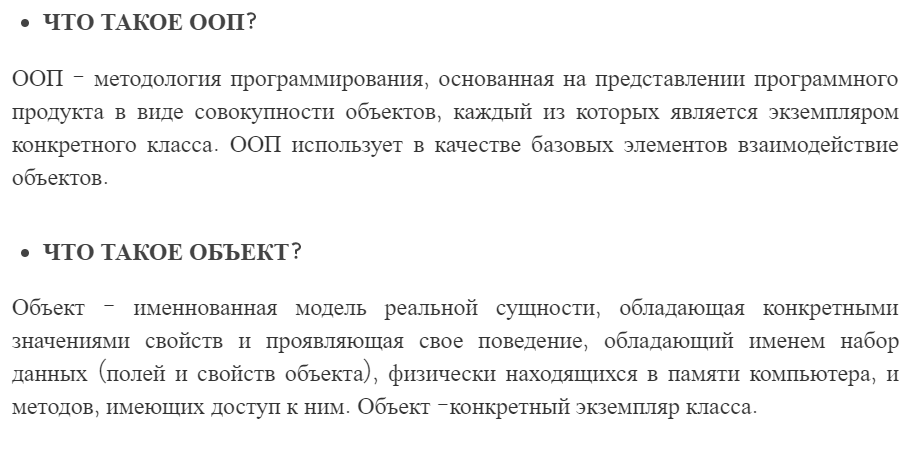
1. Объектно-ориентированное программирование. Основные понятия.

Java является объектно-ориентированным языком. Это означает, что писать программы на Java нужно с применением объектно-ориентированного стиля. И стиль этот основан на использовании в программе объектов и классов. Состав пакета java.lang. Класс Object и его методы.

Абстракция, Инкапсуляция, Наследование, Полиморфизм

* Наследование - новый класс, наследующий свойства базового (родительского) класса имеет все те свойства, которым обладает родитель. В коде используется операнд extends, после которого указывается имя базового класса. Тем самым открывается доступ ко всем полям и методам базового класса.
* Инкапсуляция означает, что поля объекта недоступны его клиентам непосредственно - они скрываются от прямого доступа извне. Инкапсуляция предохраняет данные объекта от нежелательного доступа, позволяя объекту самому управлять доступом к своим данным.
* Полиморфизм - положение теории типов, согласно которому имена (например, переменных) могут обозначать объекты разных, но имеющих общего родителя, классов. Следовательно, любой объект, обозначаемый полиморфным именем, может посвоему реагировать на некий общий набор операций.
* Абстракция реализуется с помощью интерфейсов и абстрактных классов.
  + Абстрактным называют класс, который не может иметь экземпляров. Также, он может содержать абстрактные методы (методы без реализации).
  + Интерфейс – это набор абстрактных методов. Классы, которые имплементируют интерфейсы (реализовывают) наследуют все методы интерфейса и должны их реализовать. Написание интерфейса аналогично написанию класса, отличие заключается в том, что интерфейс описывает только поведение объекта, которое должно быть реализовано классом.

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

В состав пакета java.lang входят классы, составляющие основу для всех других, и поэтому он является наиболее важным из всех, входящих в Java API. Поскольку без него не может обойтись ни один класс, каждый модуль компиляции содержит неявное импортирование этого пакета ( import java.lang.\*; ).

Перечислим классы, составляющие основу пакета.

* Object – является корневым в иерархии классов.
* Class – экземпляры этого класса являются описаниями объектных типов в памяти JVM.
* String – представляет собой символьную строку, содержит средства работы с нею.
* StringBuffer – используется для работы (создания) строк.
* Number – абстрактный класс, являющийся суперклассом для классов-объектных оберток числовых примитивных типов Java.
* Character – объектная обертка для типа char.
* Boolean – объектная обертка для типа boolean.
* Math – реализует набор базовых математических функций.
* Throwable – базовый класс для объектов, представляющих исключения. Любое исключение, которое может быть брошено и, соответственно, перехвачено блоком catch, должно быть унаследовано от Throwable.
* Thread – позволяет запускать и работать с потоками выполнения в Java. Runnable – может использоваться в сочетании с классом Thread для описания потоков выполнения.
* ThreadGroup – позволяет объединять потоки в группу и производить действия сразу над всеми потоками в ней. Существуют ограничения по безопасности на манипуляции с потоками из других групп.
* System – содержит полезные поля и методы для работы системного уровня.
* Runtime – позволяет приложению взаимодействовать с окружением, в котором оно запущено.
* Process – представляет интерфейс к внешней программе, запущенной при помощи Runtime.
* ClassLoader – отвечает за загрузку описания классов в память JVM.
* SecurityManager – для обеспечения безопасности накладывает ограничения на данную среду выполнения программ.
* Compiler – используется для поддержки Just-in-Time компиляторов.
* Интерфейсы:
* Cloneable – должен быть реализован объектами, которые планируется клонировать с помощью средств JVM;
* Comparable – позволяет упорядочивать (сортировать, сравнивать) объекты каждого класса, реализующего этот интерфейс.

Класс Object является базовым для всех остальных классов. Он определяет методы, которые поддерживаются любым классом в Java.

* getClass() возвращает объект типа Class
* equals(Object obj)
* toString() возвращает строковое представление объекта.
* hashCode() возвращает значение хэш-кода для объекта.
* Wait, notify, notifyAll

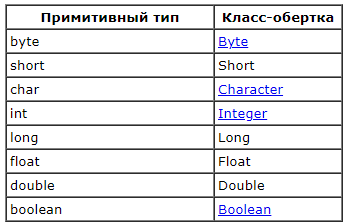
1. Класс Number. Классы-оболочки. Автоупаковка и автораспаковка.

Абстрактный класс Number является суперклассом для классов Byte, Double, Float, Integer, Long и Short. Наследники Number должны обеспечить методы преобразовывания числовых значений в byte, double, float, int, long и short.

Часто бывает предпочтительней работать с объектами, а не с примитивными типами. Так, например, при использовании коллекций, просто необходимо значения примитивных типов каким-то образом представлять в виде объектов. Для этих целей и предназначены так называемые классы-обертки .

Для каждого примитивного типа Java существует свой класс-обертки. Такой класс является неизменяемым, то есть, для изменения значения необходимо создавать новый объект. К тому же класс-обертка имеет атрибут final и его нельзя наследовать.

Классы-обертки содержат статическое поле TYPE - содержащее объект Class, соответствующий примитивному оборачиваемому типу. Также классы-обертки содержат статические методы для обеспечения удобного манипулирования соответствующими примитивными типами, например, преобразование к строковому виду.



Wrapper classes числовых типов (Byte, Short, Integer, Long, Float, Double) наследуются от класса Number, который содержит код, общий для всех классов-оберток числовых типов.

Изображение выглядит как текст

Автоматически созданное описание

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

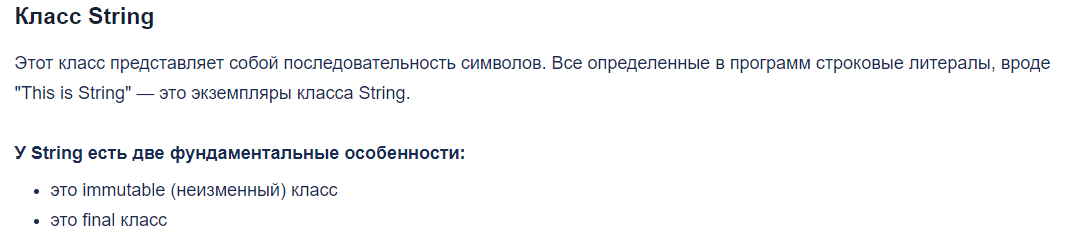
Класс Math располагается в пакете java.lang и предоставляет набор статических методов для осуществления ряда различных математических вычислений.

Ниже приведены примеры вычислений, для которых класс Math может оказаться полезным:

* ABS - модуль
* Тригонометрических функций (синусов, косинусов и т.д.) double
* POW – возведение в степень double
* SQRT – корень double
* RANDOM Генерация случайных чисел
* ROUND Округления

Если вам не хватает точности основных типов для представления целых и вещественных чисел, то можно использовать классы BigInteger и BigDecimal из пакета java.math, которые предназначены для выполнения действий с числами, состоящими из произвольного количества цифр.

1. Работа со строками. Классы String. Классы StringBuilder и StringBuffer.

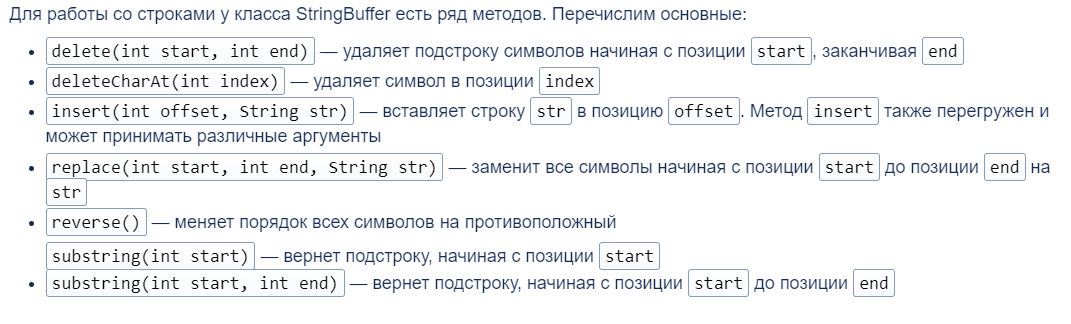


Не может иметь потомков, его объекты не могут быть изменены после создания. Любые операции над объектом String, результатом которых должен быть объект класса String, приведут к созданию нового объекта.

* + Благодаря неизменности, хэшкод экземпляра класса String кэшируется.
* +Класс String можно использовать в многопоточной среде без дополнительной синхронизации.
* + Еще одна особенность класса String — для него перегружен оператор "+" в Java.
* - большое количество мусора.

**Класс StringBuffer**

Чтобы справиться с созданием временного мусора из-за модификаций объекта String, можно использовать класс StringBuffer. Это mutable класс, т.е. изменяемый. Объект класса StringBuffer может содержать в себе определенный набор символов, длину и значение которого можно изменить через вызов определенных методов



+мусора не возникает.

- медленно, из-за синхронизации. Зато потокобезопасно

**Класс StringBuilder**

StringBuilder в Java — класс, представляющий последовательность символов. Он очень похож на StringBuffer во всем, кроме потокобезопасности.

