовым классом для всех классов, которые работают с бинарными потоками записи. Свою функциональность он реализует через следующие методы:

- `void close()`: закрывает поток
- `void flush()`: очищает буфер вывода, записывая все его содержимое
- `void write(int b)`: записывает в выходной поток один байт, который представлен целочисленным параметром b
- `void write(byte[] buffer)`: записывает в выходной поток массив байтов buffer.

- `void write(byte[] buffer, int offset, int length)`: записывает в выходной поток некоторое число байтов, равное `length`, из массива `buffer`, начиная со смещения `offset`, то есть с элемента `buffer[offset]`.

Абстрактные классы Reader и Writer

Абстрактный класс Reader предоставляет функционал для чтения текстовой информации. Рассмотрим его основные методы:

- `absract void close()`: закрывает поток ввода
- `int read()`: возвращает целочисленное представление следующего символа в потоке. Если таких символов нет, и достигнут конец файла, то возвращается число -1
- `int read(char[] buffer)`: считывает в массив buffer из потока символы, количество которых равно длине массива buffer. Возвращает количество успешно считанных символов. При достижении конца файла возвращает -1
- `int read(CharBuffer buffer)`: считывает в объект CharBuffer из потока символы. Возвращает количество успешно считанных символов. При достижении конца файла возвращает -1
- `absract int read(char[] buffer, int offset, int count)`: считывает в массив buffer, начиная со смещения offset, из потока символы, количество которых равно count
- `long skip(long count)`: пропускает количество символов, равное count. Возвращает число успешно пропущенных символов

Класс Writer определяет функционал для всех символьных потоков вывода. Его основные методы:

- `Writer append(char c)`: добавляет в конец выходного потока символ c. Возвращает объект Writer
- `Writer append(CharSequence chars)`: добавляет в конец выходного потока набор символов chars. Возвращает объект Writer
- `abstract void close()`: закрывает поток
- `abstract void flush()`: очищает буферы потока
- `void write(int c)`: записывает в поток один символ, который имеет целочисленное представление
- `void write(char[] buffer)`: записывает в поток массив символов
- `absract void write(char[] buffer, int off, int len)` : записывает в поток только несколько символов из массива buffer. Причем количество символов равно len, а отбор символов из массива начинается с индекса off
- `void write(String str)`: записывает в поток строку
- `void write(String str, int off, int len)`: записывает в поток из строки некоторое количество символов, которое равно len, причем отбор символов из строки начинается с индекса off

Функционал, описанный классами Reader и Writer, наследуется непосредственно классами символьных потоков, в частности классами `FileReader` и `FileWriter` соответственно, предназначенными для работы с текстовыми файлами.

IO: потокоориентированный, блокирующий (синхронный) ввод/вывод

Новый пакет ввода-вывода. Буферы и каналы.

NIO: буфер-ориентированный, неблокирующий (асинхронный) ввод/вывод, селекторы (позволяют потоку мониторить несколько каналов ввода).

Система ввода/вывода построена на 2-х элементах: буфере (хранение данных) и канале (предоставляет открытое соединение с устройством ввода-вывода (файл, сокет))

Класс Buffer - имеет текущую позицию (текущий индекс), предел (следующий после последнего индекс), емкость. Определена отмена и очистка буфера.

Потоки ввода/вывода (streams) в Java IO являются блокирующими. Это значит, что когда в потоке выполнения (tread) вызывается read() или write() метод любого класса из пакета java.io.*, происходит блокировка до тех пор, пока данные не будут считаны или записаны. Поток выполнения в данный момент не может делать ничего другого.
Неблокирующий режим Java NIO позволяет запрашивать считанные данные из канала (channel) и получать только то, что доступно на данный момент, или вообще ничего, если доступных данных пока нет. Вместо того, чтобы оставаться заблокированным пока данные не станут доступными для считывания, поток выполнения может заняться чем-то другим.

Каналы – это логические (не физические) порталы, через которые осуществляется ввод/вывод данных, а буферы являются источниками или приёмниками этих переданных данных. При организации вывода, данные, которые вы хотите отправить, помещаются в буфер, а он передается в канал. При вводе данные из канала помещаются в предоставленный вами буфер.

Работа с файлами в Java. Классы java.io.File и java.nio.Path.

Класс File, определенный в пакете java.io, не работает напрямую с потоками. Его задачей является управление информацией о файлах и каталогах. Хотя на уровне операционной системы файлы и каталоги отличаются, но в Java они описываются одним классом File.

В зависимости от того, что должен представлять объект File - файл или каталог, мы можем использовать один из конструкторов для создания объекта:

```
File(String путь_к_каталогу)
File(String путь_к_каталогу, String имя_файла)
File(File каталог, String имя_файла)
```

Класс File имеет ряд методов, которые позволяют управлять файлами и каталогами. Рассмотрим некоторые из них:

- **boolean createNewFile()**: создает новый файл по пути, который передан в конструктор. В случае удачного создания возвращает true, иначе false
- **boolean delete()**: удаляет каталог или файл по пути, который передан в конструктор. При удачном удалении возвращает true.
- **boolean exists()**: проверяет, существует ли по указанному в конструкторе пути файл или каталог. И если файл или каталог существует, то возвращает true, иначе возвращает false
- **String getAbsolutePath()**: возвращает абсолютный путь для пути, переданного в конструктор объекта
- **String getName()**: возвращает краткое имя файла или каталога
- **String getParent()**: возвращает имя родительского каталога

