1. Объектно-ориентированное программирование. Основные понятия.

Объе́ктно-ориенти́рованное программи́рование (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 уже 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 — возможность объектов с одинаковой спецификацией иметь различную реализацию.
- 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).

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

Пакет java.lang - основной пакет в котором содержатся классы, соответствующие типам данных. Boolean, Byte, Character, Class, Double, Float, Integer, Long, Math, Number, Object ,Short, String, System, Thread

Object является суперклассом для всех классов.
Object clone()
boolean equals(Object obj)
void finalize() - вызывается перед удалением неиспользуемого объекта

Class<?> getClass() - получает класс объекта во время выполнения
int hashCode()
void notify() - возобновляет выполнение потока, который ожидает вызывающего объекта
void notifyAll()
String toString()
void wait()
void wait(long millis)
void wait(long millis, int nanos)

3. Класс Number. Классы-оболочки. Автоупаковка и автораспаковка.

Абстрактный класс Number является суперклассом для классов Byte, Double, Float, Integer, Long и Short. Подклассы числа 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. Все классы-оболочки в Java являются неизменными и final.

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

4. Математические функции. Класс Math. Пакет java.math
`java.lang.Math` содержит две константы -- `static double Math.E` и `static double Math.PI`.
Математические операции также статические:

- 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)`.

5. Работа со строками. Классы String. Классы StringBuilder и StringBuffer.
`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`.

```
StringBuilder sb = new StringBuilder(); sb.append(“s”); sb.append(“tr”);
```

`String s = sb.toString();`
StringBuilder изменяем (при вызове `append` не создаются новые объекты `String`),поэтому он эффективнее, чем конкатенация в цикле. Внутри содержит массив символов, размер которого изменяется по необходимости.
`StringBuffer` работает аналогично, только его методы помечены `synchronized`, поэтому он потокобезопасный. Это делает его более медленным, поэтому он почти не используется -- просто он появился раньше, чем `StringBuilder`.

6. Обработка исключительных ситуаций. Классы Error, Exception, RuntimeException
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`. Примеры: `OutOfMemoryError` (нехватка памяти), `StackOverflowError` (слишком глубокая рекурсия)

`RuntimeException` -- наследуется от `Exception`, unchecked исключение, которое не обязательно обрабатывать в `try-catch`, но можно. Пример: `NullPointerException`

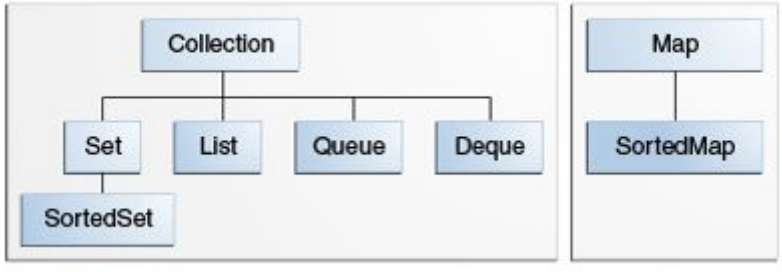
`Exception` -- исключения, которые должны или обрабатываться в `try-catch`, или быть указанными в сигнатуре метода (`String readFile() throws IOException`).

7. 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`.

8. Коллекции. Виды коллекций. Интерфейсы Set, List, Queue.



Интерфейсы -- см. 11, 12, 13

9. Обход элементов коллекции. Интерфейс Iterator.
Цикл for-each:
```
for (Object o : collection)
    System.out.println(o);
```

Итератор

```
public interface Iterator<E> {
    E next(); // вернуть следующий элемент
    boolean hasNext(); // true, если остались элементы и можно вызвать next()
    void remove(); // удалить текущий элемент
}
```

Во время итерации нельзя изменять коллекцию (добавление, удаление) не через `remove()`.

10. Сортировка элементов коллекций. Интерфейсы Comparable и Comparator.

Натуральный порядок определен для элементов, классы которых реализуют интерфейс `Comparable` (к примеру, `Integer` сортируется от меньшего к большему).

```
class MyClass implements Comparable<MyClass> {
   @Override
   public int compareTo(MyClass other) {
       if (this.something < other.something) {
           return -1;
       }
       ...
   }
}
```

`compareTo` должен возвращать отрицательное число (обычно -1), если `this` объект меньше `other`, положительное число (обычно 1), если `this` больше `other`, и 0, если они равны.