Разница лишь в том, что StringBuffer потокобезопасен, и все его методы синхронизированы, а StringBuilder — нет. Это единственная особенность.

StringBuilder в Java работает быстрее StringBuffer’а благодаря несинхронизированности методов.

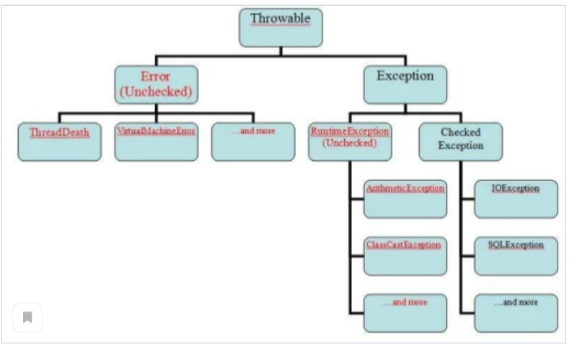
Поэтому в большинстве случаев, кроме многопоточной среды, лучше использовать для программы на Java StringBuilder.

1. Обработка исключительных ситуаций. Классы Error, Exception, RuntimeException.

В мире программирования возникновение ошибок и непредвиденных ситуаций при выполнении программы называют исключением В отличие от нашего мира, программа должна четко знать, как поступать в такой ситуации. Для этого в Java предусмотрен механизм исключений.

## Кратко о ключевых словах try, catch, finally, throws

**Обработка исключений в Java основана на использовании в программе следующих ключевых слов:**

* **try** – определяет блок кода, в котором может произойти исключение;
* **catch** – определяет блок кода, в котором происходит обработка исключения;
* **finally** – определяет блок кода, который является необязательным, но при его
* **throw** – используется для возбуждения исключения;
* **throws** – используется в сигнатуре методов для предупреждения, о том что метод может выбросить исключение.

Исключительные ситуации, возникающие в программе, можно разделить на две группы:

* Ситуации, при которых восстановление дальнейшей нормальной работы программы невозможно
  + К первой группе относят ситуации, когда возникают исключения, унаследованные из класса Error. (**OutOfMemoryError выбрасывается, когда виртуальная машина Java не может выделить (разместить) объект из-за нехватки памяти**) Это ошибки, возникающие при выполнении программы в результате сбоя работы JVM, переполнения памяти или сбоя системы. Обычно они свидетельствуют о серьезных проблемах, устранить которые программными средствами невозможно. Такой вид исключений в Java относится к неконтролируемым (unchecked) на стадии компиляции.
  + К этой группе также относят RuntimeException – исключения, наследники класса Exception, генерируемые JVM во время выполнения программы. Часто причиной возникновения их являются ошибки программирования. Эти исключения также являются неконтролируемыми (unchecked) на стадии компиляции, поэтому написание кода по их обработке не является обязательным. **ArrayIndexOfBoundException Это исключение выбрасывается когда вы пытаетесь зайти в элемент с недействительным индексом в массиве.**
* Восстановление возможно.
  + Ко второй группе относят исключительные ситуации, предвидимые еще на стадии написания программы, и для которых должен быть написан код обработки. Такие исключения являются контролируемыми (checked). Основная часть работы разработчика на Java при работе с исключениями – обработка таких ситуаций. **SQLException предоставляет информацию об ошибках доступа к базе данных или других ошибках связанных с работой с базами данных.**

1. Классы System и Runtime. Класс java.io.Console.

**System** является final, все поля и методы являются статическими (static), поэтому мы не можем создать подкласс и переопределить его методы используя наследование.Содержит набор методов для доступа к системным функциям, а также переменные in, out и err, представляющие соответственно стандартные потоки ввода вывода и ошибок.Поля

* static PrintStream err — это стандартный поток вывода ошибок.
* static InputStream in — это «стандартный» поток ввода.
* static PrintStream out — это «стандартный» поток вывода.

Методы

* Arraycopy - копирует элементы массива в другой массив
* Exit - выполняет выход из программы
* gc() - запускает сборщик мусора
* loadLibrary -загружает динамическую библиотеку
* currentTimeMillis - возвращает миллисекунды с 1 января 1970 г.

**Runtime** позволяет приложению взаимодействовать со средой, в которой оно запущено. Приложение не может создать свой собственный экземпляр этого класса.

Методы

* getRuntime() Возвращает соответсвующий приложению Runtime
* exit() Осуществляет завершение программы с кодом завершения status
* gc() Сигнализирует сборщику мусора о необходимости запуска
* freeMemory() Возвращает количество свободной памяти.
* totalMemory() Возвращает суммарное количество памяти, выделенное Java машине.

Console не получает консольный ввод-вывод сам по себе, а использует уже имеющиеся потоки System.in и System.out. Но в то же время Console значительно упрощает ряд операций, связанных с консолью. Основные методы класса

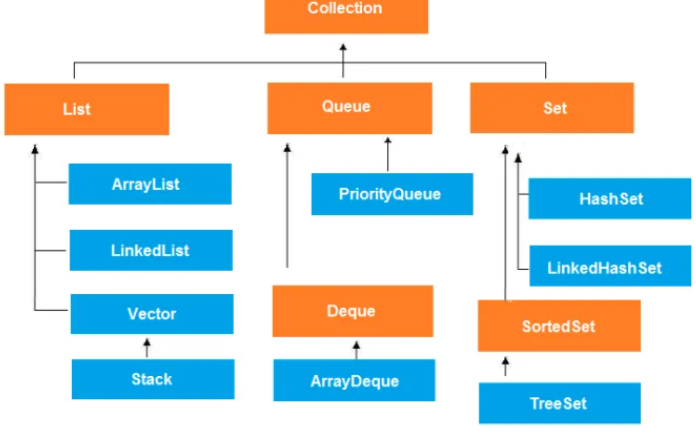
* flush(): выводит на консоль все данные из буфера
* format(): выводит на консоль строку с использованием форматирования
* printf(): выводит на консоль строку с использованием форматирования (фактически то же самое, что и предыдущий метод)
* String readLine(): считывает с консоли введенную пользователем строку
* char[] readPassword(): считывает с консоли введенную пользователем строку, при этом символы строки не отображаются на консоли

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

Для хранения групп объектов одинакового типа в языке программирования Java был разработан Collections Framework. К нему были предъявлены следующие требования:

* Должен позволять работать с разными типами данных одинаково и с высокой степенью совместимости.
* Должен иметь высокую производительность и эффективно реализовывать фундаментальные структуры данных такие, как связный список, деревья, хэш-таблицы и динамические массивы.
* Должен позволять легко создавать собственные коллекции для специфических задач.

Для реализации этих целей, был разработан набор интерфейсов, на базе которых были созданы готовые коллекции, позволяющие разрабатывать коллекции собственные.



* List Список объекты хранятся в порядке их добавления в список. Доступ к элементам списка осуществляется по индексу.
  + Элементы могут быть вставлены или доступны по их позиции в списке, используя индекс на основе нуля.
  + Список может содержать повторяющиеся элементы.
* Queue- определяет поведение класса в качестве однонаправленной очереди FIFO - первый пришел, первый ушёл. Предназначенн для вставки элементов в конец очереди и удаления элементов из начала очереди. Это похоже на работу очереди в супермаркете.
* Set – множество неповторяющихся объектов. В коллекции этого типа разрешено наличие только одной ссылки типа null. Вы можете добавить любой объект в набор. Если набор не типизирован с использованием Java Generics, то вы можете даже смешивать объекты разных типов (классов) в одном наборе.

1. Обход элементов коллекции. Интерфейс Iterator.

Мы можем использовать итератор Java для извлечения элементов и перебора любых Типы коллекции. Отсюда и название Универсальный итератор. Он поддерживает операции чтения и удаления.

Одним из ключевых методов интерфейса Collection является метод Iterator<E> iterator(). Он возвращает итератор - то есть объект, реализующий интерфейс Iterator.

* Next
* hasNext
* Remove

Изображение выглядит как текст

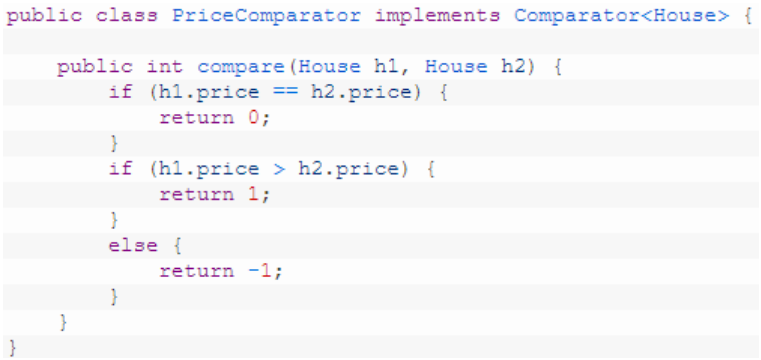
Автоматически созданное описание

1. Сортировка элементов коллекций. Интерфейсы Comparable и Comparator.

Comparable - автоматическая сортировки при условии реализации коллекции (сортировка по умолчанию) Имплеyментируя этот интерфейс мы как бы говорим "Эй, теперь объекты этого класса можно сравнивать между собой! И я знаю, как это сделать!" А до этого было нельзя

* – ноль, если два объекта равны;
* – число >0, если первый объект (на котором вызывается метод) больше, чем второй (который передается в качестве параметра);
* число <0, если первый объект меньше второго.

Collections.sort(List)

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

Collections.sort(List, comparator)

1. Интерфейс Set, его варианты и реализации.

Интерфейс Java Set, java.util.Set, представляет коллекцию объектов, где каждый объект уникален. Другими словами, один и тот же объект не может встречаться более одного раза в наборе. Интерфейс является стандартным и подтипом интерфейса Collection, что означает, что Set наследуется от Collection.

Вы можете добавить любой объект в набор. Если набор не типизирован с использованием Java Generics, то вы можете даже смешивать объекты разных типов (классов) в одном наборе. Однако в действительности смешивание объектов разных типов в одном наборе не часто выполняется. Будучи подтипом Collection, все методы в интерфейсе Collection также доступны в интерфейсе Set.

Поскольку Set – это интерфейс, вам нужно создать конкретную реализацию, чтобы использовать его. Вы можете выбрать:

* EnumSet это реализация наборов Set для работы с типом enum.
* HashSet HashSet поддерживается HashMap. Он не дает никаких гарантий относительно последовательности элементов при их итерации.
* LinkedHashSet отличается от HashSet тем, что гарантирует, что порядок элементов во время итерации совпадает с порядком их вставки в LinkedHashSet.
* TreeSet также гарантирует порядок элементов при повторении, но он является порядком сортировки элементов. Другими словами, порядок, в котором элементы должны быть отсортированы, если вы использовали Collections.sort() для List или массива, содержащего эти элементы.

