Оглавление

[1. Парадигма объектно-ориентированного программирование. Основные принципы ООП и их реализация в языке программирования Java и С++ 2](#_heading=h.gjdgxs)

[2. Организация программы на Java. Основные структурные единицы. Процесс интерпретации и компиляции. Роль JVM 2](#_heading=h.30j0zll)

[3. Структурирование Java приложения, пакеты. Уровни доступа и видимости 3](#_heading=h.1fob9te)

[4. Примитивные и ссылочные типы данных. Использование мехаизмов автоупаковки и автораспаковки. Операция приведения типов. Понижающее и повышающее приведение. 3](#_heading=h.3znysh7)

[5. Этапы проектирования, разработки и отладки ООП программ. Понятие конвенции кода языка и стиля программирования. 4](#_heading=h.2et92p0)

[6. Массивы в Java. Способы создания массивов. Индексы. Размерность массивов. Доступ к элементам массива и примеры использования 4](#_heading=h.tyjcwt)

[7. Класс Scanner и его использование для чтения стандартного потока ввода, конструктор класса Scanner 4](#_heading=h.3dy6vkm)

[8. Методы класса Scanner nextLine(), nextInt(), hasNextInt(), hasNextLine() и их использование для чтения ввода пользователя с клавиатуры 5](#_heading=h.1t3h5sf)

[9. Виды типов данных в Джава. Примитивные типы данных, объявление и присваивание переменных. Константы в Джава: объявление и использование константы 5](#_heading=h.4d34og8)

[10. Виды типов данных в Джава. Объектные типы данных 6](#_heading=h.2s8eyo1)

[11. Объявление и использование бестиповых переменных в Джава 6](#_heading=h.17dp8vu)

[12. Объявление переменных типа класс и их инициализация 7](#_heading=h.3rdcrjn)

[13. Массивы в Джава, как объектные типы данных, контроль доступа за выход за границы массива. Объявление и инициализация массивов, длина массива, получение доступа к элементу массива 7](#_heading=h.26in1rg)

[14. Способы объявления массивов в Джава, использование операции new для выделения памяти для элементов массива. Объявление с инициализацией, объявление массива определенного размера без инициализации. 7](#_heading=h.lnxbz9)

[15. Инициализация полей класса и локальных переменных (отличие), инициализатор и статический инициализатор (когда вызывается). 7](#_heading=h.35nkun2)

[16. Циклические конструкции в Java. Использование циклов для работы с массивами. Использование итераторов для обработки массивов. Использование итераторов для работы с коллекциями 7](#_heading=h.1ksv4uv)

[17. Статические поля и методы. Класс Math, его основные методы. 7](#_heading=h.44sinio)

[18. Понятие перечисления. Состав и приемы использования в ООП программах на Jаva 7](#_heading=h.2jxsxqh)

[19. Понятие класса. Определение, инициализация. Модификаторы доступа. Константы и переменные. Объявление классов. 8](#_heading=h.z337ya)

[20. Получение информации о типе. Создание экземпляров классов. Вызов методов класса Объявление класса на Джава, пример объявления 8](#_heading=h.3j2qqm3)

[21. ООП в Java. Понятие объекта. Что представляет собой Java приложение с точки зрения ООП. Основные характеристики объектов в Java. 8](#_heading=h.1y810tw)

[22. Конструкторы, назначение и использование. Конструктор с параметром, конструктор по умолчанию. 8](#_heading=h.4i7ojhp)

[23. Конструкторы, назначение и использование. Вызов конструктора родительского класса, неявный вызов конструктора родительского класса, порядок инициализации экземпляра Java класса. 8](#_heading=h.2xcytpi)

[24. Использование языка UML для проектирования и документирования объектно-ориентированных программ. Основные UML диаграммы для отображения отношений между классами в ООП программах 9](#_heading=h.1ci93xb)

[25. Управление памятью в Java и C++, процесс освобождения памяти, занимаемой объектом. Метод finalize. 9](#_heading=h.3whwml4)

[26. Понятие рекурсии, виды рекурсии и ее использование. Реализация Рекурсивных алгоритмов в ООП программах 9](#_heading=h.2bn6wsx)