С помощью `Comparator` можно задавать отличный от натурального порядок элементов, или же сортировать объекты, которые не реализуют `Comparable`. Интерфейс содержит единственный метод `int compare(T o1, T o2)`, который возвращает отриц. число, 0, положит. число, если `o1` меньше, равен, больше `o2`.

11. Интерфейс Set, его варианты и реализации.

`Set` не может содержать повторяющихся элементов (математическое множество). Два объекта `Set` считаются `equals()`, если они имеют одинаковую длину и содержат одинаковые элементы, вне зависимости от порядки вставки.

Реализации: `HashSet` работает быстрее, чем `TreeSet`, но у него отсутствует сортировка. `TreeSet` позволяет сортировать элементы *в натуральном порядке* или при помощи `Comparator`.

`LinkedHashSet` близок по скорости к операций к `HashSet` и сохраняет порядок, в котором элементы вставлены (от старых к новым).

12. Интерфейс List и его реализации. Особенности класса Vector.

`List` -- отсортированная коллекция, которая в отличии от `Set` может содержать повторяющиеся элементы. 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.

13. Интерфейс Map, его варианты и реализации.

`Map<K, V>` определяется двумя типами (ключ и значение), поэтому не наследуется от `Collection<T>`, но также является частью Java Collections Framework.

Реализации `HashMap, TreeMap, LinkedHashMap` схожи с реализациями `Set` (отличаются

быстродействием и наличием сортировки).

14. Классы Collections и Arrays.

`Collections` содержит статические методы для работы с коллекциями, а именно `sort(List<T>)` и `sort(List<T>, Comparator<T>)`, `reverse(List<T>)` 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<T>.toArray()` (`set.toArray()`, `list.toArray()`, …)

15. Обобщенные и параметризованные типы. Создание параметризованных классов.

Благодаря generics, один и тот же алгоритм не нужно реализовывать для каждого типа данных, достаточно использовать параметризованный тип.
Это особенно важно для коллекций: типы проверяются на этапе компиляции, поэтому не требуется их приведение:

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0); // get возвращает Object
list.add(1);
String s = (String) list.get(1); // ошибка во время выполнения

List<String> list = new ArrayList<>();
list.add("hello");
String s = list.get(0); // get возвращает String
list.add(1); // ошибка на этапе компиляции
```

Простейший класс:

```
public class Box<T> {
    private T t;
    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

Важно помнить, что на этапе компиляции обобщенные типы стираются (второй пример с `List<String>` превратится в первый, где в `List` хранятся `Object`, а приведение типа вставляется компилятором).