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

Интерфейс Java List, java.util.List, представляет упорядоченную последовательность объектов. Элементы, содержащиеся в списке Java, могут быть вставлены, повторены и удалены в соответствии с порядком их появления в нем. Порядок элементов – вот почему эта структура данных называется списком. Каждый элемент имеет индекс: первый элемент – 0, второй – 1 и т. д. Индекс означает «сколько элементов находится в начале». Таким образом, первый элемент находится на расстоянии 0 элементов от начала списка, потому что он находится в начале.Будучи подтипом Collection, все методы в интерфейсе Collection также доступны в интерфейсе List.

Поскольку List – это интерфейс, вам нужно создать конкретную реализацию интерфейса, чтобы использовать его, либо выбрать между следующими в API коллекций:

* ArrayList лежит идея динамического массива. А именно, возможность добавлять и удалять элементы, при этом будет увеличиваться или уменьшаться по мере необходимости. ArrayList хранит Только ссылочные типы, любые объекты, включая сторонние классы. Строки, потоки вывода, другие коллекции. Для хранения примитивных типов данных используются классы-обертки.
* LinkedList элементы фактически представляют собой звенья одной цепи АССОЦИАТИВНЫЙ МАССИВ. У каждого элемента помимо тех данных, которые он хранит, имеется ссылка на предыдущий и следующий элемент. По этим ссылкам можно переходить от одного элемента к другому.
* Vector реализует динамический массив. Он похож на ArrayList, , но с двумя отличиями: Vector синхронизирован. AND 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.

* Stack. LIFO — "last in — first out"

1. Интерфейс Map, его варианты и реализации.

Map — это структура данных, которая содержит набор пар “ключ-значение”. По своей структуре данных напоминает словарь, поэтому ее часто так и называют. В то же время, Map является интерфейсом, и в стандартном jdk содержит основные реализации: Hashmap, LinkedHashMap, Hashtable, TreeMap.

Класс HashMap в Java использует хэш-таблицу для реализации интерфейса Map. Это позволяет времени для выполнения основных операций, таких как get () и put (), оставаться постоянным даже для больших множеств. (Не синхронизирован, null в качестве ключа и значения можно).

LinkedHashMap — это упорядоченная реализация хэш-таблицы, в которой имеются двунаправленные связи между элементами подобно LinkedList. Это преимущество имеет и недостаток — увеличение памяти, которое занимет коллекция.

TreeMap, как и LinkedHashMap, является упорядоченным набором данных. По-умолчанию, TreeMap сортируется по ключам с использованием принципа "natural ordering". Но это поведение может быть настроено под конкретную задачу при помощи объекта Comparator, который указывается в качестве параметра при создании объекта TreeMap.

Класс Hashtable позволяет реализовывать структуру данных типа hash-таблица, содержащую пары вида "ключ - значение". Значение "null" в качестве значения или ключа использовать нельзя. Hashtable является синхронизированной, т.е. почти все методы помечены как synchronized, в связи с чем могут быть проблемы с производительностью.

1. 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.

* Метод Collections.sort() Используется для сортировки элементов, присутствующих в указанном списке коллекции, в порядке возрастания.
* Метод Collections.binarySearch() который возвращает позицию объекта в отсортированном списке
* Методы Collections.reverse(), Collections.shuffle() reverse меняет порядок элементов в списке, передаваемом в качестве аргумента. Shuffle Он перемешивает заданный список, используя предоставленный пользователем источник случайности
* Методы Collections.max(), Collections.min()
* Метод Collections.copy()
* Метод Collections.rotate() Используется для поворота элементов, присутствующих в указанном списке Коллекции, на заданное расстояние

Arrays помогает решать типовые задачи при работе с массивами. В этот класс были добавлены методы для решения самых распространенных задач, с которыми сталкиваются Java-программисты в работе.

* Arrays.sort() выполняет сортировку массива
* Arrays.toString() для преобразования массива в строку
* Arrays.copyOf() мы передаем наш оригинальный массив и длину нового массива, в который мы копируем данные.
* equals(), который делает ровно то, что нам нужно

1. Обобщенные и параметризованные типы. Создание параметризованных классов.

Все классы в Java неявно (скрытно) унаследованы от класса Object.

Обобщения могут быть применены к классам, интерфейсам или методам. Если класс, интерфейс или метод оперирует некоторым обобщенным типом T, то этот класс (интерфейс, метод) называется обобщенным. Тип, который получает обобщенный класс в качестве параметра, называется параметризованным типом. Имя параметризованного типа можно задавать любым

Использование обобщений в языке Java дает следующие преимущества:

* обеспечивается компактность программного кода;
* благодаря обобщениям все операции приведения типов выполняются автоматически
* обобщения обеспечивают типовую безопасность типов в отличие от использования ссылок на тип Object.

Чтобы метод некоторого класса оперировал обобщенным типом Type, нужно чтобы класс (интерфейс) был объявлен как обобщенный. В языке Java допускается объявление класса для одного или нескольких обобщенных типов. Общая форма класса, который использует обобщенные типы, следующая:

class ClassName<Type1, Type2, ..., TypeN> {

// ...

Изображение выглядит как текст

Автоматически созданное описание}

1. Работа с параметризованными методами. Ограничение типа сверху или снизу.

В методах параметризованного класса можно использовать параметр типа, а следовательно, они становятся параметризованными относительно параметра типа.

Но можно объявить параметризованный метод, в котором непосредственно используется один или несколько параметров типа. Более того, можно объявить параметризованный метод, входящий в не параметризованный класс

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

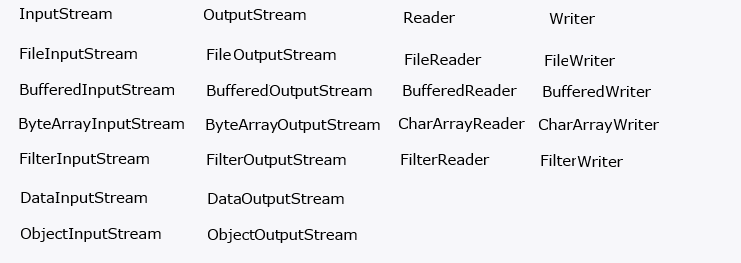
Автоматически созданное описание

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

Объект, из которого можно считать данные, называется потоком ввода, а объект, в который можно записывать данные, - потоком вывода.

В основе всех классов, управляющих потоками байтов, находятся два абстрактных класса: InputStream (представляющий потоки ввода) и OutputStream (представляющий потоки вывода) – Можно задать кодировку

Но поскольку работать с байтами не очень удобно, то для работы с потоками символов были добавлены абстрактные классы Reader (для чтения потоков символов) и Writer (для записи потоков символов).



1. Новый пакет ввода-вывода. Буферы и каналы.

Основное отличие между двумя подходами к организации ввода/вывода в том, что Java IO является потокоориентированным, а Java NIO – буфер-ориентированным.

Потокоориентированный ввод/вывод подразумевает чтение/запись из потока/в поток одного или нескольких байт в единицу времени поочередно. Данная информация нигде не кэшируются. Таким образом, невозможно произвольно двигаться по потоку данных вперед или назад. Если вы хотите произвести подобные манипуляции, вам придется сначала кэшировать данные в буфере. Подход, на котором основан Java NIO немного отличается. Данные считываются в буфер для последующей обработки. Вы можете двигаться по буферу вперед и назад. Это дает немного больше гибкости при обработке данных. В то же время, вам необходимо проверять содержит ли буфер необходимый для корректной обработки объем данных. Также необходимо следить, чтобы при чтении данных в буфер вы не уничтожили ещё не обработанные данные, находящиеся в буфере.

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.

Каналы напоминают трубопроводы, по которым эффективно транспортируются данные между буферами байтов и сущностями по ту сторону каналов. Каналы – это шлюзы, которые позволяют получить доступ к сервисам ввода/вывода операционной системы с минимальными накладными расходами, а буферы – внутренние конечные точки этих шлюзов, используемые для передачи и приема данных.

Таким образом неблокирующий режим Java NIO позволяет использовать один поток выполнения для решения нескольких задач вместо пустого прожигания времени на ожидание в заблокированном состояний. Наиболее частой практикой является использование сэкономленного времени работы потока выполнения на обслуживание операций ввода/вывода в другом или других каналах.

1. 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.

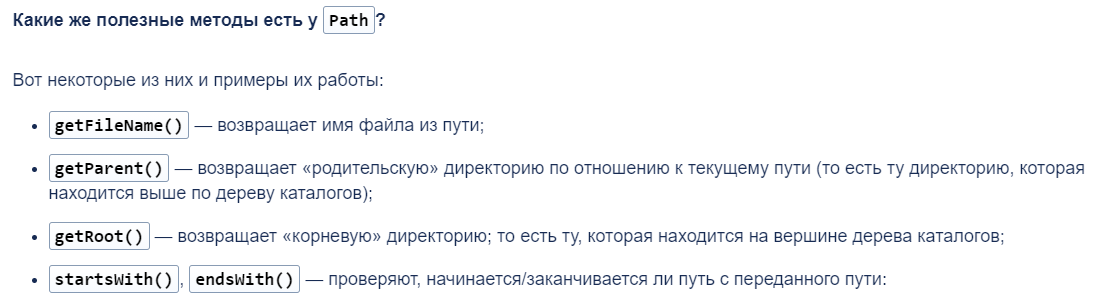
Если объект File представляет каталог, то его метод isDirectory() возвращает true. И поэтому мы можем получить его содержимое - вложенные подкаталоги и файлы с помощью методов list() и listFiles().

Экземпляры класса File являются неизменяемыми; то есть после создания абстрактный путь, представленный объектом File, никогда не изменится.

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. Поэтому в интерфейсе Path метод getParent() и другие методы работы с файлами возвращают просто объект Path. Никакой кучи вариантов — все легко и просто.



1. Сериализация объектов. Интерфейс Serializable.

Сериализация — это процесс сохранения состояния объекта в последовательность байт.