[27. Оператор new. Понятие ссылки и указателя на объект. Реализация в С++ и Java. Время жизни объекта 9](#_heading=h.qsh70q)

[28. Переопределение методов в Java, абстрактные методы. 10](#_heading=h.3as4poj)

[29. Преобразование ссылочных типов в Java, instanceof (экземпляр класса). 10](#_heading=h.1pxezwc)

[30. Графическая подсистема. Основы AWT, Swing components. Событийная модель при программировании GUI в ООП программах 10](#_heading=h.49x2ik5)

[31. Использование языка UML для проектирования и документирования объектно-ориентированных программ. Основные UML диаграммы для отображения отношений между классами в ООП программах 10](#_heading=h.2p2csry)

[32. ООП в Java. Понятие объекта. Что представляет собой Java приложение с точки зрения ООП. Основные характеристики объектов в Java. 10](#_heading=h.147n2zr)

[33. Модификатор доступа или видимости в Джава, виды и использование. Использования this для доступа к компонентам класса. 10](#_heading=h.3o7alnk)

[34. Чем отличаются static-метод класса от обычного метода класса. Можно ли вызвать static-метод внутри обычного метода? 10](#_heading=h.23ckvvd)

[35. Объявление и использование методов, объявленных с модификатором public static. Как вызвать обычный метод класса внутри static-метода? 11](#_heading=h.ihv636)

[36. Синтаксис объявления методов в Джава, тип возвращаемого значения, формальные параметры и аргументы. Методы с пустым списком параметров 11](#_heading=h.32hioqz)

[37. Стандартные методы класса сеттеры и геттеры, синтаксис и их назначение? 11](#_heading=h.1hmsyys)

[38. Может ли быть поле данных класса объявлено как с модификатором static и final одновременно и что это означает? 11](#_heading=h.41mghml)

[39. Наследование, виды наследования и его реализация в Java и C++ 12](#_heading=h.2grqrue)

[40. Расширение классов. Порядок создания экземпляра дочернего класса. 12](#_heading=h.vx1227)

[41. Наследование в Джава. Вид наследования и синтаксис Ключевое слово extends 12](#_heading=h.3fwokq0)

[42. Что означает перегрузка метода в Java (overload) и переопределение метода в Java (override)? В чем разница? 12](#_heading=h.1v1yuxt)

[43. Абстрактные классы в Джава и абстрактные методы класса. Вложенные и анонимные классы. 13](#_heading=h.4f1mdlm)

[44. Виды наследования в Джава, использование интерфейсов для реализации наследования 13](#_heading=h.2u6wntf)

[45. Что наследуется при реализации наследования в Джава (какие компоненты класса), а что нет? 13](#_heading=h.19c6y18)

[46. К каким методам и полям базового класса производный класс имеет доступ (даже если базовый класс находится в другом пакете), а каким нет? Область видимости полей и данных из производного класса 13](#_heading=h.3tbugp1)

[47. Класс Object, его методы, их назначение. Иерархия классов в Java. 13](#_heading=h.28h4qwu)

[48. Наследование. Использование ключевых слов this и super. Пример использования в языках Си++ и Java 13](#_heading=h.nmf14n)

[49. Паттерны проектирования программ. Паттерн Фабрика. 13](#_heading=h.37m2jsg)

[50. Паттерны проектирования программ. Паттерн Фабричный метод. 13](#_heading=h.1mrcu09)

[51. Расширение классов в Джава. Переопределение методов. Сокрытие полей данных. 14](#_heading=h.46r0co2)

[52. Паттерны проектирования программ. Паттерн Observer и модель MVC 14](#_heading=h.2lwamvv)

[53. Интерфейсы. Общий синтаксис и расширение. Пустые интерфейсы. Реализация и применение. Сравнение с абстрактными классами. 14](#_heading=h.111kx3o)

[54. Обработка строк в Java. Класс StringBuffer. Класс StringBuilder 15](#_heading=h.3l18frh)

[55. Работа со строками в Java, строковый кэш. Операция конкатенации строк 15](#_heading=h.206ipza)

[56. Интерфейс Comparable и Comparator. Использование интерфейсных ссылок для написания обобщенных алгоритмов 15](#_heading=h.4k668n3)