- **boolean isDirectory()**: возвращает значение true, если по указанному пути располагается каталог
- **boolean isFile()**: возвращает значение true, если по указанному пути находится файл
- **boolean isHidden()**: возвращает значение true, если каталог или файл являются скрытыми
- **long length()**: возвращает размер файла в байтах
- **long lastModified()**: возвращает время последнего изменения файла или каталога. Значение представляет количество миллисекунд, прошедших с начала эпохи Unix
- **String[] list()**: возвращает массив файлов и подкаталогов, которые находятся в определенном каталоге
- **File[] listFiles()**: возвращает массив файлов и подкаталогов, которые находятся в определенном каталоге
- **boolean mkdir()**: создает новый каталог и при удачном создании возвращает значение true
- **boolean renameTo(File dest)**: переименовывает файл или каталог

Path, по большому счету, — это переработанный аналог класса File. Работать с ним значительно проще, чем с File. **Во-первых**, из него убрали многие утилитные (статические) методы, и перенесли их в класс Files. **Во-вторых**, в Path были упорядочены возвращаемые значения методов. В классе File методы возвращали то String, то boolean, то File — разобраться было непросто. Например, был метод **getParent()**, который возвращал родительский путь для текущего файла в виде строки. Но при этом был метод **getParentFile()**, который возвращал то же самое, но в виде объекта File! Это явно избыточно. Поэтому в интерфейсе Path метод getParent() и другие методы работы с файлами возвращают просто объект Path. Никакой кучи вариантов — все легко и просто.

Вот некоторые из них и примеры их работы:

- **getFileName()** — возвращает имя файла из пути;
- **getParent()** — возвращает «родительскую» директорию по отношению к текущему пути (то есть ту директорию, которая находится выше по дереву каталогов);
- **getRoot()** — возвращает «корневую» директорию; то есть ту, которая находится на вершине дерева каталогов;
- **startsWith()**, **endsWith()** — проверяют, начинается/заканчивается ли путь с переданного пути

Сериализация объектов. Интерфейс Serializable.

Сериализация представляет процесс записи состояния объекта в поток, соответственно процесс извлечения или восстановления состояния объекта из потока называется десериализацией. Сериализация очень удобна, когда идет работа со сложными объектами.

Сразу надо сказать, что сериализовать можно только те объекты, которые реализуют интерфейс Serializable. Этот интерфейс не определяет никаких методов, просто он служит указателем системе, что объект, реализующий его, может быть сериализован.

Для сериализации объектов в поток используется класс ObjectOutputStream. Он записывает данные в поток.

Для создания объекта ObjectOutputStream в конструктор передается поток, в который производится запись:

```
ObjectOutputStream(OutputStream os)
```

Для записи данных ObjectOutputStream использует ряд методов, среди которых можно выделить следующие:

- **void close()**: закрывает поток
- **void flush()**: очищает буфер и сбрасывает его содержимое в выходной поток

- **void write(byte[] buf)**: записывает в поток массив байтов
- **void write(int val)**: записывает в поток один младший байт из val
- **void writeBoolean(boolean val)**: записывает в поток значение boolean
- **void writeByte(int val)**: записывает в поток один младший байт из val
- **void writeChar(int val)**: записывает в поток значение типа char, представленное целочисленным значением
- **void writeDouble(double val)**: записывает в поток значение типа double
- **void writeFloat(float val)**: записывает в поток значение типа float
- **void writeInt(int val)**: записывает целочисленное значение int
- **void writeLong(long val)**: записывает значение типа long
- **void writeShort(int val)**: записывает значение типа short
- **void writeUTF(String str)**: записывает в поток строку в кодировке UTF-8
- **void writeObject(Object obj)**: записывает в поток отдельный объект

Класс ObjectInputStream отвечает за обратный процесс - чтение ранее сериализованных данных из потока. В конструкторе он принимает ссылку на поток ввода:

ObjectInputStream ois = new ObjectInputStream(InputStream is)

Функционал ObjectInputStream сосредоточен в методах, предназначенных для чтения различных типов данных. Рассмотрим основные методы этого класса:

- **void close()**: закрывает поток
- **int skipBytes(int len)**: пропускает при чтении несколько байт, количество которых равно len
- **int available()**: возвращает количество байт, доступных для чтения
- **int read()**: считывает из потока один байт и возвращает его целочисленное представление
- **boolean readBoolean()**: считывает из потока одно значение boolean
- **byte readByte()**: считывает из потока один байт
- **char readChar()**: считывает из потока один символ char
- **double readDouble()**: считывает значение типа double
- **float readFloat()**: считывает из потока значение типа float
- **int readInt()**: считывает целочисленное значение int
- **long readLong()**: считывает значение типа long
- **short readShort()**: считывает значение типа short
- **String readUTF()**: считывает строку в кодировке UTF-8
- **Object readObject()**: считывает из потока объект

Библиотеки графического интерфейса. Особенности и различия.

Abstract Window Toolkit

- Библиотека зависимая от графической подсистемы ОС
- Должна выглядеть одинаково «хорошо» на разных платформах, но получилось, что все компоненты выглядят плохо и по-разному

Достоинства:

- часть JDK;
- скорость работы;
- графические компоненты похожи на стандартные.

Недостатки:

- использование нативных компонентов налагает ограничения на использование их свойств. Некоторые компоненты могут вообще не работать на «неродных» платформах;
- некоторые свойства, такие как иконки и всплывающие подсказки, в AWT вообще отсутствуют;
- стандартных компонентов AWT очень немного, программисту приходится реализовывать много кастомных;
- программа выглядит по-разному на разных платформах (может быть кривоватой).

Библиотека Swing. Особенности.