В Java за процессы сериализации отвечает интерфейс Serializable. Этот интерфейс крайне прост: чтобы им пользоваться, не нужно реализовывать ни одного метода! ЭТО МАРКЕР

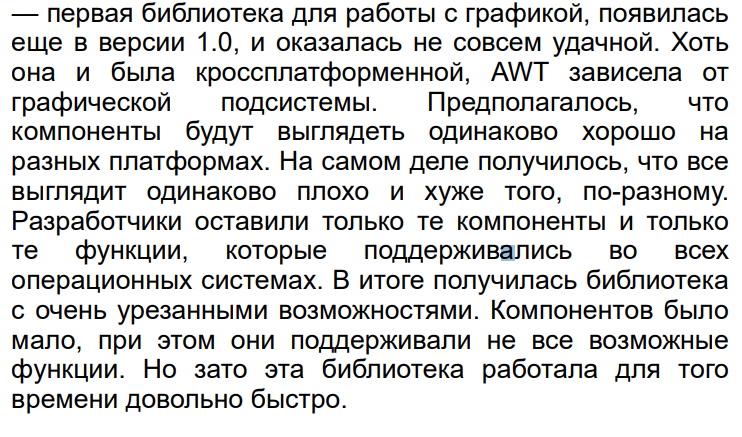
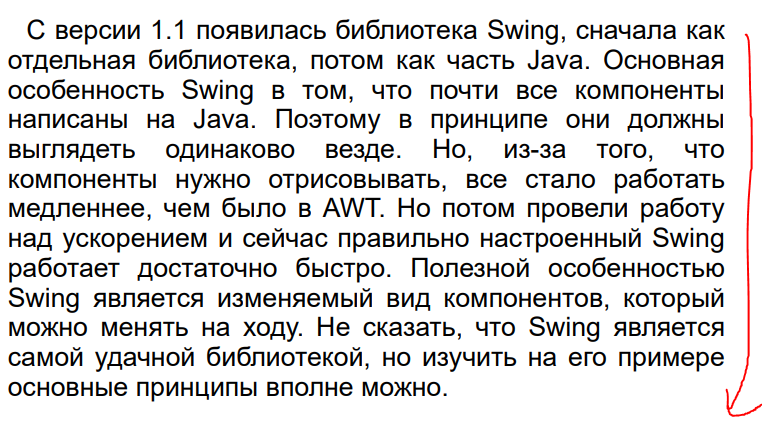
Еще один важный момент: переменная private static final long serialVersionUID, которую мы определили в классе. Зачем она нужна? Это поле содержит уникальный идентификатор версии сериализованного класса.

Выполняет запись в поток метаданных о классе, ассоциированном с объектом (имя класса, идентификатор SerialVersionUID, идентификаторы полей класса), рекурсивную запись в поток описания суперклассов до класса java.lang.Object (не включительно), запись примитивных значений полей сериализуемого экземпляра, начиная с полей самого верхнего суперкласса, рекурсивную запись объектов, которые являются полями сериализуемого объекта. При этом ранее сериализованные объекты повторно не сериализуются, что позволяет алгоритму корректно работать с циклическими ссылками.

Метод writeObject отвечает за запись состояния объекта для его конкретного класса, чтобы соответствующий метод readObject мог его восстановить.

Метод readObject отвечает за чтение из потока и восстановление полей класса

1. Библиотеки графического интерфейса. Особенности и различия.

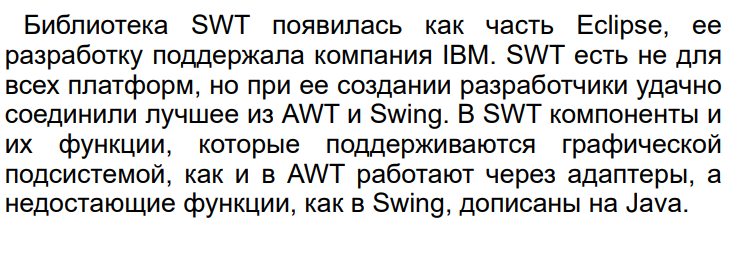
AWT — Abstract Window Toolkit

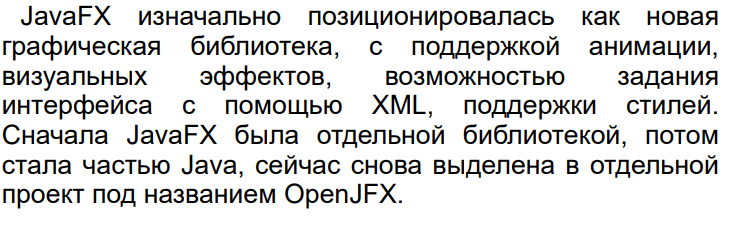
Swing - надстройка над AWT в виде легковесных Java-компонентов, изменяемый вид компонентов

Функциональность такая же как в AWT, только добавлены новые классы, Менеджеры компоновки, Обработка событий

Функциональность изменилась, Компоненты - легковесные, соответствующие компоненты ОС не используются.

SWT - Часть Eclipse, компоненты-оболочки для компонентов ОС , Часть Eclipse, компоненты-оболочки для компонентов ОС





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

JavaFX — новая библиотека для разработки RIA (Rich

Internet Applications) Поддержка XML для создания интерфейса Поддержка стилей CSS Поддержка 2D- и 3D-графики Легковесные компоненты Интеграция с библиотекой Swing

1. Библиотека Swing. Особенности.

Swing в Java является частью базового класса Java, который является независимым от платформы. Он используется для создания оконных приложений и включает в себя такие компоненты, как кнопка, полоса прокрутки, текстовое поле и т. д.

Swing в Java — это легкий инструментарий с графическим интерфейсом, который имеет широкий спектр виджетов для создания оптимизированных оконных приложений.

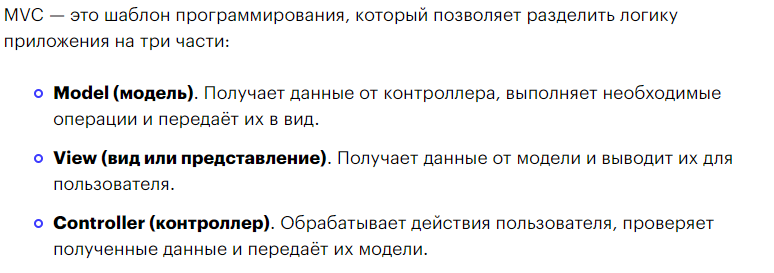
Он построен на основе AWT API и полностью написан на Java. Он не зависит от платформы в отличие от AWT и имеет легкие компоненты. Создавать приложения становится проще, поскольку у нас уже есть компоненты GUI, такие как кнопка, флажок и т. д.

Все компоненты в свинге, такие как JButton, JComboBox, JList, JLabel, унаследованы от класса JComponent, который можно добавить в классы контейнера.

Любой класс, в котором есть другие компоненты, называется контейнерным классом. Для создания приложений с графическим интерфейсом необходим как минимум один класс контейнеров. Ниже приведены три типа контейнерных классов:

* Панель — используется для организации компонентов в окне.
* Рамка — полностью функционирующее окно со значками и заголовками.
* Диалог — это как всплывающее окно, но не полностью функциональное, как рамка.

Контейнеры — это окна, такие как рамка и диалоговые окна. Основные компоненты являются строительными блоками любого графического приложения. Такие методы, как setLayout, переопределяют макет по умолчанию в каждом контейнере. Контейнеры, такие как JFrame и JDialog, могут добавлять только компонент к себе. Ниже приведены несколько компонентов с примерами, чтобы понять, как мы можем их использовать.



1. Библиотека SWT. Особенности.

SWT библиотека представляет собой кросс-платформенную оболочку для графических библиотек конкретных операционных систем.

Библиотека SWT написана на Java. Он использует собственные библиотеки операционной системы через собственный интерфейс Java (JNI ) для визуализации элементов графического интерфейса, аналогично программам, написанным с использованием собственного API операционной системы.

Программы, использующие SWT, переносимы, но реализация «набора инструментов», хотя и написана на Java, уникальна для каждой платформы.

SWT и Swing — это разные инструменты, созданные для разных целей. Цель SWT — предоставить API для доступа к нативной графике на широком спектре платформ. Основными целями дизайна являются: высокая скорость, естественный вид и глубокая интеграция с платформой. Swing — противоположность. разработан, чтобы обеспечить возможность гибкой настройки внешнего вида, одинакового на всех платформах «.

SWT относительно проще, чем Swing, без «ненужных» (для среднего разработчика) дополнительных функций. Однако это приводит к аргументу, что SWT не обладает некоторыми функциональными возможностями по сравнению со Swing.

SWT был разработан, чтобы быть очень мощным; быстрее, легче и лучше использует системные ресурсы, чем Swing.

Однако другие «реальные» тесты не выявили явного победителя, а скорее показали, что результаты сильно зависят от контекста и среды приложения.

1. Библиотека JavaFX. Особенности.

JavaFX - новая графическая библиотека, улучшенная поддержка анимации, визуальные эффекты,

JavaFX — новая библиотека для разработки RIA (Rich Internet Applications)

JavaFX нацелен на создание игр и настольных приложений на Java. По сути им заменят Swing из-за предложенного нового инструмента GUI для Java. Также, он позволяет нам стилизовать файлы компоновки GUI (XML) и сделать их элегантнее с помощью CSS, подобно тому, как мы привыкли к сетевым приложениям.

JavaFX дополнительно работает с интегрированной 3D-графикой, а также аудио, видео и встроенными сетевыми приложениями в единый инструментарий GUI… Он прост в освоении и хорошо оптимизирован. Он поддерживает множество операционных систем, а также Windows, UNIX системы и Mac OS.