[57. Понятие сортировки массивов. Сортировка пузырьком. Сортировка вставками. Использование полиморфизма (ООП) для программирования алгоритмов сортировок в массивах и коллекциях 15](#_heading=h.2zbgiuw)

[58. Понятие поиска в массивах. Последовательный поиск. Сортировка методом прямого выбора. Использование полиморфизма (ООП) для программирования алгоритмов поиска в массивах и коллекциях 16](#_heading=h.1egqt2p)

[59. Объявление и инициализация переменных типа String. Операция конкатенации строк и ее использование 16](#_heading=h.3ygebqi)

[60. При создании объектов строк с помощью класса StringBuffer, например StringBuffer strBuffer = new StringBuffer(str) можно ли использовать операцию конкатенации строк или необходимо использовать методы класса StringBuffer 16](#_heading=h.2dlolyb)

[61. Объявление и инициализация массива строк. Организация просмотра элементов массива 16](#_heading=h.sqyw64)

[62. Понятие и объявление интерфейсов в Джава. Может ли один класс реализовывать несколько интерфейсов? 16](#_heading=h.3cqmetx)

[63. Что входит в состав интерфейса (какие компоненты может содержать интерфейс)? Может ли интерфейс наследоваться от другого интерфейса? 16](#_heading=h.1rvwp1q)

[64. Интерфейсные ссылки и их использование в Джава 16](#_heading=h.4bvk7pj)

[65. Интерфейс Camparable, назначение, его методы и использование в Джава 16](#_heading=h.2r0uhxc)

[66. Какое значение возвращает вызов метода object1.compareTo(object2), который сравнивает 2 объекта obj1 и obj2 в зависимости от объектов? 16](#_heading=h.1664s55)

[67. Понятие исключительной ситуации, причины возникновения, механизм обработки. Классификация исключений. Исключения, классификация и использование исключений. Генерация (порождение исключений). 17](#_heading=h.3q5sasy)

[68. Служебное слово throw и его использование при определении методов. В каком случае программа должна использовать оператор throw? 17](#_heading=h.25b2l0r)

[69. Создание собственных классов исключений 17](#_heading=h.kgcv8k)

[70. Блок try/catch/finally, его предназначение и особенности 17](#_heading=h.34g0dwd)

[71. В Java все исключения делятся на два основных типа. Что это за типы и какие виды ошибок ни обрабатывают? 18](#_heading=h.1jlao46)

[72. Код ниже вызовет ошибку: Exception <...> java. 18](#_heading=h.43ky6rz)

[73. lang.ArrayIndexOutOfBoundsException: 4: Что она означает? 18](#_heading=h.2iq8gzs)

[74. Контролируемые исключения (checked) и неконтролируемые исключения (unchecked) и ошибки, которые они обрабатывают 18](#_heading=h.xvir7l)

[75. Как реализуется принципы ООП в Java при создании исключений? Порядок выполнения операторов при обработке блока блока try...catch 18](#_heading=h.3hv69ve)

[76. Абстрактный тип данных Stack (cтек) в Джава 18](#_heading=h.1x0gk37)

[77. Абстрактный тип данных Queue (очередь) в Джава 18](#_heading=h.4h042r0)

[78. Универсальные типы или обобщенные типы данных, для чего создаются? 19](#_heading=h.2w5ecyt)

[79. Объявление обобщённого класса коллекции с параметризованным методом для обработки массива элементов коллекции на основе цикла for each (определение общего метода для отображения элементов массива) 19](#_heading=h.1baon6m)

[80. Что представляет из себя класс ArrayList и в каком случае используется 19](#_heading=h.3vac5uf)

[81. Обобщенное программирование. Понятие и использование дженериков в Java. 19](#_heading=h.2afmg28)

[82. Параметризованные классы и методы. Их определение и использование 19](#_heading=h.pkwqa1)

[83. Стирание типов. 19](#_heading=h.39kk8xu)

[84. Понятие структуры данных список. Линейный список. Виды списков и их реализация на Java. Доступ к элементу структуры данных список. Использование списков. Трудоемкость операций со списками. 19](#_heading=h.1opuj5n)

[85. Односвязный и двусвязный список. Способы реализации на языке Джава 19](#_heading=h.48pi1tg)

[86. Возможости Java Framework Collection. Контейнер ArrayList и его основные методы. 19](#_heading=h.2nusc19)