С версии 1.1 появилась библиотека Swing, сначала как отдельная библиотека, потом как часть Java. Основная особенность Swing в том, что почти все компоненты написаны на Java. Поэтому в принципе они должны выглядеть одинаково везде. Но, из-за того, что компоненты нужно отрисовывать, все стало работать медленнее, чем было в AWT. Но потом провели работу над ускорением и сейчас правильно настроенный Swing работает достаточно быстро. Полезной особенностью Swing является изменяемый вид компонентов, который можно менять на ходу. Не сказать, что Swing является самой удачной библиотекой, но изучить на его примере основные принципы вполне можно.

Библиотека SWT. Особенности.

Библиотека SWT появилась как часть Eclipse, ее разработку поддержала компания IBM. SWT есть не для всех платформ, но при ее создании разработчики удачно соединили лучшее из AWT и Swing. В SWT компоненты и их функции, которые поддерживаются графической подсистемой, как и в AWT работают через адаптеры, а недостающие функции, как в Swing, дописаны на Java.

Библиотека JavaFX. Особенности.

- новая графическая библиотека
- улучшенная поддержка анимации
- визуальные эффекты
- XML для задания интерфейса
- CSS для задания стилей

Компоненты графического интерфейса. Класс Component.

Если рассматривать процесс создания графических приложений в общем, то все происходит примерно так: создается основное окно, в нем размещаются компоненты графического интерфейса. Далее обеспечивается реакция компонентов на события, и можно считать, что все работает.

Компонент (widget, control) – отображаемый и взаимодействующий с пользователем элемент GUI.

- Java.awt.Component – абстрактный класс – элемент GUI
- Размер, цвет, местоположение - свойства
- Порождает основные события

Цвет компонента характеризуется классом Color в котором есть набор констант для основных цветов (Color.RED, Color.BLACK...), можно создать цвет с помощью конструкторов (Color (r, g, b, [a])). Цвет задается либо с помощью трех составляющих RGB, возможно с указанием прозрачности. Значения составляющих могут быть от 0 до 255, либо от 0 до 1.

Есть методы, которые позволяют получить отдельные составляющие части цвета (getRed(), getGreen()...), а есть методы, которые позволяют получить цвет светлее или темнее текущего (brighter(), darker()).

Соответственно, у каждого компонента есть методы, которые позволяют управлять его цветом - получать или устанавливать основной или фоновый цвет:

- Color getForeground()
- void setForeground(Color)
- Color getBackground()
- void setBackground(Color)

Положение и размер компонента характеризуются прямоугольником, в который вписывается компонент. Чтобы задать прямоугольник, нужно указать его положение (координаты верхней левой точки), и размер (высоту и ширину). Положение (Location) задается классом Point - точка с координатами x, y. Размер (Size) задается классом Dimension, представляющим высоту и ширину. Еще одной характеристикой можно считать границы компонента (Bounds), которые совмещают положение и размер, и задаются с помощью класса Rectangle - прямоугольник с начальной точкой и размерами.

У компонента есть методы, которые позволяют задать или получить ограничивающий прямоугольник — setBounds и getBounds, а также отдельно получить или изменить положение начальной точки и размер компонента.

Класс Font

- физические (Arial, Times)
- логические (Dialog, DialogInput, Serif, SansSerif, Monospaced)
- Константы: ◊ Font.DIALOG, Font.MONOSPACED, Font.SERIF, Font.SANS_SERIF ◊ Font.PLAIN, Font.BOLD, Font.ITALIC
- Конструктор Font(String name, int style, int size)
- Методы ◊ String getFontName(), int getStyle(), int getSize()

У класса Component есть методы getFont() и setFont().

Видимость – компоненты изначально видимы, кроме основных окон (Window, Frame).

boolean isVisible(), void setVisible (boolean)

Активность – компоненты изначально активны (воспринимают действия пользователя и порождают события).

boolean isEnabled(), void setEnabled (boolean)

Методы для рисования тоже находятся в классе Component. Это paint, update и repaint. Методам paint и update передается объект класса Graphics — это графический контекст компонента, набор пикселей который определяет как этот компонент выглядит. Метод paint может вызываться двумя способами: 1) системным, когда paint вызывается при первом отображении компонента или при необходимости перерисовки из-за перемещения окон, либо изменении размера компонента; 2) программным, если мы сами изменили картинку компонента, например, при анимации, тогда в программе нужно вызвать repaint(), который занесет в очередь событие для перерисовки компонента и через какое-то время вызовется paint().