Легковесные компоненты

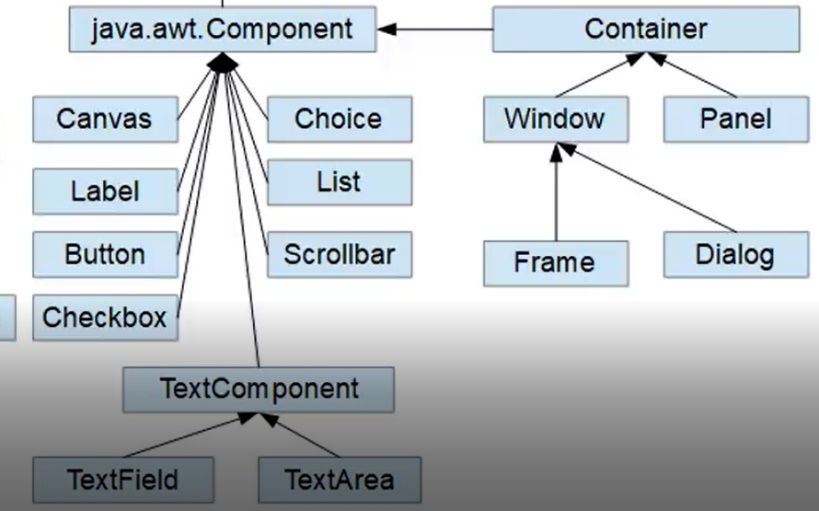
Особенности JavaFX:

* JavaFX изначально поставляется с большим набором частей графического интерфейса, таких как всякие там кнопки, текстовые поля, таблицы, деревья, меню, диаграммы и т.д., что в свою очередь сэкономит нам вагон времени.
* JavaFX часто юзает стили CSS, и мы сможем использовать специальный формат FXML для создания GUI, а не делать это в коде Java. Это облегчает быстрое размещение графического интерфейса пользователя или изменение внешнего вида или композиции без необходимости долго играться в коде Java.
* JavaFX имеет готовые к использованию части диаграммы, поэтому нам не нужно писать их с нуля в любое время, когда вам нужна базовая диаграмма.
* JavaFX дополнительно поставляется с поддержкой 3D графики, которая часто полезна, если мы разрабатываем какую-то игру или подобные приложения.

Интеграция с библиотекой Swing

1. Компоненты графического интерфейса. Класс Component.

* Component – абстрактный класс, элемент GUI, который взаимодействует с пользователем. Определять цвет размер и местоположение компонента. Порождает основные события.
  + Цвет – Color(RGB), константа
  + Размер и местоположение – Point(x, y) Demention(high, width) Rectangle(x, y, high, width)
  + Шрифт – Font
  + Видимость – isVisible(), setVisible() Frame Window не видимы после создания
  + Активность – isEnabled, setEnabled()



1. Размещение компонентов в контейнерах. Менеджеры компоновки.

Класс Container - это потомок класса Component, который может содержать другие компоненты. Простейший контейнер - это панель, на которой расположены компоненты. Каждый компонент может находиться только на одном контейнере. У класса Container есть методы add(), setLayout(), validate() и многие другие, но эти три являются основными, которые используются наиболее часто.

Разместить компоненты в контейнере можно двумя способами - плохим и хорошим. Плохой способ – это абсолютное позиционирование, когда, например, вы указываете, что такая-то кнопка должна располагаться на расстоянии 20 пикселей от левой границы экрана и 10 пикселей от верхней границы, размер она должна иметь такой-то и т.д. С одной стороны, вроде бы все удобно и несложно, но при этом отсутствует реакция на изменения размера контейнера. При изменении размера окна компонент все так же будет находиться в том же месте и иметь тот же размер. Чтобы этих проблем избежать, есть классы, которые называются менеджерами компоновки. Они управляют размером компонентов и их взаимным расположением.

Управляют местоположением компонентов и их размером

* Layout
* Layout2 -можно задать дополнительную характеристику.

Мы можем каждый компонент добавлять в менеджер компановки с помощью метода add. И менять с помощью set

Основные менеджеры компановки

* FlowLayout – заполняет контейнер слева направо построчно. Если компонент не влезает в строку – начинается следующая, они сохраняют свой предпочитаемый размер.
* GridLayout – таблица с ячейками одинакового размера. Если не хватит элементов – пустые ячейки можно задать число строк столбцов, границы. Элементы растягиваются под размеры ячейки.
* CardLayout – понель с имитацией вкладок. Мы добавляем вкладки и в каждый момент времени отображается только одна из них. Выбрать - show
* BorderLayout – Используется для окон верхнего уровня. Делит область на 5 частей. Добавление add(компонент , место)
* GridBagLayout – Контейнер делится на ячейки по строкам и столбцам. Вы сами все реализуете.

1. Обработка событий графического интерфейса.

Обработка событий обычно реализуется с помощью паттерна Observer, и такой вариант программирования называется «событийно-ориентированным», смысл его состоит в том, что последовательность исполнения кода не задается вообще, время, когда часть кода будет исполнена определяется наступланием событий. Пользователь нажимает на кнопки, выбирает элементы из списка, двигает мышкой и все эти действия вызывают асинхронное выполнение определенных участков кода.

Представим, что у нас есть кнопка и метка. Вызываем у кнопки метод addActionListener, которому передадим метку в качестве параметра. В переводе на человеческий язык это значит, что метка подписывается на извещения о событиях кнопки. Когда кнопку нажмут, она должна сказать об этом метке. Для того, чтобы это сработало, метка должна реализовывать интерфейс ActionListener, при этом у нее должен быть реализован метод actionPerformed(). Когда кнопку нажимают, кнопка радостно кричит «Меня нажали!» Этот крик передается каждому слушателю из списка с помощью вызова как раз именно этого метода actionPerformed(). И соответственно метка в этом методе реагирует на событие нажатия кнопки и перекрашивается в другой цвет. Источником событий может быть любой компонент. На него могут нажать мышкой, выбрать пункт меню или сделать что-то еще. Любое событие графического интерфейса — потомок класса AWTEvent. В классе события фиксируется время, когда событие произошло, компонент, где событие произошло, и, соответствено, что именно произошло. Обработчик события реализует интерфейс Что-то-там Listener. Слушателей много и они разные. Для того, чтобы подписаться на события, нужно вызвать метод «добавить слушателя» у компонента, где ожидается событие, чтобы отреагировать на событие — компонент вызывает метод слушателя, соответствующих событий, и зависящий от интерфейса слушателя.

Обработку событий можно реализовать анонимным классом, можно лямбда-выражением. Далее перейдем к рассмотрению конкретных слушателей

MouseListener, MouseMotionListener, (MouseAdapter можно использовать)

KeyListener, KeyEvent

WindowListener - слушатель, связанный с окном. Реагирует на события - окно открыто, окно закрывается, окно закрыто, окно активировано, окно деактивировано, окно свернуто, окно развернуто. WindowEvent - соответствующий класс-событие.

ActionListener, AdjustmentListener, ItemListener, ItemEvent, TextListener, TextEvent

1. Многопоточные программы. Класс Thread и интерфейс Runnable.

Класс Thread: для того, чтобы создать поток, необходимо создать объект этого класса. При создании собственного потока – необходимо переопределить метод run где вы пропишете то, что хотите от данного потока. Запустить поток после создания – метод start у объекта.

* Thread.currentThread() – получаем текущий поток выполнения.
* getName() – возвращает имя потока
* setName(String name) – устанавливает имя потока
* getPriority() – возвращает приоритет потока
* setPriority(int proirity) – устанавливает приоритет потока. Приоритет является одним из ключевых факторов для выбора системой потока из кучи потоков для выполнения. В этот метод в качестве параметра передается числовое значение приоритета - от 1 до 10. По умолчанию главному потоку выставляется средний приоритет - 5.
* isAlive(): возвращает true, если поток активен
* isInterrupted(): возвращает true, если поток был прерван
* join(): ожидает завершение потока
* run(): определяет точку входа в поток
* sleep(): приостанавливает поток на заданное количество миллисекунд
* start(): запускает поток, вызывая его метод run()

Интерфейс Runnable: этот интерфейс имеет один метод run().

1. Состояние потока. Синхронизация потока. Модификатор synchronized.

Можно приостановить выполнение потока на определенное время с помощью метода sleep(), которому передается время сна в мс. И еще можно дождаться завершения выполнения некоторого потока, вызвав у него метод join(). Также в классе Thread есть много других полезных методов, о которых можно прочитать в документации. С их помощью можно получить ссылку на текущий исполняющийся поток, получить идентификатор или имя потока, узнать, живой ли поток, получить приоритет потока и узнать группу, к которой поток принадлежит

При создании синхронизированного блока кода после оператора synchronized идет объект-заглушка: synchronized(res). Причем в качестве объекта может использоваться только объект какого-нибудь класса, но не примитивного типа.Синхронизация потоков. для работы с общими ресурсами.

Оператор synchronised()

Синхронизированный блок:         synchronized(res){...}

Синхронизированный метод:     synchronized void increment(){...}

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

Иногда нужно обеспечить взаимодействие потоков. Например, в случае, когда один поток генерирует значения, а другие потоки забирают их. Для простоты рассмотрим случай, когда значения передаются по одному. Один поток в методе put() при сброшенном флаге ready записывает значение в переменную value и устанавливает флаг ready, что значение готово. Другой поток в методе get() ждет, пока флаг не станет равным true, читает значение, и сбрасывает флаг, показывая что можно выдавать новое. Запись значение в переменную value должно находиться с сихнхронизированном блоке, чтобы исключить гонки, а флаг ready должен иметь модификатор volatile, чтобы компилятор не оптимизировал циклы с его проверкой. Недостаток данного способа реализации в том, что потоки нагружают процессор, пока ждут нужного значения флага. Такой вариант работы называется busy-wait.

Чтобы зря не грузить процессор, можно использовать методы wait и notify класса Object. Эти методы разрешено вызывать только при наличии захваченной блокировки, то есть внутри синхронизированного метода или блока. При вызове метода wait() поток помещается в очередь ожидания (wait set) объекта, у которого вызван метод. Дальше поток освобождает блокировку и начинает ждать. Ждет он, пока у объекта не будет вызван метод notify, либо поток не получит прерывание, либо (если метод wait был вызван с параметром времени) не истечет указанное время. Дальше поток вместе с другими кандидатами пытается захватить блокировку. Если ему это удается, он продолжает работу после метода wait. Если нет — поток возвращается в состояние ожидания. Метод notify выводит из ожидания один случайный поток, метод notifyAll — все потоки, которые ждут, но только один из них войдет в синхронизированный блок.

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

Но не всегда можно обойтить только неизменяемыми объектами. Есть много алгоритмов решения проблем многопоточности, которые реализованы в классах, входящих в пакет java.util.concurrent и его подпакеты. Это интерфейсы и классы, реализующие исполнителей (Executors), которые позволяют запускать задачи на параллельное исполнение, пулы потоков, которые дают возможность использовать потоки повторно, синхронизаторы, которые предоставляют готовые способы синхронизировать работу потоков, интерфейсы и классы для очередей задач и потокобезопасных коллекций. Также рассмотрим интерфейсы Lock и Condition — более удобные способы обеспечить механизмы блокировки доступа, а также ожидания-нотификации. И в пакет java.util.concurrent.atomic входят классы для работы с атомарными переменными, реализую атомарные операции для работы с ними.