[87. Возможости Java Framework Collection. Контейнер LinkedList и его основные методы. 20](#_heading=h.1302m92)

[88. Возможости Java Framework Collection. Интерфейс Map и его основные методы. 20](#_heading=h.3mzq4wv)

[89. Возможости Java Framework Collection. Контейнер HashMap и его основные методы. 20](#_heading=h.2250f4o)

[90. Коллекция HashMap, принципы создания и методы работы с ней 20](#_heading=h.haapch)

[91. Использование обобщенного класса HashMap, которая реализует интерфейс Map для хранения пар ключ-значение в разработке программ. 20](#_heading=h.319y80a)

[92. Возможости Java Framework Collection. Контейнер HashSet и его основные методы. Обобщенный класс HashSet класс коллекция, наследует свой функционал от класса AbstractSet, а также реализует интерфейс Set. Что он себя представляет? 20](#_heading=h.1gf8i83)

[93. Регулярные выражения и организация работы с ними в Java. Примеры 21](#_heading=h.40ew0vw)

[94. Структура коллекций в Java Collection Framework. Иерархия интефейсов 21](#_heading=h.2fk6b3p)

[95. Одним из ключевых методов интерфейса Collection является метод Iterator<E> iterator(). Что возвращает это метод? 21](#_heading=h.upglbi)

[96. Класс Pattern и его использование 21](#_heading=h.3ep43zb)

[97. Класс Math и его использование 21](#_heading=h.1tuee74)

[98. Возможости Java Framework Collection. Интерфейс Map и его основные методы 21](#_heading=h.4du1wux)

[99. Возможости Java Framework Collection. Контейнер Hashtable его основные методы. 21](#_heading=h.2szc72q)

[100. Возможости Java Framework Collection. Интерфейс Iterator и Itarable. 22](#_heading=h.184mhaj)

[101. Работа с Датой и временем в Java. Примеры использования 22](#_heading=h.3s49zyc)

[102. Класс System. Стандартные поток ввода-вывода, предоставляемые Java. Работа со стандартами потоками вывода 22](#_heading=h.279ka65)

[103. Перегруженные методы out.println() класса System и их использование для вывода в консоль 22](#_heading=h.meukdy)

[104. Класс Scanner. Ввод и вывод данных. Стандартные потоки ввода и вывода. 23](#_heading=h.36ei31r)

[105. Использование форматированного ввода-вывода в Java. Классы для форматирования вывода. 23](#_heading=h.1ljsd9k)

[106. Понятие сериализации и ее использование в ООП программах. Использование интерфейса Serializable в программах на Джава 23](#_heading=h.45jfvxd)

[107. Какие объекты можно сериализовать? 23](#_heading=h.2koq656)

[108. Какие методы определяет интерфейс Serializable? 23](#_heading=h.zu0gcz)

[109. Что означает понятие десериализация? 24](#_heading=h.3jtnz0s)

[110. Организация работы с файлами в Джава. Класс File, определенный в пакете java.io, не работает напрямую с потоками. В чем состоит его задача? 24](#_heading=h.1yyy98l)

[111. При работе с объектом класса FileOutputStream происходит вызов метода FileOutputStream.write(), что в результате этого происходит? 24](#_heading=h.4iylrwe)

[112. Иерархия классов ввода вывода. Работа с файлами в Java. Работа с файлами. Сериализация объектов 24](#_heading=h.2y3w247)

**Вопросы по Джаве**

Тема 1. Особенности платформы Java. Синтаксис языка Java

# Парадигма объектно-ориентированного программирование. Основные принципы ООП и их реализация в языке программирования Java и С++

ООП — это одна из парадигм разработки. Парадигмой называют набор правил и критериев, которые соблюдают разработчики при написании кода. Если представить, что код — это рецепт блюда, то парадигма — то, как рецепт оформлен в кулинарной книге. Парадигма помогает стандартизировать написание кода. Это снижает риск ошибок, ускоряет разработку и делает код более читабельным для других программистов.

Принцип 1. Наследование— механизм, который позволяет описать новый класс на основе существующего (родительского). При этом свойства и функциональность родительского класса заимствуются новым классом.