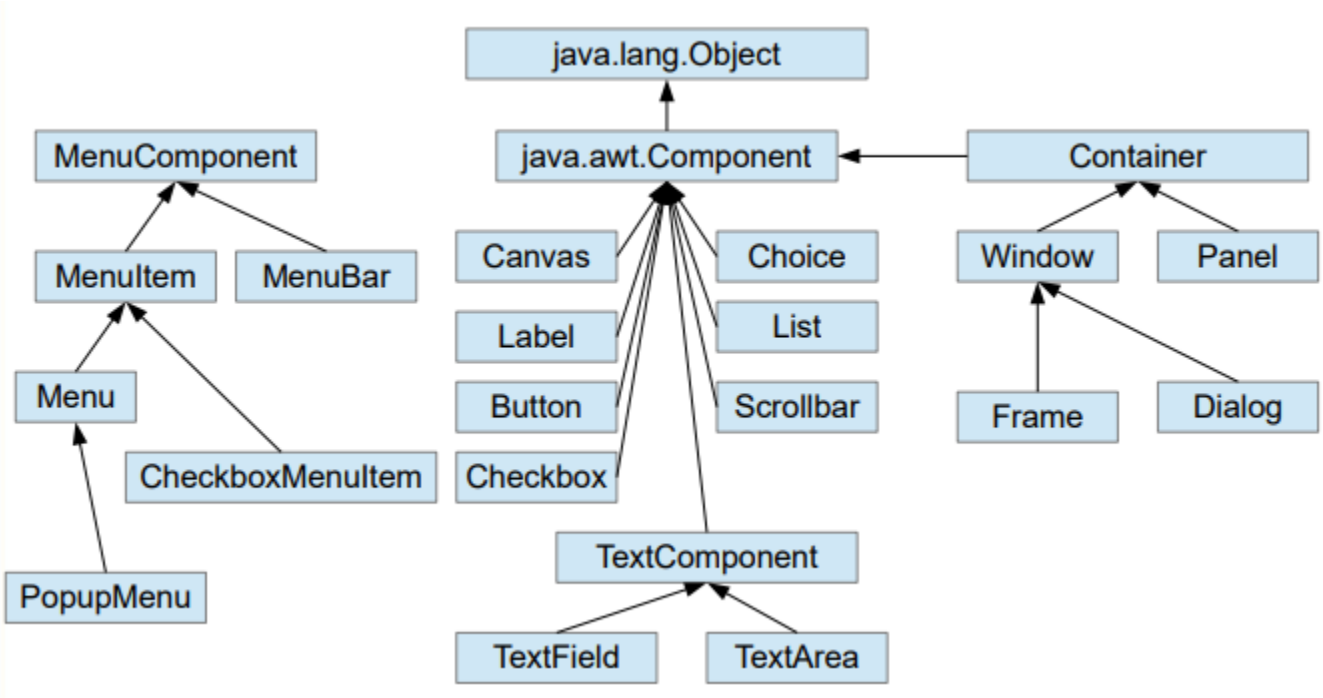
Размещение компонентов в контейнерах. Менеджеры компоновки.

Контейнер – компонент, который содержит другие компоненты.

Класс java.awt.Container наследуется от java.awt.Component.

Компонент может находиться только в одном контейнере.

Основные методы: add(Component), setLayout(LayoutManager), validate().



Размещение компонентов в контейнере:

- Абсолютное позиционирование – отсутствует реакция на изменение размера контейнера, могут быть проблемы с изменением шрифта или локали. **плохо**
- Менеджер компоновки – управляет взаимным расположением и размером компонентов. **Хорошо**

Менеджеры компоновки:

BorderLayout - размещает элементы в один из пяти регионов, как было указано при добавлении элемента в контейнер: наверх, вниз, влево, вправо, в центр;

FlowLayout - размещает элементы по порядку в том же направлении, что и ориентация контейнера (слева направо по умолчанию) применяя один из пяти видов выравнивания, указанного при создании менеджера. Данный менеджер используется по умолчанию в большинстве контейнерах;

GridLayout - размещает элементы таблично. Количество столбцов и строк указывается при создании менеджера. По умолчанию одна строка, а число столбцов равно числу элементов;

BoxLayout - размещает элементы по вертикали или по горизонтали. Обычно он используется не напрямую а через контейнерный класс Box, который имеет дополнительные возможности;

SpringLayout - это менеджер низкого уровня и был разработан для программ построителей форм;

GroupLayout - размещает компоненты в простой равномерной сетке. Конструктор этого класса позволяет задавать количество строк и столбцов

Обработка событий графического интерфейса

Компоненты графического интерфейса способны вызывать события. С помощью метода addActionListener() можно подписать объект ActionListener на это событие, что означает, что при вызове события у каждого ActionListener`a, подписанного на это событие будет вызван метод actionPerformed(ActionEvent a), который мы бережно переопределим в соответствии с логикой нашей программы. На одно событие может быть подписано сразу несколько слушателей. JavaFX позволяет добавлять слушателей не только на компоненты, но и на отдельные значения - properties. В таком случае, вызов события произойдет при изменении значения этой property.

Атомарные типы данных

Атомарность означает выполнение операции целиком непрерывно (либо невыполнение ее вовсе). В Java атомарными являются операции чтения/записи всех примитивных типов данных за исключением типов long и double, поскольку эти типы данных занимают два машинных слова, и операции чтения/записи являются составными операциями из двух атомарных операций над старшими и младшими битами числа соответственно. Однако операции над volatile long и volatile double атомарны. Операции над ссылками на объекты в Java являются всегда атомарными независимо от разрядности JVM и гарантируются JMM (Java Memory Model).

Шаблоны проектирования. Структурные шаблоны

Структурные шаблоны — шаблоны проектирования, в которых рассматривается вопрос о том, как из классов и объектов образуются более крупные структуры. К структурным шаблонам относятся:

● Адаптер: обеспечивает доступ к объекту, который содержит необходимые данные и поведение, но имеет неподходящий интерфейс.

● Мост: используется для того, чтобы «разделять абстракцию и реализацию так, чтобы они могли изменяться независимо» ?

● Компоновщик: структурный шаблон проектирования, объединяющий объекты в древовидную структуру для представления иерархии от частного к целому, определяет иерархию классов, которые одновременно могут состоять из примитивных и сложных объектов, упрощает архитектуру клиента, делает процесс добавления новых видов объекта более простым

● Декоратор: структурный шаблон проектирования, предназначенный для динамического подключения дополнительного поведения к объекту. Декоратор принимает объект класса, функциональность которого необходимо расширить, и по необходимости переопределяет существующие или добавляет новые методы, при этом интерфейс взаимодействия с декоратором остается таким же, как с исходным классом

● Фасад: структурный шаблон проектирования, позволяющий скрыть сложность системы путем сведения всех возможных внешних вызовов к одному объекту, делегирующему их соответствующим объектам системы

● Приспособленец: структурный шаблон проектирования, при котором объект, пр едставляющий себя как уникальный экземпляр в разных местах программы, по факту не является таковым. Есть внутренние свойства (неизменные) и внешние (переменные). Когда хотим создать новый вызываем фабрику, внутри - хеш с созданными, если нужный уже есть - берём его, иначе - создаем, кладем в хеш и возвращаем созданный (например)

● Заместитель: структурный шаблон проектирования, предоставляющий объект, который контролирует доступ к другому объекту, перехватывая все вызовы (выполняет функцию контейнера). «Заместитель» может быть использован везде, где ожидается «Реальный Субъект». При необходимости запросы могут быть переадресованы «Заместителем» «Реальному Субъекту»

Шаблоны проектирования. Порождающие шаблоны

– это паттерны, которые имеют дело с механизмами создания объекта и пытаются создать объекты в порядке, подходящем к ситуации.