Синхронизация действий потоков с помощью блокировок и условий реализована в Java на низком уровне — каждый объект имеет встроенную блокировку, которая захыватывается потоком при входе в синхронизированный блок, а также методы wait и notify, позволяющие входить в состояние ожидания и посылать извещения об изменении некоторого условия. Однако, часто требуется более гибкие методы решения той же задачи с расширенными возможностями. Для этого в стандартную библиотеку были добавлены интерфейсы Lock и Condition. Lock — это блокировка, аналогичная встроенной блокировке при использовании инхронизировнных методов и блоков, методы lock() и unlock() аналогичны захвату и освобождению блокировки, методы tryLock и lockInterruptibly позволяют попытаться захватить блокировку с возможностью прервать операцию.

1. Атомарные типы данных.

Пакет java.util.concurrent.atomic содержит девять классов для выполнения атомарных операций. Операция называется атомарной, если её можно безопасно выполнять при параллельных вычислениях в нескольких потоках, не используя при этом ни блокировок, ни синхронизацию synchronized. Оптимистичная блокировка.

Пакет java.util.concurrent.atomic содержит девять классов для выполнения атомарных операций. Операция называется атомарной, если её можно безопасно выполнять при параллельных вычислениях в нескольких потоках, не используя при этом ни блокировок, ни синхронизацию synchronized.

Согласно множеству источников, неблокирующие алгоритмы в большинстве случаев более масштабируемы и намного производительнее, чем блокировки. Это связано с тем, что операции CAS реализованы на уровне машинных инструкций, а блокировки тяжеловесны и используют приостановку и возобновление потоков, переключение контекста и т.д. Тем не менее, блокировки демонстрируют лучший результат только при очень «высокой конкуренции», что в реальной жизни встречается не так часто.

Атомарные типы данных – типы данные, которые могут иметь либо начальное значение до изменения, либо конечное. Никакого промежуточного значения быть не может.

1. Шаблоны проектирования. Структурные шаблоны.

К этим двум базовым принципам можно добавить третий - чтобы решить проблему возможных изменений, нужно отделить то, что изменяется, от того, что остается постоянным. Потом выделить эти изменения в отдельную сущность и инкапсулировать. Применив принципы, можно получить шаблон. Шаблоны проектирования традиционно делятся на порождающие, структурные и поведенческие.

Структурные шаблоны обычно решают задачу совместного использования классов, многие из них являются частными случаями делегирования.

**Adapter, Bridge, Composite, Decorator, Façade, Flyweight, Proxy.**

Шаблон Заместитель (Proxy) представляет собой случай, когда класс-делегат имеет тот же интерфейс, что и делегирующий класс. Методы те же, но вместо прямых вызовов используется прокси. (Объект создается долго или доступ переменный, доп дей-я можно)

Шаблон Адаптер (Adapter) применяется, когда есть уже готовый клиентский класс и готовая библиотека, предоставляющая нужную клиенту функциональность, но имеющая другой интерфейс. В этом случае, мы можем не переписывать код клиента и код библиотеки

Шаблон Декоратор (Decorator) также использует тот же самый интерфейс для делегата и для делегирующего класса, при этом добавляя функциональность. Декоратор позволяет динамически добавлять классам новые функции, не переписывая при этом код.

Шаблон Мост (Bridge) используется, чтобы раздельно можно было управлять иерархией абстракций и иерархией реализаций.

Шаблон Фасад (Facade) нужен, когда имеется сложная система объектов, но для упрощения пользования этой системой желательно предоставить клиенту простой интерфейс типа "включить-выключить", а все необходимые действия с системой выполнит фасад, вызвав методы объектов системы в нужной последовательности.

Шаблон Приспособленец (Flyweight) предназначен для случая, когда имеется огромное количество однотипных объектов, и для их хранения требуется большое количество памяти. В этом случае можно создавать один экземпляр для множества представляемых объектов, передавая уникальные характеристики как параметры, чтобы не тратить память на их хранение.

1. Шаблоны проектирования. Порождающие шаблоны.

К этим двум базовым принципам можно добавить третий - чтобы решить проблему возможных изменений, нужно отделить то, что изменяется, от того, что остается постоянным. Потом выделить эти изменения в отдельную сущность и инкапсулировать. Применив принципы, можно получить шаблон. Шаблоны проектирования традиционно делятся на порождающие, структурные и поведенческие.

Порождающие шаблоны регулируют создание новых объектов. Это обычно делается ограничением доступа к конструкторам и предоставлением методов для создания объектов.

**Factory Method, Abstract Factory, Builder, Prototype, Singleton, Object Pool.**

Если взаимодействовать на уровне абстракций, то в какой-то момент все равно понадобится создать конкретный объект. Шаблон Фабричный метод (Factory Method) позволяет отложить решение о конкретном типе создаваемого объекта на этап исполнения. Вместо прямого создания объекта конструктором у нас есть абстракция фабрики, метод которой возвращает абстракцию продукта. При этом метод конкретной фабрики возвращает уже конкретный продукт.

Шаблон Абстрактная фабрика (Abstract Factory) делает почти то же самое, но предназначен для генерации объектов не одного типа, а сразу целого семейства связанных типов. Но все равно вместо явного вызова конструктора мы получаем фабрику, которая умеет производить любой объект из заданного набора.

Если создаваемый объект зависит от большого числа параметров, причем некоторые из них могут иметь стандартные значения и тогда их не обязательно указывать, то есть два варианта создания объекта - либо создать объект со значениями по умолчанию, а потом с помощью сеттеров установить его поля, либо написать пару десятков конструкторов на все возможные варианты исходных значений. В первом случае проблема в том, что в процессе вызова сеттеров объект не доделан, во втором проблема в запутанных конструкторах. Шаблон Строитель (Builder) позволяет задать все параметры в объекте-строителе, а потот одним легким движением руки одномоментно создать нужный объект.

Другой способ создания сложных объектов - это клонирование. Можно заранее создать разные варианты, а потом в нужный момент создать копию и возвратить ее. Такой способ реализуется шаблоном Прототип (Prototype).

Шаблон Одиночка (Singleton) - это по факту тоже фабрика, у которой установлено ограничение на количество создаваемых объектов - он должен быть только один. После создания этого объекта, фабрика возвращает всегда один и тот же объект, не создавая новые.

1. Шаблоны проектирования. Поведенческие шаблоны.

К этим двум базовым принципам можно добавить третий - чтобы решить проблему возможных изменений, нужно отделить то, что изменяется, от того, что остается постоянным. Потом выделить эти изменения в отдельную сущность и инкапсулировать. Применив принципы, можно получить шаблон. Шаблоны проектирования традиционно делятся на порождающие, структурные и поведенческие.

Поведенческие шаблоны реализуют различные варианты поведения. Обычно они позволяют динамически управлять ответственностью.

**Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor.**

Шаблон Цепочка ответственности (Chain of Responsibility) позволяет клиенту не думать о том, какой из объектов выполнит его запрос. Запрос передается по цепочке исполнителей, и в итоге его выполнит тот исполнитель, который решил, что именно он способен выполнить необходимое действие.

Шаблон Команда (Command) позволяет разделить вызов и исполнение команды. Для вызываемого объекта все команды - экземпляры интерфейса Command, каждая отдельная команда знает, какой метод какого исполнителя нужно вызвать.

Интерпретатор (Interpreter) позволяет управлять поведением с помощью простого языка. Примерами могут быть регулярные выражения и форматирование строк методом printf()

Шаблон Итератор (Iterator) позволяет обеспечить перебор элементов коллекции. Обычно сама коллекция предоставляет итератор для обхода содержащихся в ней элементов.

Шаблон Посредник (Mediator) нужен, когда имеется сложная структура объектов, которым нужно как-то взаимодействовать между собой. Чтобы при добавлении нового класса, который должен взаимодействовать со всеми другими, не пришлось вносить изменения в код остальных классов

Чтобы иметь возможность сохранять состояние объектов, не нарушая инкапсуляцию, и не предоставляя доступ к закрытым элементам класса, можно использовать шаблон Хранитель (Memento). Класс может создать Memento, и восстановить свое состояние из него. А единый интерфейс Memento позволяет управлять сохраненными состояниями единым образом.

Шаблон Наблюдатель (Observer) позволяет реализовать так называемое событийное программирование. Объект-наблюдатель может подписаться на оповещения об изменениях состояния наблюдаемого объекта. Каждый раз, когда происходит обновление состояния, наблюдаемый объект вызывает метод оповещения у всех, кто за ним наблюдает.

Шаблон Состояние (State) позволяет реализовать конечный автомат, то есть автомат, имеющий конечное количество состояний, переход из которых в другие состояния задается таблицей переходов

Шаблон Стратегия (Strategy) позволяет подменять алгоритм действий. Каждый алгоритм реализуется отдельным классом-стратегией, который можно выбрать динамически.

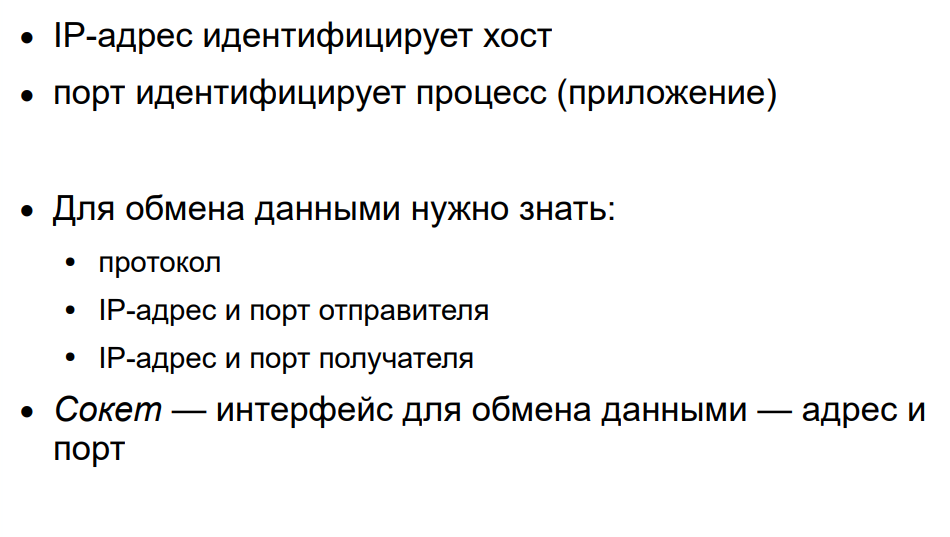
Шаблонный метод (Template Method) является основой фреймворков. В нем реализуется обычно закрытый метод, в котором прописан некий алгоритм или последовательность действий. Методы, которые вызываются внутри шаблонного метода – protected