Принцип 2. Абстракция означает выделение главных, наиболее значимых характеристик предмета и наоборот — отбрасывание второстепенных, незначительных.

Принцип 3. Инкапсуляция означает ограничение доступа к данным и возможностям их изменения (мы прячем какие-то важные для нас данные, которые не хотим, чтобы кто-то меня).

Принцип 4. Полиморфизм — это возможность работать с несколькими типами так, будто это один и тот же тип. При этом поведение объектов будет разным в зависимости от типа, к которому они принадлежат.

# Организация программы на Java. Основные структурные единицы. Процесс интерпретации и компиляции. Роль JVM

Java — мультиплатформенный язык программирования. Это значит, что программы, написанные на языке Java, можно выполнять на любой платформе, где установлена специальная исполняющая система Java. Такая система называется Java Virtual Machine (JVM). Для того, чтобы перевести программу из исходного кода в код, понятный JVM, нужно её скомпилировать. Код, понятный JVM называется байт-кодом и содержит набор инструкций, которые в дальнейшем будет исполнять виртуальная машина.

Компиляция — трансляция программы, составленной на исходном языке высокого уровня, в эквивалентную программу на низкоуровневом языке, близком машинному коду.

Интерпретация — пооператорный (покомандный, построчный) анализ, обработка и тут же выполнение исходной программы или запроса (в отличие от компиляции, при которой программа транслируется без её выполнения).

Роль Java Virtual Machine (JVM) заключается в выполнении Java-программ. JVM является частью исполняющей системы Java и отвечает за запуск, выполнение и управление Java-приложениями.

Основные задачи JVM:

* Интерпретация байт-кода: Java исходный код компилируется в байт-код, который затем выполняется JVM. JVM интерпретирует байт-код и выполняет соответствующие операции.
* Управление памятью: JVM управляет памятью, выделяя и освобождая память для объектов, управляя сборкой мусора и обеспечивая безопасность памяти.
* Управление потоками: JVM обеспечивает многопоточность, управляя потоками выполнения и обеспечивая их безопасное выполнение.
* Загрузка и выполнение классов: JVM загружает классы из файлов .class, проверяет их на соответствие спецификации Java и выполняет их.
* Обеспечение безопасности: JVM обеспечивает безопасность выполнения Java-приложений, например, путем проверки байт-кода на наличие потенциально опасных операций.
* Оптимизация выполнения: JVM может выполнять оптимизацию байт-кода во время выполнения для повышения производительности.

JVM является ключевым компонентом платформы Java и обеспечивает переносимость Java-приложений, поскольку одна и та же программа может быть выполнена на различных операционных системах и аппаратных платформах, поддерживающих JVM.

# Структурирование Java приложения, пакеты. Уровни доступа и видимости

Пакеты - это способ логической группировки классов или комплект программного обеспечения, который может распространяться независимо и применяться в сочетании с другими пакетами.

Например, java.lang, java.util, java.io

Пакеты могут входить следующие компоненты:

- классы

- интерфейсы

- вложенные пакеты

Для создания пакета в Java используется ключевое слово package.

Например, package com.example.myapp; объявляет пакет com.example.myapp

Модификаторы доступа — это чаще всего ключевые слова, которые регулируют уровень доступа к разным частям твоего кода.

Всего в Java есть четыре модификатора доступа.

* private;
* default (package visible);
* protected;
* public.

private — наиболее строгий модификатор доступа. Он ограничивает видимость данных и методов пределами одного класса.

protected, будут видны:

* в пределах всех классов, находящихся в том же пакете, что и наш;
* в пределах всех классов-наследников нашего класса.

модификатор default или, как его еще называют, package visible. Он не обозначается ключевым словом, поскольку установлен в Java по умолчанию для всех полей и методов.

Уровень доступа public означает, что член класса доступен из любого места.

# Примитивные и ссылочные типы данных. Использование мехаизмов автоупаковки и автораспаковки. Операция приведения типов. Понижающее и повышающее приведение.

К примитивным типам данных относятся типы для представления числовых значений и также логический тип:

* char —16-битовый символ Unicode
* byte — 8-битовое целое число со знаком
* short — 16-битовое целое число со знаком
* int — 32-битовое целое число со знаком
* long — 64-битовое целое число со знаком
* float — 32-битовое число с плавающей точкой
* double — 64-битовое число с плавающей точкой
* boolean — хранят значения типа "true/false".