● Абстрактная фабрика: создает ряд связанных или зависимых объектов без указания их конкретных классов. Обычно создаваемые классы стремятся реализовать один и тот же интерфейс. Клиент абстрактной фабрики не заботится о том, как создаются эти объекты, он просто знает, по каким признакам они взаимосвязаны и как с ними обращаться.

● Строитель: отделяет конструирование сложного объекта от его представления так, что в результате одно го и того же процесса конструирования могут получаться разные представления.

● Фабричный метод: порождающий шаблон проектирования, предоставляющий подклассам интерфейс для создания экземпляров некоторого класса. В момент создания наследники могут определить, какой класс создавать. Иными словами, данный шаблон делегирует создание объектов наследникам родительского класса. Это позволяет использовать в коде программы не специфические классы, а манипулировать абстрактными объектами на более высоком уровне.

● Объектный пул: Порождающий паттерн, который предоставляет набор заранее инициализированных объектов, готовых к использованию («пул»), что не требует каждый раз создавать и уничтожать их. Хранение объектов в пуле может заметно повысить производительность в ситуациях, когда стоимость инициализации экземпляра класса высока, скорость экземпляра класса высока, а количество одновременно используемых экземпляров в любой момент времени является низкой. Время на извлечение объекта из пула легко прогнозируется, в отл ичие от создания новых объектов (особенно с сетевым оверхедом), что занимает неопределённое время.

● Прототип: помогает избежать затрат на создание объектов стандартным способом (new Foo()), а вместо этого создаёт прототип и затем клонирует его

● Синглтон: позволяет содержать только один экземпляр объекта в приложении, которое будет обрабатывать все обращения, запрещая создавать новый экземпляр.

Шаблоны проектирования. Поведенческие шаблоны

Поведенческие шаблоны проектирования определяют общие закономерно сти связей между объектами, реализующими данные паттерны. Следование этим шаблонам уменьшает связность системы и облегчает коммуникацию между объектами, что улучшает гибкость программного продукта.

● Цепочка обязанностей: построить цепочку объектов для об работки вызова в последовательном порядке. Если один объект не может справиться с вызовом, он делегирует вызов для следующего в цепи и так далее.

● Команда: В объектно-ориентированном программировании шаблон проектирования Команда является поведенческим шаблоном, в котором объект используется для инкапсуляции всей информации, необходимой для выполнения действия или вызова события в более позднее время. Эта информация включает в себя имя

метода, объект, который является владельцем метода и значения параметров метода. Четыре термина всегда связаны с шаблоном Команда: команды (command), приёмник команд (receiver), вызывающий команды (invoker) и клиент (client).

● Интерпретатор: позволяет управлять поведением с помощью простого языка

● Итератор: поведенческий шаблон проектирования. Представляет собой объект, позволяющий получить последовательный доступ к элементам объекта-агрегата без использования описаний каждого из агрегированных объектов.

● Посредник: поведенческий шаблон проектирования, обеспечивающий взаимодействие множества объектов, формируя при этом слабую связанность и избавляя объекты от необходимости явно ссылаться друг на друга. "Посредник" определяет интерфейс для обмена информацией с объектами "Коллеги", "Конкретный посредник" координирует действия объектов "Коллеги". Каждый класс "Коллеги" знает о своем объекте "Посредник", все "Коллеги" обмениваются информацией только с посредником, при его отсутствии им пришлось бы обмениваться информацией напрямую. "Коллеги" посылают запросы посреднику и получают запросы от него. "Посредник" реализует кооперативное поведение, пересылая каждый запрос одному или нескольким "Коллегам".

● Хранитель: поведенческий шаблон проектирования, позволяющий, не нарушая инкапсуляцию, зафиксировать и сохранить внутреннее состояние объекта так, чтобы позднее восстановить его в это состояние.

● Наблюдатель: Для реализации публикации/подписки на поведение объекта, всякий раз, когда объект «Subject» меняет свое состояние, прикрепленные объекты «Observers» будут уведомлены. Паттерн используется, чтобы сократить количество связанных напрямую объектов и вместо этого использует слабую связь (loose coupling) .

● Состояние: Изменяет поведение объекта в зависимости от состояния — реализация конечного автомата ● Стратегия: Выбор одного из алгоритмов, реализованных в классе

● Шаблонный метод: Позволяет реализовать часть поведения в базовом классе, остальное реал изуется в подклассах

● Посетитель: поведенческий шаблон проектирования, описывающий операцию, которая выполняется над объектами других классов. При изменении visitor нет необходимости изменять обслуживаемые классы. ○ Добавьте метод accept(Visitor) в иерархию «элемент». ○ Создайте базовый класс Visitor и определите методы visit() для каждого типа элемента. ○ Создайте производные классы Visitor для каждой операции, исполняемой над элементами. ○ Клиент создаёт объект Visitor и передаёт его в вызываемый метод accept().

Сетевое взаимодействие. Основные протоколы, их сходства и отличия.