16. 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
```

На тип можно наложить ограничение сверху (разрешены только тип и его подтипы, `<T extends Class>`) и снизу (разрешены только тип и то, от чего он наследуется, `<T super Class>`).

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

Класс OutputStream - абстрактный класс, определяющий потоковый байтовый вывод.
Класс InputStream - абстрактный класс, определяющий потоковый байтовый ввод.
Методы класса: int close(), void flush(),
abstract void write (int oneByte) // read(int oneByte) - записывает/считывает единственный байт в выходной поток (возвращает (int)-1, если достигнут конец файла)
void write (byte[] buffer) - записывает полный массив байтов в выходной поток,
void write (byte[] buffer, int offset, int count) - записывает диапазон из count байт из массива, начиная с смещения offset.

Символьные потоки определены в двух иерархиях классов. Наверху этой иерархии два абстрактных класса: Reader и Writer. Они определяют ключевые методы read() и write()

IO: потокоориентированный, блокирующий (синхронный) ввод/вывод

18. 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.

19. Работа с файлами в Java. Классы java.io.File и java.nio.Path.
Интерфейс Path(nio) - описывает расположение файлов в структуре каталогов.
методы:
Path getFileName()
Path getName(int индекс)
int getNameCount()
Path getParent()
Path getRoot()

Класс File - предоставляет статические методы для работы с файлом. Путь к файлу задается объектом типа Path. В классе File имеются методы для открытия, создания файла, можно получить сведения о файле (исполняемый? скрытый?). Определены методы для копирования и перемещения, list(), walk(), lines(), find() - возвращают Stream

20. Сериализация объектов. Интерфейс Serializable.
При сериализации объект переводится в поток байт, который может быть передан по сети, а затем десериализован, то есть переведен обратно в объект. Классы, объекты которых могут быть сериализованы, должны реализовывать маркер-интерфейс `Serializable` (маркер -- необязательно что-либо переопределять).
Если класс содержит поля типа, который не может быть сериализован, при попытке его сериализации будет брошено исключение.
При записи объекта сохраняется уникальный номер класса (`serialVersionUID`), который меняется при изменении его полей. Если сохраненный номер не совпадает с номером класса, который загружен в приложении, будет брошено исключение.
Класс может переопределить стандартную сериализацию Java (применяется в случаях, если не все

его поля можно сериализовать), переопределив методы `writeObject(ObjectOutputStream oos)` и `readObject(ObjectInputStream ois)`.

21. Библиотеки графического интерфейса. Особенности и различия.

AWT -- набор независимых от платформы виджетов (widgets), имеющих имплементацию для каждой платформы (peers).

Для передачи событий, источники (event sources) -- то есть компоненты, напр. Button, -- содержат в себе список слушателей (event listeners), которые вызываются в нужный момент. Для разных типов событий (нажатие кнопки мыши, ввод текста) есть разные классы слушателей, которые добавляются через разные методы (`addMouseListener`, `addActionListener`, etc.).

Недостаток AWT -- компоненты отрисовываются на каждой платформе по-своему, поэтому одна и та же программа выглядит по-разному на разных системах и вполне может иметь баги, которые появляются только на определенной ОС. В тоже время, нативная имплементация отрисовки компонентов конкретной ОС ведет к некоторому увеличению производительности (не используем посредников в виде JVM)

Swing
SWT
JavaFX

22. Библиотека Swing. Особенности.
В Swing виджеты отрисовываются не системными библиотеками, а в самой Java, поэтому они выглядят и работают одинаково на разных ОС. У некоторых компонентов Swing есть прямые аналоги из AWT (`JButton` <-> `Button`), другие, более сложные, есть только в Swing (`JTree`).

23. Библиотека SWT. Особенности.

SWT библиотека - кросс-платформенная оболочка для графических библиотек конкретных операционных систем. SWT разработана с использованием стандартного языка Java и получает доступ к специфичным библиотекам различных ОС через JNI (Java Native Interface), который используется для доступа к родным визуальным компонентам операционной системы.

Библиотека SWT поддерживает большинство популярных операционных систем. Также существует возможность компиляции SWT java приложений в нативный бинарный код, что повышает производительность созданных приложений и не требует установки JRE.

SWT использует: widget («виджет») - это элемент графического интерфейса, имеющий стандартный внешний вид и выполняющий стандартные действия. Под виджетом подразумеваются окно (диалоговое, модальное), кнопка (стандартная, радиокнопка, флаговая), список (иерархический, раскрывающийся) и т.д.

JFace — набор классов, реализующих наиболее общие задачи построения GUI.JFace

представляет собой дополнительный программный слой над SWT, который реализует шаблон Model-View-Controller.

24. Библиотека JavaFX. Особенности.