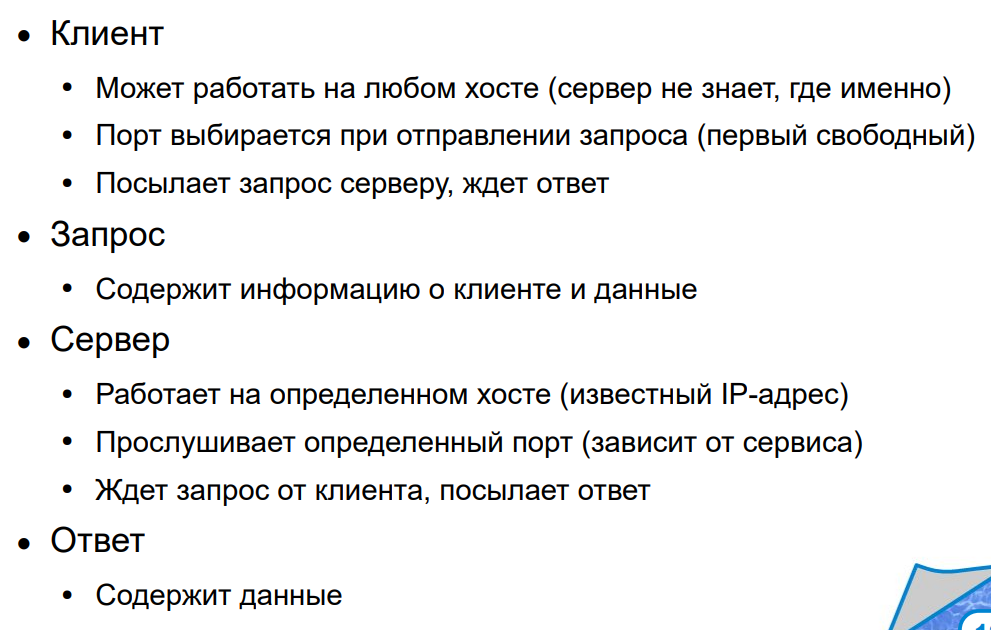
Шаблон Посетитель (Visitor) применяется в случаях, когда есть заданный стабильный набор объектов, при этом возникает необходимость выполнять над ними разный набор операций

1. Изображение выглядит как текст, газета

   Автоматически созданное описаниеСетевое взаимодействие. Основные протоколы, их сходства и отличия.



DNS — Domain Name Service



1. Протокол TCP. Классы Socket и ServerSocket.

Изображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описаниеИзображение выглядит как текст

Автоматически созданное описаниеИзображение выглядит как текст

Автоматически созданное описаниеИзображение выглядит как текст

Автоматически созданное описание

1. Протокол UDP. Классы DatagramSocket и DatagramPacket.

Изображение выглядит как текст

Автоматически созданное описаниеИзображение выглядит как текст

Автоматически созданное описание

Изображение выглядит как текст

Автоматически созданное описание

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

JDBC – набор интерфейсов и некоторых классов, которые позволяют работать с базами данных. Главный принцип архитектуры – универсальный способ общения с разными базами данных.

Основные классы и интерфейсы

* DriverManager: - Это класс, использующийся для управления списком Driver (database drivers).
* Driver - Это интерфейс, использующийся для соединения коммуникации с базой данных, управления коммуникации с базой данных. Когда загружается Driver, программисту не нужно конкретно вызывать его.
* Connection - Интерфейс со всеми методами связи с базой данных. Он описывает коммуникационный контекст. Вся связь с базой данных осуществляется только через объект соединения (connection).
* Statement -Это интерфейс, включающий команду SQL отправленный в базу данных для анализа, обобщения, планирования и выполнения.
* ResultSet - представляет набор записей, извлеченных из-за выполнения запроса.

Подключение происходит так: мы скачиваем нужный нам драйвер, вызвав class.forName() мы загружаем наш драйвер и он регистрируется у DriverManager, далее с помощью DriverManager.getConnection мы создаем соединение с нашей БД.

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

Типы драйверов JDBC:

1 тип — драйвер, работающий через ODBC-драйвер. Он появился самым первым, в то время для большинства СУБД уже существовали ODBC-драйверы, поэтому проще всего было реализовать драйвер, преобразующий вызовы JDBC в вызовы ODBC, а уже драйвер ODBC непосредственно работал с базой данных.

2 тип — драйвер, использующий для работы нативный интерфейс вызовов базы данных — DB API.

3 тип — драйвер, использующий промежуточный компонент — сервер приложений, взаимодействующий с JDBC по сети, и перенаправляющий запросы в СУБД.

4 тип (сейчас наиболее распространенный) — драйвер для СУБД, написанный на Java, напрямую реализующий спецификацию JDBC.

Для организации работы с базой данных используется класс DriverManager, который содержит список драйверов. В прошлых версиях JDBC драйверы надо было загружать явно, сейчас это делается автоматически. Загрузить драйвер явно можно, вызвав метод Class.forName() и передав ему имя класса драйвера. Метод getConnection() возвращает объект типа Connection.

Рассмотрим вкратце пример взаимодействия. Вызываем метод getConnection у класса DriverManager, которому передаем параметры соединения с базой данных. Этот метод вернет объект типа Connection, представляющий собой абстракцию соединения. У соединения вызывается метод createStatement, который создает запрос - объект типа Statement. Исполнение запроса производится с помощью метода executeQuery. Он возвращает результат - объект типа ResultSet.

Кроме классов, находящихся в пакете java.sql и реализующих базовые возможности взаимодействия с базой данных, имеется библиотека дополнительных классов и интерфейсов в пакете javax.sql. Начнем их рассмотрение с интерфейса DataSource. Этот интерфейс представляет источник данных и позволяет получить соединение с базой данных. В JDBC определены 3 типа источников данных: простой, с поддержкой пула соединений и с поддержкой распределенных транзакций. ConnectionPoolDataSource позволяет снизить затраты на создание соединений. Он работает аналогично пулу потоков, когда после использования соединение не закрывается, а возвращается в пул. XADataSource, позволяет работать с таблицами, находящимися в разных базах данных, при этом менеджер транзакций обеспечивает непротиворечивость данных.

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

интерфейс Statement и его потомков. Объект интерфейса Statement сам запрос не содержит. Запрос передается в качестве параметра при вызове метода executeQuery(). При вызове данного метода нужно быть внимательным при формировании запроса, если он составляется из данных, предоставленных пользователем. Никогда нельзя доверять полученным от пользователя данным, нужно контролировать их тип и содержимое, чтобы не допустить применение SQL-инъекций.

Объект интерфейса PreparedStatement представляет подготовленный запрос с параметрами. Он применяется для выполнения идентичных запросов, отличающиеся одним или несколькими параметрами, например, ID. В таких случаях запрос подготавливается заранее, а на месте параметров указываются знаки вопроса. Затем можно задать значения параметров с помощью одного из методов setInt, setString, или подобных (выбор метода зависит от типа параметра). Запрос выполняется при вызове метода executeQuery(). Этот способ имеет несколько преимуществ. Во-первых, запросы исполняются быстрее, так как при подготовке запроса СУБД формирует оптимальный план исполнения запроса, остается только подставить параметры и выполнить его. Во-вторых, такие запросы защищены от SQL-инъекций, так как все специальные символы экранируются автоматически.

Интерфейс CallableStatement расширяет PreparedStatement, и еще умеет вызывать хранимые процедуры, то есть процедуры, которые хранятся в базе данных. В языке SQL они вызываются с помощью команды CALL. Для этого в объекте типа CallableStatement, задаются входные параметры, и регистрируются выходные параметры. При регистрации выходного параметра его номеру ставится в соответствие тип параметра. После выполнения запроса можно получить значения выходных параметров.

1. Обработка результатов запроса. Интерфейсы ResultSet и RowSet.

Для объектов Statement запрос указывается в параметрах методов исполнения запроса. Для объектов PreparedStatement и CallableStatement параметр не нужен, так как запрос уже задан. Метод execeuteQuery выполняет запрос (обычно это запрос типа SELECT) и возвращает в качестве результата ResultSet, из которого потом можно будет получить данные.

Интерфейс ResultSet, объект которого возвращают методы executeQuery и getResultSet. Получить результат из ResultSet можно как из обычного итератора, в цикле while вызывается метод next для установки курсора на очередную строку. С помощью методов getString(), getInt() и т. д. можно получать значения столбцов для текущей строки.

При создании запроса можно указать, какие характеристики будет иметь выдаваемый запросом ResultSet. По умолчанию ResultSet позволяет перемещать курсор только вперед и работает только на чтение.

Часть методов интерфейса ResultSet предназначена для перемещения курсора на следующую, предыдущую, первую или последнюю строку, также курсор можно установить перед первой или после последней строки, переместить его абсолютно или относительно, либо установить на специальную строку для вставки.

Другие методы нужны для получения данных - это методы get с разными типами данных SQL. Все они принимают в качестве параметра либо порядковый номер элемента в строке, либо имя столбца. Второй способ предпочтительнее, так как при изменении запроса не придется менять код для получения результата.

С помощью метода getMetaData у объекта ResultSet можно получить метаданные результата

В расширенный пакет входит интерфейс RowSet. Он расширяет ResultSet. Кроме работы с результатом запроса RowSet может осуществить соединение с базой данных и выполнить сам запрос. В этом интерфейсе есть методы setURL, setUsername и setPassword для задания параметров соединения. Метод setCommand() позволяет задать запрос, метод execute выполняет запрос и получает результат. RowSet может поддерживать постоянное соединение с базой данных, а может соединяться с базой только в случае необходимости.

JdbcRowSet — это самый простой из интерфейсовнаследников RowSet. Он единственный из них поддерживает соединение активным. Его удобно использовать вместо обычного ResultSet

1. Интернационализация. Локализация. Хранение локализованных ресурсов.

**Локализация** — адаптация программы для определенного языка или местности.

* Локализация обычно включает, перевод текста на другой язык,
* использование определенных форматов чисел, отображения даты и времени,
* замена звуковой и визуальной информации.

**Интернационализация** – это не процесс, это стиль, как правильно писать программу, чтобы потом ее было удобно локализовывать. Интернационализация обычно заключается в том, что текстовые данные отделяются от кода и хранятся отдельно, а числа и даты отображаются не напрямую, а прогоняются через форматтеры, которые преобразуют данные в нужный формат.