Пакет java.net обеспечивает поддержку двух общих сетевых протоколов:

- **TCP** - TCP - это протокол управления передачей, который обеспечивает надежную связь между двумя приложениями. В Java TCP обычно используется через Интернет-протокол, который называется TCP/IP.
- **UDP** - UDP - это протокол пользовательских дейтаграмм, протокол без установления соединения, который позволяет передавать пакеты данных между приложениями

Протокол TCP. Классы Socket и ServerSocket.

TCP – транспортный протокол передачи данных в сетях TCP/IP, предварительно устанавливающий соединение с сетью. (Электронная почта, загрузка)

TCP гарантирует доставку пакетов данных в неизменных виде, последовательности и без потерь

TCP нумерует пакеты при передаче

TCP работает в дуплексном режиме, в одном пакете можно отправлять информацию и подтверждать получение предыдущего пакета.

TCP требует заранее установленного соединения

TCP надежнее и осуществляет контроль над процессом обмена данными.

ServerSocket - заставляет программу ждать подключений от клиентов. При создании необходимо указать порт, на котором будет работать сервер и вызвать метод accept() (заставляет ждать подключения к порту). После подключения создается нормальный Socket для выполнения операций с сокетом. (Этот же сокет отображает другой конец подключения)

ServerSocket ss = new ServerSocket(port);

Socket socket = ss.accept();

С другой стороны, Socket класс можно создать, указав IP-адрес и порт

Протокол UDP. Классы DatagramSocket и DatagramPacket.

UDP (User Datagram Protocol) не устанавливает виртуального соединения и не гарантирует доставку данных. Отправитель просто посылает пакеты по указанному адресу; если отосланная информация была повреждена или вообще не дошла, отправитель об этом даже не узнает. Однако достоинством UDP является высокая скорость передачи данных. Данный протокол часто используется при трансляции аудио- и видеосигналов, где потеря небольшого количества данных не может привести к серьезным искажениям всей информации. UPD не содержит функций восстановления данных и не нумерует пакеты при передаче.

По протоколу UDP данные передаются пакетами. Пакетом в этом случае UDP является объект класса **DatagramPacket**. Этот класс содержит в себе передаваемые данные, представленные в виде массива байт.

Конструкторы класса:

DatagramPacket(byte[] buf, int length)

DatagramPacket(byte[] buf, int length, InetAddress address, int port)

DatagramPacket(byte[] buf, int length, SocketAddress address)

Класс **DatagramSocket** может выступать в роли клиента и сервера, то есть он способен получать и отправлять пакеты. Отправить пакет можно с помощью метода **send(DatagramPacket pac)**, для получения пакета используется метод **receive(DatagramPacket pac)**.

Интернационализация. Локализация. Хранение локализованных ресурсов.

Интернационализация — проектирование программы таким образом, чтобы локализация была возможна без конструктивных изменений.

- выделение текстовых данных из кода
- отображение данных с учетом местных форматов

Локализация - процесс адаптации программного обеспечения к культуре какой-либо страны.

- перевод текстов
- использование соответствующих форматов данных
- замена звуковой и визуальной информации

Класс Java java.util.Locale позволяет учесть особенности региональных представлений алфавита, символов, чисел и дат. Автоматически виртуальная машина использует текущие региональные установки операционной системы, но при необходимости их можно изменять.

Для некоторых стран региональные параметры устанавливаются с помощью констант, например: Locale.US, Locale.FRANCE.

Для всех остальных объект класса Locale нужно создавать с помощью конструктора, например: Locale rus = new Locale("ru", "RU");

Существует константа Locale.ROOT. Она представляет локаль, для которой язык, страна равны пустой строке (""). Эта локаль является базовой для всех остальных локалей. Используется для написания приложений, не зависящих от локали.

Определить текущий вариант региональных параметров можно следующим образом: Locale current = Locale.getDefault();

А можно и изменить для текущего экземпляра (instance) JVM: Locale.setDefault(Locale.CANADA).

Некоторые методы Locale: getCOuntry(), getDisplayedLanguage(), getLanguage, getDisplayCountry().

Класс ResourceBundle предназначен для чтения данных из текстовых файлов свойств (расширение - properties). Каждый отдельный файл с расширением properties содержит информацию для отдельной локали.

В файлах свойств (*.properties) информация должна быть организована по принципу:

```
#Комментарий
group1.key1 = value1
group1.key2 = value2
group2.key1 = value3…
```

Например:

```
label.button = submit
label.field = login
message.welcome = Welcome!
```

Рассмотрим правила выбора имени для properties файлов. Выбираем базовое имя для группы properties файлов. Например text. Добавляем к базовому имени через пробел код языка (uk) и код страны (UA).

Чтобы выбрать определенный объект ResourceBundle, следует вызвать один из статических перегруженных методов getBundle(параметры).

Следующий фрагмент выбирает text объекта ResourceBundle для объекта Locale, который соответствует французскому языку и стране Канада:

```
Locale locale = new Locale("fr", "CA");
ResourceBundle rb = ResourceBundle.getBundle("text", locale);
```

Если объект ResourceBundle для заданного объекта Locale не существует, то метод getBundle() извлечет наиболее общий.

Если общее определение файла ресурсов не задано, то метод getBundle() генерирует исключительную ситуацию MissingResourceException.

В классе ResourceBundle определен ряд полезных методов:

1. getKeys() - возвращает объект Enumeration, который применяется для последовательного обращения к элементам.
2. keySet() – возвращает множество Set всех ключей.
3. getString(String key) - извлекается конкретное значение по конкретному ключу.
4. boolean containsKey(String key) - проверить наличие ключа в файле.

Форматирование локализованных числовых данных, текста, даты и времени

Класс NumberFormat языка Java используется для форматирования чисел.