JavaFX был создан как последователь Swing. Он включает поддержку CSS, что упрощает кастомизацию интерфейса и облегчает создание переключаемых тем. Для разметки интерфейса можно использовать XML с визуальным редактором Scene Builder, что позволяет разделить логику компонентов и разметку интерфейсов, удовлетворяя Single Responsibility принципу. Анимацию и специальные эффекты тоже проще делать в JavaFX, чем в Swing, так как там присутствует высокоуровневая их реализация - достаточно задать объект, начальное состояние, конечное состояние и длительность перехода (классы Transition).

Окно в JavaFX представлено классом `Stage`, компоненты содержатся в `Scene`. Сцены могут меняться, но в одно время в `Stage` может быть только одна сцена. Сцена содержит граф всех элементов (групп, компонентов), которые наследуются от класса `Node`. В отличии от Swing, менеджеры компоновки не отделены от контейнеров и тоже наследуются от `Node`. Еще одно важное отличие -- наличие свойств (properties), на которые можно добавить слушатели.

25. Компоненты графического интерфейса. Класс Component.

`Component` -- абстрактный класс-родитель для компонентов AWT. Имеет абстрактный метод `paint`, в котором наследники определяют, как отрисовывается компонент, и метод `repaint`, позволяющий пользователю вызвать перерисовку.
Класс наследуется виджетами `Button`, `Label`, `TextField`, `TextArea`, `Checkbox`, `ComboBox`, `PasswordField`, `ProgressBar`, `RadioButton`, `Slider`, `Spinner`, ``Table , `TextArea`, `Tree`

26. Размещение компонентов в контейнерах. Менеджеры компоновки.

Контейнеры верхнего уровня:
JApplet - главное окно апплета;
JFrame - окно приложения;
JDialog - диалог приложения.
JColorChooser - диалог выбора цвета;
JFileChooser - диалог выбора файлов и директорий;
FileDialog - диалог выбора файлов и директорий

Простые контейнеры:
JPanel - простая панель для группировки элементов, включая вложенные панели;
JToolBar - панель инструментов (обычно это кнопки);
JScroolPane - панель прокрутки, позволяющая прокручивать содержимое дочернего элемента;
JDesktopPane - контейнер для создания виртуального рабочего стола или приложений на основе MDI (multiple-document interface);
JEditorPane, JTextPane - контейнеры для отображения сложного документа как HTML или RTF;
JTabbedPane - контейнер для управления закладками;
JSplitPane - контейнер, разделяющий два элемента и позволяющий пользователю изменять

их размер.

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.

27. 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`а, 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.

28. Многопоточные программы. Класс Thread и интерфейс Runnable.

Один поток – это одна единица исполнения кода. Каждый поток последовательно выполняет инструкции процесса, которому он принадлежит, параллельно с другими потоками этого процесса. Параллельные потоки независимы друг от друга, за исключением тех случаев, когда программист сам описывает зависимости между потоками с помощью предусмотренных для этого средств языка. Для того, чтобы быть потоком, класс должен либо наследоваться от класса Thread, либо имплементировать интерфейс Runnable. В любом случае, логика, выполняемая потоком, задается переопределением метода run(), а начало исполнения - вызовом метода start()
Помимо этих методов, потоки имеют также:

- start() Этот метод запускает выполнение потока, а JVM (виртуальная машина Java) вызывает в потоке метод Run ().
- Sleep(int milliseconds) Делает поток спящим. Его выполнение будет приостановлено на указанное количество миллисекунд, после чего он снова начнет выполняться. Этот метод полезен при синхронизации потоков.
- getName() Возвращает имя потока.
- setPriority(int newpriority) Изменяет приоритет потока.
- yield () Останавливает текущий поток и запускает другие.

29. Состояние потока. Синхронизация потока. Модификатор synchronized.

Поток в Java может находиться в одном из следующих состояний:

- New: от создания до вызова
- Runnable: после вызова start(), переходит под контроль планировщика потоков

- Running: поток выполняется
- Blocked: поток находится в wait set, после вызова wait(), ждем notify (/all)
- Waiting: поток не смог захватить монитор, находится в Entry set под контролем планировщика ОС
- Dead: поток закончил своё выполнение

Концепция многопоточности предполагает проблему соперничества потоков за разделяемые ресурсы, то есть, работа одного потока может повлиять на валидность данных, выбранных другим потоком
Для решения проблемы соперничества потоков фактически все многопоточные схемы синхронизируют доступ к разделяемым ресурсам. Это означает, что доступ к разделяемому ресурсу в один момент времени может получить только один поток. Чаще всего это выполняется помещением фрагмента кода в секцию блокировки так, что одновременно пройти по этому фрагменту кода может только один поток. Поскольку такое предложение блокировки дает эффект взаимного исключения, этот механизм часто называют мьютексом (MUTual Exclusion).