Есть у нас некое приложение, есть дополнительный текстовый файл, в нем есть некоторый раздел, и дальше мы пишем, что при использовании английского языка нужно будет вывести строку «hello». Это уже интернационализованное приложение, но оно пока не локализовано. Чтобы оно стало локализованным, надо текстовый файл дополнить. Код уже трогать не надо. Остается только раздать текстовые файлы переводчикам, которые добавят поддержку языков. Это приложение уже интернационализованное и локализованное одновременно.

Локаль — это объект, задающий местность и язык. Элементы локали — это язык, страна и вариант, изредка встречаются дополнииельные элементы. Язык обозначается двумя строчными буквами: ru, en. Страна - обычно двумя заглавными буквами, иногда тремя цифрами. Вариант - показывает либо кодировку, либо применяемый календарь

Хранение данных

* PropertyResourceBundle - используется для хранения исключительно текствой информации (при этом его проще использовать)
  + При использовании PropertyResourceBundle достаточно создать текстовые файлы с опредленным именем. Имя должно состоять из названия ресурса, локали и расширения .properties. Несколько файлов с одним именем ресурса, но разными локалями образуют набор ресурсов (ResourceBundle). С помощью метода getBundle набор ресурсов загружается и после этого его можно использовать. Текстовые данные берутся из набора ресурсов по ключу, возможно с указанием локали. Основное преимущество файлов свойств - простой формат файлов с данными. Недостаток - возможность использовать только текстовые данные.
* ListResourceBundle - можно применять для хранения любых объектов.
  + Нужно создать наследников класса ListResourceBundle с именами этих классов, сформированными по тому же принципу как и у файлов свойств, Затем в каждом классе реализовать метод getContents(), возвращающий двумерный массив объектов, содержащий ключи и соответствующие им значения. Такой способ немного сложнее, потому что формат класса более сложный, после того, как класс создан, его необходимо скомпилировать. Но при этом можно хранить любые объекты, не только текстовые.

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

NumberFormat - абстрактный класс с фабричными методами, которые возвращают одного из потомков NumberFormat, умеющего форматировать числа в соответствии с заданными правилами (для числа, для валюты и для процентов). Обычно этим потомком является объект класса DecimalFormat. С помощью метода format можно получить число в нужном формате. Наиболее общий формат для всех локалей выдают фабричные методы NumberFormat.

Абстрактный класс DateFormat имеет фабричные методы для получения форматтера для даты, времени, и совмещенного. Можно задать тип формата: полный, длинный, средний, короткий. Если локаль не задана, используется системная. Данные методы возвращают объект класса SimpleDateFormat, являющийся потомком DateFormat. SimpleDateFormat позволяет указать произвольный шаблон формата. И класс DateFormatSymbols используется для замены стандартных элементов формата даты или времени.

MessageFormat, который позволяет использовать параметры в строке. Соответственно, можно задать для разных языков строки с параметрами в разных местах. На слайде показан пример для фраз на русском и английском языках, где в строку подставляется время, дата и другая строка.

Класс ChoiceFormat позволяет выбрать и подставитьт один вариант из нескольких в зависимости от используемого значения. Его можно использовать, например, для выбора нужной формы слова при числительных (хотя для русского языка это реализовать сложнее, чем для английского). Для создания формата нужно задать массив чисел, задающих диапазоны, и соответствующие этим диапазонам ключи.

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

Ну наконец-то, аж в восьмой яве, появилось отличное и правильное API для работы со [временем](http://blog.gelin.ru/2015/12/blog-post.html). Теперь можно смело выкинуть java.util.Date и java.text.DateFormat. Теперь у нас есть [java.time](http://docs.oracle.com/javase/8/docs/api/java/time/package-summary.html" \l "package.description). Чаще всего вам понадобится просто отметка времени, точка на временной оси. Это — java.time.Instant.

Можно получить момент времени сейчас. Можно получить момент времени из юниксовых секунд от начала эпохи. Можно получить момент времени из миллисекунд от начала эпохи, как это представляется в java.util.Date. А можно взять момент времени из строки в формате ISO 8601 с буковкой «Z» в конце, что означает, что это время в UTC, а значит, не подвержено особенностям разных таймзон.

Эта возможность гонять из чисел и строк туда и обратно очень полезна. Сохраняйте таким образом Instant куда угодно и загружайте обратно, это будет совершенно правильно и безопасно.

Допустим, у нас есть какая-то дата вида «12/07/2017» (тут главное, сразу выяснить, где месяц, а где день месяца). Её можно распарсить с помощью java.time.format.DateTimeFormatter и получить java.time.LocalDate.

Также со временем и временными зонами, удобно работать, кастовать и не путаться.

1. Рефлексия. Классы Class, Field, Method, Constructor.

Рефлексия (от позднелат. reflexio — обращение назад) — это механизм исследования данных о программе во время её выполнения. Рефлексия позволяет исследовать информацию о полях, методах и конструкторах классов. Сам же механизм рефлексии позволяет обрабатывать типы, отсутствующие при компиляции, но появившиеся во время выполнения программы.

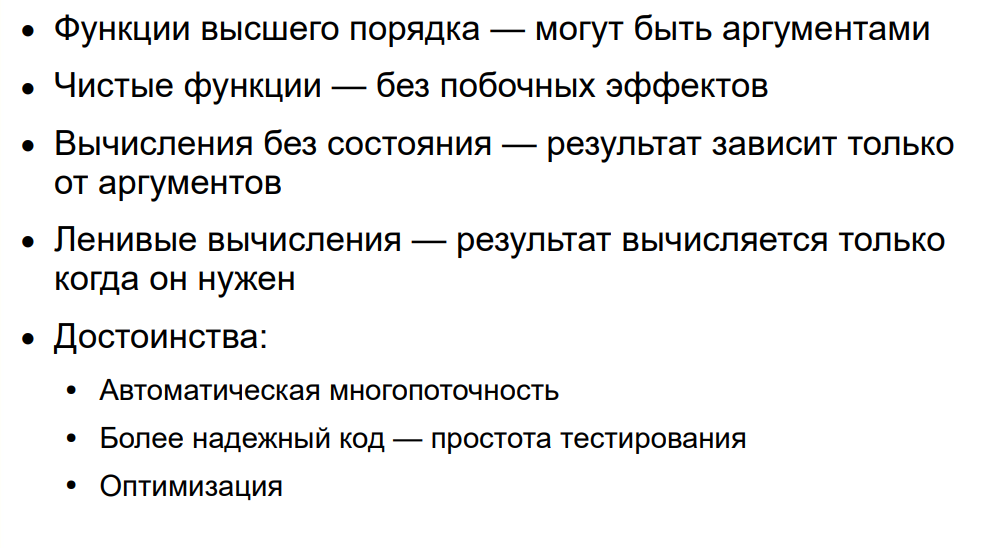
В java есть замечательный класс Class. Он представляет классы и интерфейсы в исполняемом приложении Java. Чтобы получить поля этого класса нужно вызвать метод getFields(), этот метод вернет нам все доступные поля класса. Нам это не подходит, так как наше поле private, поэтому используем метод getDeclaredFields(), этот метод также возвращает массив полей класса, но теперь и private и protected. В нашей ситуации мы знаем имя поля, которое нас интересует, и можем использовать метод getDeclaredField(String), где String — имя нужного поля.

Примечание: getFields() и getDeclaredFields() не возвращают поля класса-родителя!

Отлично, мы получили объект Field с ссылкой на наш name. Т.к. поле не было публичным (public) следует дать доступ для работы с ним. Метод setAccessible(true) разрешает нам дальнейшую работу.

Здесь примерно такая же процедура как и с получением поля — получаем нужный метод по имени и даем доступ к нему. И для вызова объекта Method используем invoke(Оbject, Args), где Оbject — все также экземпляр класса MyClass.

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

Если в двух словах, то функциональное программирование — это программирование, в котором функции являются объектами, и их можно присваивать переменным, передавать в качестве аргументов другим функциям, возвращать в качестве результата от функций и т. п.

Лямбда-выражение — это такая функция. Можете считать, что это обычный метод в Java, только его особенность в том, что его можно передавать в другие методы в качестве аргумента.

Те интерфейсы, у которых только один метод, также называют функциональными интерфейсами. В Java 8 они даже помечены специальной аннотацией @FunctionalInterface.

Формальную систему для выражения вычислений, основанных на абстракции функций . Эта универсальная модель вычислений стала известна как Лямбда-исчисление

Функции более высокого порядка способны принимать функцию в качестве аргументов и возвращать функцию в результате .

Определение чистой функции подчеркивает, что чистая функция должна возвращать значение, основанное только на аргументах, и не должна иметь побочных эффектов .

Неизменяемость является одним из основных принципов функционального программирования, и она относится к свойству, что сущность не может быть изменена после создания экземпляра .

Мы называем выражение ссылочно прозрачным, если замена его соответствующим значением не влияет на поведение программы .

Композиция функций относится к составлению сложных функций путем объединения более простых функций

монада позволяет нам обернуть значение, применить набор преобразований и получить значение обратно со всеми примененными преобразованиями .

Каррирование – это математический метод преобразования функции, которая принимает несколько аргументов, в последовательность функций, которые принимают один аргумент.

Рекурсия – это еще один мощный метод в функциональном программировании, который позволяет нам разбить проблему на более мелкие части .

В JDK 8 вместе с самой функциональностью лямбда-выражений также было добавлено некоторое количество встроенных функциональных интерфейсов, которые мы можем использовать в различных ситуациях и в различные API в рамках JDK 8. В частности, ряд далее рассматриваемых интерфейсов широко применяется в Stream API - новом прикладном интерфейсе для работы с данными. Рассмотрим основные из этих интерфейсов:

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

1. Конвейерная обработка данных. Пакет java.util.stream.

Stream API — это новый способ работать со структурами данных в функциональном стиле. Stream (поток) API (описание способов, которыми одна компьютерная программа может взаимодействовать с другой программой) — это по своей сути поток данных. Сам термин "поток" довольно размыт в программировании в целом и в Java в частности.

С появлением Java 8 Stream API позволило программистам писать существенно короче то, что раньше занимало много строк кода, а именно — упростить работу с наборами данных, в частности, упростить операции фильтрации, сортировки и другие манипуляции с данными. Если у вас промежуточных операций нет, часто можно и нужно обойтись без стрима, иначе код будет сложнее чем без потока.