Чтобы получить объект класса для форматирования в национальном стандарте по умолчанию, используются следующие методы: NumberFormat.getInstance().

Чтобы получить объект класса для форматирования в других национальных стандартах используются следующие методы: NumberFormat.getInstance(Locale locale).

Некоторые методы:

- setMaximumFractionDigits(int digits) - устанавливает максимальное количество цифр после десятичной точки в форматируемом объекте. Последняя отображаемая цифра округляется.
- setMaximumIntegerDigits(int digits) - устанавливает максимальное количество цифр перед десятичной точкой в форматируемом объекте.
- Метод parse() класса NumberFormat преобразует строку к числу. Если перед вызовом метода parse() вызвать метод setParseIntegerOnly(true), как показано в следующем примере, то преобразовываться будет только целая часть числа.
- Метод String format(double Q) позволяет форматировать параметры в строку через NumberFormat.

Класс MessageFormat предназначен для создания строк. Данный класс Java принимает набор объектов, форматирует их, а затем вставляет форматированные строки в шаблон в соответствующих местах. Это своего рода альтернатива (или даже дополнение) к статическому методу String.format.

Constructor and Description
`MessageFormat(String pattern)` Constructs a MessageFormat for the default locale and the specified pattern.
`MessageFormat(String pattern, Locale locale)` Constructs a MessageFormat for the specified locale and pattern.

Статический метод MessageFormat.format, в который передаются аргументами шаблон строки и, собственно, объекты, которые будут вставлены в места, ограниченные скобками {}. В скобках задается позиция объекта начиная с 0, а также тип форматирования, если таковой имеется.

При создании объекта класса MessageFormat в его конструктор передается шаблон строки. Далее, при вызове метода format у объекта, туда в качестве аргумента передается массив объектов, которые будут вставлены в шаблон строки.

Также возможно сделать так, что в зависимости от значения переменной будет выбираться необходимый текст. Своего рода реализация оператора if...else, только с помощью класса ChoiceFormat.

SimpleDateFormat является подклассом ***DateFormat***, который позволяет форматировать ввод-вывод даты и времени в рамках предопределенных стилей. В отличие от ***DateFormat***, ***SimpleDateFormat*** позволяет создавать собственные настраиваемые форматы ввода-вывода.

Для создания экземпляра класса ***SimpleDateFormat*** используется один из 4 конструкторов:

```
SimpleDateFormat()
SimpleDateFormat(String pattern)
SimpleDateFormat(String pattern, DateFormatSymbols formatSymbols)
SimpleDateFormat(String pattern, Locale locale)
```

pattern – шаблон определяющий формат даты и времени
formatSymbols – символы формата даты (например название месяцев или дней недели)
locale — локаль

SimpleDateFormat чувствителен к локали. При создании экземпляра ***SimpleDateFormat*** без параметра ***Locale***, вывод будет форматироваться в соответствии с ***Locale*** по умолчанию.

Пакет java.time. Классы для представления даты и времени.

Для удобной работы с датой и временем в Java используются классы Date и Calendar.

Класс **Date** хранит время в миллисекундах начиная с 1 января 1970 года. Данный класс имеет конструктор по умолчанию, который возвращает текущее время. Кроме этого, можно создать объект Date используя конструктор, который принимает количество миллисекунд начиная с 1 января 1970 года. Для получения этого внутреннего времени используется метод **getTime()**. Кроме этого уже после создания экземпляра класса можно изменить время с помощью **setTime(long date)**. Некоторые методы:

- boolean after(Date date) - если объект содержит более позднюю дату, чем указано в параметре date, то возвращается true;
- boolean before(Date date) - если объект содержит более раннюю дату, чем указано в параметре date, то возвращается true.

Абстрактный класс **Calendar** позволяет работать с датой в рамках календаря, т.е он умеет прибавлять день, при этом учитывать високосные год и прочее, а также позволяет преобразовать время в миллисекундах в более удобном виде - год, месяц, день, часы, минуты, секунды. Единственной реализацией Calendar является класс **GregorianCalendar**, так же как и у даты конструктор по умолчанию возвращает календарь на текущий день, но можно задать его явно указав все параметры.

Календарь достаточно мощный класс, который позволяет получать названия месяцев и дней недели, увеличивать или уменьшать различные параметры текущей даты, а также получать их. Для удобства работы с ним нужно просто разобраться с типами данных, с которыми он работает:

- DAY_OF_YEAR — день года (0- 365);
- DAY_OF_MONTH — день месяца (какой по счету день в месяце 0 — 31);
- WEEK_OF_MONTH — неделя месяца;
- WEEK_OF_YEAR — неделя в году;
- MONTH — номер месяца;
- Year — номер года;
- Calendar.ERA — эра.

Некоторые методы:

- abstract void add(int field, int value) - добавляет value к компоненту времени или даты, указанному в параметре field (например, Calendar.HOUR). Чтобы отнять, используйте отрицательное значение.
- boolean after(Object calendar)
- boolean before(Object calendar)
- int get(int field) - возвращает значение одного компонента, например, Calendar.MINUTE
- synchronized static Locale[] getAvailableLocales() - возвращает массив объектов класса Locale, содержащий региональные данные
- final Date getTime()
- TimeZone getTimeZone()
- void set(int field, int value)

Класс TimeZone: Временная зона GMT и UTC

java.time — основной пакет для работы с датой и временем
- java.time.chrono — календарные системы
- java.time.format — классы для форматирования
- java.time.temporal — преобразования времени и вычисления
- java.time.zone — работа с поясным временем

Рефлексия. Классы Class, Field, Method, Constructor.