В Java есть встроенная поддержка для предотвращения конфликтов в виде ключевого слова synchronized. Каждый объект в Java имеет ассоциированный с ним монитор. Когда поток желает выполнить фрагмент кода, охраняемый словом synchronized, он проверяет, доступен ли монитор, получает доступ к монитору, выполняет код и освобождает монитор. В один момент времени монитор может быть захвачен одним потоком
synchronized может быть метод или блок кода

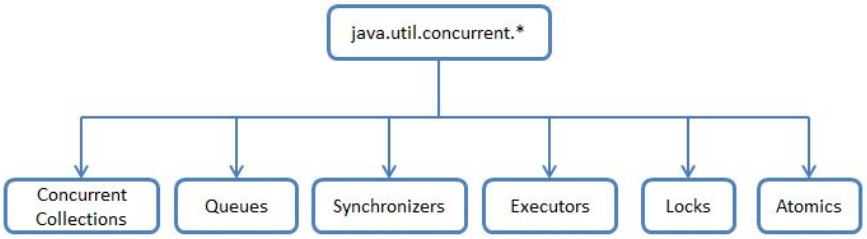
30. Взаимодействие потоков. Методы wait() и notify().

Иногда при взаимодействии потоков встает вопрос о извещении одних потоков о действиях других. Например, действия одного потока зависят от результата действий другого потока, и надо как-то известить один поток, что второй поток произвел некую работу. И для подобных ситуаций у класса Object определено ряд методов:

- wait(): освобождает монитор и переводит вызывающий поток в состояние ожидания до тех пор, пока другой поток не вызовет метод notify()
- notify(): продолжает работу потока, у которого ранее был вызван метод wait()
- notifyAll(): возобновляет работу всех потоков, у которых ранее был вызван метод wait()

Все эти методы вызываются только из синхронизированного контекста - синхронизированного блока или метода.

31. Пакет java.util.concurrent. Интерфейс Lock и его реализации.



Concurrent Collections — набор коллекций, более эффективно работающие в многопоточной среде нежели стандартные универсальные коллекции из java.util пакета. Вместо базового враппера Collections.synchronizedList с блокированием доступа ко всей коллекции используются блокировки по сегментам данных или же оптимизируется работа для параллельного чтения данных по wait-free алгоритмам.
Queues — неблокирующие и блокирующие очереди с поддержкой многопоточности. Неблокирующие очереди заточены на скорость и работу без блокирования потоков. Блокирующие очереди

используются, когда нужно «притормозить» потоки «Producer» или «Consumer», если не выполнены какие-либо условия, например, очередь пуста или перепонена, или же нет свободного «Consumer»'а.
Synchronizers — вспомогательные утилиты для синхронизации потоков. Представляют собой мощное оружие в «параллельных» вычислениях.
Executors — содержит в себе отличные фрейморки для создания пулов потоков, планирования работы асинхронных задач с получением результатов.
Locks — представляет собой альтернативные и более гибкие механизмы синхронизации потоков по сравнению с базовыми synchronized, wait, notify, notifyAll.
Atomics — классы с поддержкой атомарных операций над примитивами и ссылками.
Классы блокировок реализуют интерфейс Lock, который определяет следующие методы:

- void lock(): ожидает, пока не будет получена блокировка
- void lockInterruptibly() throws InterruptedException: ожидает, пока не будет получена блокировка, если поток не прерван
- boolean tryLock(): пытается получить блокировку, если блокировка получена, то возвращает true. Если блокировка не получена, то возвращает false. В отличие от метода lock() не ожидает получения блокировки, если она недоступна
- void unlock(): снимает блокировку
- Condition newCondition(): возвращает объект Condition, который связан с текущей блокировкой

Организация блокировки в общем случае довольно проста: для получения блокировки вызывается метод lock(), а после окончания работы с общими ресурсами вызывается метод unlock(), который снимает блокировку.
Объект Condition позволяет управлять блокировкой.

32. 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).

33. 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, представляющий 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»

34. 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 одного 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 отличие 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.

35. Шаблоны проектирования. Поведенческие шаблоны.

Поведенческие шаблоны проектирования определяют общие закономерности 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 обработки 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 реализуется 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().

36. Сетевое взаимодействие. Основные протоколы, их сходства и отличия.

37. 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.

38. Протокол UDP. Классы DatagramSocket и DatagramPacket.

UDP – транспортный протокол, передающий сообщения-датаграммы без необходимости установки соединения в IP-сети. (Голосовые данные, видео)

UDP обеспечивает более высокую скорость передачи данных.
UDP предпочтительнее для программ, воспроизводящих потоковое видео, видеофонии и телефонии, сетевых игр.
UPD не содержит функций восстановления данных
UDP не гарантирует доставку пакетов данных в неизменных виде, последовательности и без потерь
UDP не нумерует пакеты при передаче
UDP соединения не требует, у него это просто поток данных

Для передачи по протоколу UDP отправитель и получатель создают сокеты типа DatagramSocket. Этот класс содержит методы send(DatagramPacket pack) - отправляет пакет и receive(DatagramPacket pack) - дожидается получения и заносит его в пакет