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

- Определить класс объекта. (Класс Class)
- Получить информацию о модификаторах класса, полях, методах, конструкторах и суперклассах. (Класс Field, Method, Constructor)
- Выяснить, какие константы и методы принадлежат интерфейсу.
- Создать экземпляр класса, имя которого неизвестно до момента выполнения программы.
- Получить и установить значение свойства объекта.
- Вызвать метод объекта.
- Создать новый массив, размер и тип компонентов которого неизвестны до момента выполнения программ.

Class.getFields() возвращает объекты типа Field – поля, которые нам доступны. Если поле private, то используем getDeclaredFileds() или getDeclaredField(fieldname) – возвращает и private и protected. Field.setAccessible(true) – разрешает работать с private и protected полями. Можем использовать get и set.

Class.getDectaredMethod(methodname) – возвращает метод. Method.invoke(Object, args) – вызов метода.

Функциональные интерфейсы и λ-выражения. Пакет java.util.function.

Перед функциональным интерфейсом ставиться аннотация @FunctionalInterface

Функциональный интерфейс - интерфейс, который содержит ровно один абстрактный метод. Кроме (static, default и абстрактных методов, которые переопределяют методы в Object)

Основу лямбда-выражения составляет лямбда-оператор ->. Он разделяет лямбда-выражение на список параметров и тело лямбда-выражения, где выполняются все действия.
Лямбда-выражение образует реализацию метода, определенного в функциональном интерфейсе.

```
interface Operationable{
    int calculate(int x, int y);
}
Operationable operation = (x, y) -> x + y;

Пакет java.util.function
Пакет, содержащий функциональные интерфейсы
```

Конвейерная обработка данных. Пакет java.util.stream.

Конвейерная обработка - способ выполнения команд процессором, при котором выполнение следующей команды начинается до полного окончания выполнения предыдущей команды.

Поток – последовательность элементов.

Конвейер – последовательность операций.

Отличия от коллекций: элементы не хранятся, неявная итерация, функциональный стиль (операции не меняют источник).

Конвейер состоит из источника (коллекция, массив, фабрика элементов, канал ввода-вывода, ...), нескольких (0 или больше) промежуточных операций и одной завершающей операции.

Способы получения потока из источника: Collection.stream(), Collection.parallelStream(), Stream.of(Object []).

В Java stream API есть 2 вида методов для работы со стримами:

- Промежуточные (возвращают стрим): filter, skip, distinct(возвращает стрим без дубликатов), map, peek(применяет функцию к каждому элементу), limit, sorted
- Терминальные (возвращают объект): find first, findAny, collect, count, anyMatch(true, если условие выполняется хотя бы для одного элемента), noneMatch(true, если условие не выполняется для всех элементов), min, max, forEach, toArray

Аннотации

Аннотации – это пометки, с помощью которых программист указывает компилятору Java и средствам разработки, что делать с участками кода помимо исполнения программы. Аннотировать можно переменные, параметры, классы, пакеты. Можно писать свои аннотации или использовать стандартные – встроенные в Джава. Вы узнаете аннотацию по символу @.

Они позволяют:

- автоматически создавать конфигурационные XML-файлы и дополнительный Java-код на основе исходного аннотированного кода;
- документировать приложения и базы данных параллельно с их разработкой;
- проектировать классы без применения маркерных интерфейсов;
- быстрее подключать зависимости к программным компонентам;
- выявлять ошибки, незаметные компилятору;
- решать другие задачи по усмотрению программиста.

Корректное хранение паролей в БД

Вместо пароля можно хранить его хеш. При вводе пароля вычисляется его хеш и сравнивается с хранящимся. Если они совпали, то считаем, что был введен верный пароль. Хешфункция должна быть необратимой, то есть не должно быть возможности расшифровать зашифрованный пароль. Также хеш-функция должна обеспечивать минимальное число коллизий. Есть разные алгоритмы хеширования: MD4, MD5, SHA-1, SHA-2 и т. д.

Хотя хеш-функции необратимы, но пароль можно подобрать. При наличии коллизий можно подобрать другой пароль, имеющий тот же самый хеш, тогда даже с неверным паролем можно войти в аккаунт

Как можно усилить защиту? Можно пытаться заставить пользователей выбирать сложные пароли. Но их сложно запоминать. Можно добавить к простому паролю, типа (12345 или hello) «соль» - некую случайную последовательность, которая тоже будет храниться в базе. Хеш тогда будет генерироваться из склеенных пароля и соли. Тогда в базе данных хранится соль и хеш комбинации пароля с солью. Когда пользователь вводит свой пароль, из базы берется соль, вычисляется хеш пароля с солью, и сравнивается с тем хешем, что хранится в базе.

Для каждого пользователя можно использовать свою соль, чтобы для одинаковых паролей получались разные хеши. Но и такой способ подвержен атакам. Так как соль в базе хранится практически в открытом виде, то хакер берет просто словарь возможных паролей, добавляет соль, и сравнивает с хешем.

Можно еще усилить защиту дополнительно добавив «перец». Перец - еще одна последовательность символов, которая хранится уже в коде приложения. В базе перец не хранится, чтобы узнать и соль и перец нужно иметь доступ и к базе, и к коду приложения. Соответственно, алгоритм может быть таким: приложение, получив пароль, получает из базы данных соль, добавляет соль и перец к паролю, получает хеш, и сравнивает его с хешем, который хранится в базе.

```
   java.security.MessageDigest
MessageDigest md = MessageDigest.getInstance("MD5");
String user   = Console.readLine()
String passwd = Console.readPassword();
String salt   = getRandomString();
String pepper = "*63&^mVLC(#"
byte[] hash = md.digest(
      (pepper + passwd + salt).getBytes("UTF-8"));

insert into users (user, salt, hash);
```