39. Взаимодействие с базами данных. Протокол JDBC. Основные элементы.

JDBC (Java DataBase Connectivity) — платформенно-независимый промышленный стандарт взаимодействия Java-приложений с различными СУБД.
JDBC основан на концепции так называемых драйверов, позволяющих получать соединение с базой данных по специально описанному URL. Драйверы могут загружаться динамически (во время работы программы). Загрузившись, драйвер сам регистрирует себя и вызывается автоматически, когда программа требует URL, содержащий протокол, за который драйвер отвечает.

Основные элементы:
Менеджер драйверов (Driver Manager)
  Этот элемент управляет списком драйверов БД. Каждой запрос на соединение требует соответствующего драйвера. Первое совпадение даёт нам соединение.
Драйвер (Driver)

Этот элемент отвечает за связь с БД. Работать с ним нам приходится крайне редко. Вместо этого мы чаще используем объекты DriverManager, которые управляют объектами
этого типа.

Соединение (Connection)

Этот интерфейс обеспечивает нас методами для работы с БД. Все взаимодействия с БД происходят исключительно через Connection.

Выражение (Statement)

Для подтверждения SQL-запросов мы используем объекты, созданные с использованием этого интерфейса.

Результат (ResultSet)

Экземпляры этого элемента содержат данные, которые были получены в результате выполнения SQL – запроса. Он работает как итератор и “пробегает” по полученным данным.

Исключения (SQL Exception)

Этот класс обрабатывает все ошибки, которые могут возникнуть при работе с БД.

40. Создание соединения с базой данных. Класс DriverManager. Интерфейс DataSource.

Для подключения к базе данных среда выполнения Java должна загрузить соответствующий драйвер указанной базы данных. Загрузка и выгрузка таких драйверов осуществляется с помощью класса DriverManager.

DriverManager - это синглтон, который содержит информацию о всех зарегистрированных драйверах. Метод getConnection на основании параметра URL находит java.sql.Driver соответствующей базы данных и вызывает у него метод connect.

JDBC DataSource объекты используются для получения физического соединения с базой данных и являются альтернативой DriverManager. При этом нет необходимости регистрировать драйвер. Необходимо только установить соответствующие параметры для установки соединения и выполнить метод getConnection().

https://pastebin.com/ZEgwBzmy

41. Создание запросов. Интерфейсы Statement, PreparedStatement, CallableStatement.

Statement, prepared statement, callable statement

Объект Statement используется для выполнения SQL-запросов к БД. Существует три типа объектов Statement. Они специализируются на различных типах запросов: Statement используется для выполнения простых SQL-запросов без параметров; PreparedStatement используется для выполнения прекомпилированных SQL-запросов с или без входных (IN) параметров; CallableStatement используется для вызовов хранимых процедур.

SQL-выражения в PreparedStatement могут иметь один или более входной (IN) параметр. Входной параметр - это параметр, чье значение не указывается при создании SQL-выражения. Вместо него в выражении на месте каждого входного параметра ставится знак ("?"). Значение каждого вопросительного знака устанавливается методами setXXX

перед выполнением запроса.

42. Обработка результатов запроса. Интерфейсы ResultSet и RowSet.

Интерфейс java.sql.ResultSet представляет собой множество результатов запроса в БД. Экземпляр ResultSet имеет указатель, который указывает на текущую строку в полученном множестве и имеет методы получения, изменения данных, и методы навигации
Курсор двигается на основе свойств: ResultSet.TYPE-FORWARD_ONLY - только вперед, ResultSet.TYPE_SCROLL_INTENSIVE - вперед и назад, не чувствителен к изменениям в бд, ResultSet.TYPE_SCROLL_SENSITIVE - вперед и назад, чувствителен к изменениям в бд

Интерфейс RowSet расширяет интерфейс ResultSet, но набор строк не привязан к соединению с базой данных. Достаточно создать Rowset, указать ему на ResultSet, и когда он заполнится, использовать его вместо ResultSet

- CachedRowSet — это просто Rowset без подключения;
- WebRowSet — это подкласс CachedRowSet, который "знает", как преобразовать свои результаты в XML и обратно;
- JoinRowSet — это WebRowSet, который также "умеет" формировать эквивалент SQL JOIN без необходимости подключения к базе данных;
- FilteredRowSet — это WebRowSet, способный еще и отфильтровать полученные данные без необходимости подключения к базе данных.

Реализации Rowset, как правило, предоставляются драйвером JDBC

43. Интернационализация. Локализация. Хранение локализованных ресурсов.

Интернационализа́ция — 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

Хранение локализованных ресурсов:
ListResourceBundle
ListResourceBundle абстрактный подкласс ResourceBundle, который управляет ресурсами для локализации используя List, находящийся в классе. Ресурсы хранятся в классе-наследнике ListResourceBundle, в котором необходимо переопределить getContents и создать массив, в котором каждому элементу соответствует пара объектов “ключ”-”значение”.

```
public class MyResources extends ListResourceBundle {
        protected Object[][] getContents() {
                return new Object[][] = { /*...*/ }
        }
}
public class MyResources_fr extends ListResourceBundle {
    protected Object[][] getContents() {
```

```
        return new Object[][] = { /*...*/}
    }
}
```

PropertyResourceBundle

PropertyResourceBundle абстрактный подкласс ResourceBundle, который управляет ресурсами для локализации, используя ряд статических строк из файла свойств. ResourceBundle.getBundle будет автоматически искать соответствующий файл свойств и создавать а PropertyResourceBundle

44. Форматирование локализованных числовых данных, текста, даты и времени.

Форматирование числовых данных:
Класс java.text.NumberFormat: абстрактный класс, экземпляры можно получить с помощью методов getInstance(), getInstance(Locale inLocale)

Полученный экземпляр NumberFormat можно использовать для форматирования чисел с помощью метода format и парсинга чисел с помощью метода parse:

Класс java.text.DecimalFormat расширяет класс java.text.NumberFormat. Содержит дополнительно два конструктора:
public DecimalFormat(String pattern)
public DecimalFormat(String pattern, DecimalFormatSymbols symbols)
DecimalFormatSymbols, позволяет полностью настроить форматирование числа.

Форматирование даты и времени:
Класс java.text.DateFormat.
Для получения экземпляра класса:
public static final DateFormat getInstance()
public static final DateFormat getDateInstance() // дата без времени
public static final DateFormat getDateInstance(int style*)
public static final DateFormat getDateInstance(int style, Locale locale)

public static final DateFormat getTimeInstance() //время без даты
public static final DateFormat getDateTimeInstance() //и дата и время

*Style: DateFormat.FULL, DateFormat.LONG, DateFormat.SHORT, DateFormat.MEDIUM, DateFormat.DEFAULT.

Форматирование и парсинг происходит с помощью методов format и parse .

Класс java.text.SimpleDateFormat наследуется от java.text.DateFormat и позволяет указать пользовательский шаблон форматирования. (G - эра, y - год, …)

Форматирование текста:
Класс MessageFormat: форматирует текст, содержащий фрагменты, представленные в виде переменных.

Solar_en.properties
msg = {0} solar eclipse will be at {1,time,short} on {1,date,short}.
Статический метод MessageFormat.format() позволяет подставить значение переменных.

Класс ChoiceFormat:

```
    double limits[] = { 0.0, 0.1, 0.3, 0.7 };
    String labels[] = { "very low", "low", "moderate", "high" };
    ChoiceFormat format = new ChoiceFormat(limits, labels);
    System.out.println(format.format(r) + " (" + r + ").");
//r == 0.0    very low(0.0)
//r == 0.1     low(0.1)
//r == 0.3    moderate(0.3)
//r == 0.7    high(0.7)
```

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

Классы для представления даты и времени:
Класс Date:
Методы: boolean after(Date date), boolean before(Date date), long getTime() - возвращает количество миллисекунд, прошедших с 1 января 1970 года

Почти все методы объявлены deprecated

Класс: Calendar: Абстрактный класс — преобразование из машинных в человеческие единицы

	год	месяц	день	час	минута	секунда	нано
Year	x						
YearMonth	x	x					
MonthDay		x	x				
LocalDate	x	x	x				
LocalTime				x	x	x	x
LocalDateTime	x	x	x	x	x	x	x

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

46. 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.

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

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
```

48. 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

Stream -способ обрабатывать структуры данных в функциональном стиле.
В Java stream API есть 2 вида методов для работа со стримами:
конвеерные(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