Ссылочными типы — это переменные типа класс, которые используются для ссылок на объекты:

К объектным типам относят:

1) Классы – переменные типа класс

2) Интерфейсы – ссылки интерфейсного типа

3) Перечисления –поименованные константы

4) Типы оболочки соответствуют каждому примитивному типу

Автоупаковка является механизмом для скрытого преобразования примитивных типов данных в соответствующие классы-оболочки (объекты).

Integer obj;

int num = 42;

obj = num;

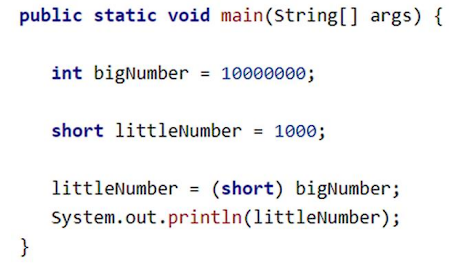
Присваивание создает соответствующий объект Integer

Обратный процесс соответственно – Распаковка т.е. процесс преобразования объектов в соответствующие им примитивные типы.

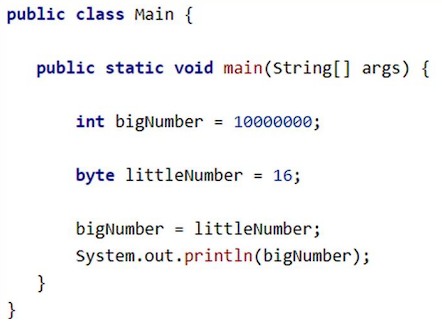
int in = 0;

in = new Integer(9);

Приведение типов для интов работает так:



если мы делаем повышающее приведение, то можно ещё и так сделать



Но нужно учитывать что разрядная сетка уменьшится, можно потерять значение.

Char можно также перевести к другим типам, ведь это просто цифра в юникоде обозначающая символ.

Понижающее приведение - это когда ты приводишь int к byte, можно потерять значение, так как уменьшается разрядная сетка

Повышающее приведение - это когда ты приводишь byte к int, разрядная сетка повышается, большие цифры хранить можно

# Этапы проектирования, разработки и отладки ООП программ. Понятие конвенции кода языка и стиля программирования.

* Проектирование:

- Определение требований к программе.

- Анализ предметной области и выделение объектов и классов.

- Проектирование структуры классов и их взаимодействия.

* Разработка:

- Написание кода классов и методов.

- Создание объектов и их взаимодействие.

- Реализация алгоритмов и логики программы.

* Отладка:

- Поиск и исправление ошибок в программе.

- Тестирование функциональности и корректности работы программы.

Конвенции кода и стили программирования включают в себя следующие основные принципы:   
- Именование переменных, классов, методов и пакетов: в Java рекомендуется использовать CamelCase нотацию. Например, MyClass, myMethod, myVariable.

- Отступы: используются отступы в 4 пробела (табуляция) для обозначения вложенности кода.

- Комментирование кода: для улучшения читаемости и понимания кода следует использовать комментарии, описывающие основные функции и блоки кода.

- Использование классов и объектно-ориентированного подхода: в Java стиль программирования ориентирован на использование объектов и классов для реализации функциональности.

- Использование библиотек и фреймворков: Java имеет обширный набор стандартных библиотек, поэтому рекомендуется использовать их при разработке.

Следование этим конвенциям и стилю программирования помогает создавать более читаемый, структурированный и легко поддерживаемый код.

# Массивы в Java. Способы создания массивов. Индексы. Размерность массивов. Доступ к элементам массива и примеры использования

Массив — это структура данных, в которой хранятся элементы одного типа. Его можно представить, как набор пронумерованных ячеек, в каждую из которых можно поместить какие-то данные (один элемент данных в одну ячейку). Доступ к конкретной ячейке осуществляется через ее номер. Номер элемента в массиве также называют индексом.

Способы создания массивов:

* Способ 1: С помощью оператора new и указания размера массива:

int[] sample = new int[10];

* Способ 2: С помощью инициализации при объявлении:

int[] sample = {12, 56, 7, 34, 89, 23, 9};

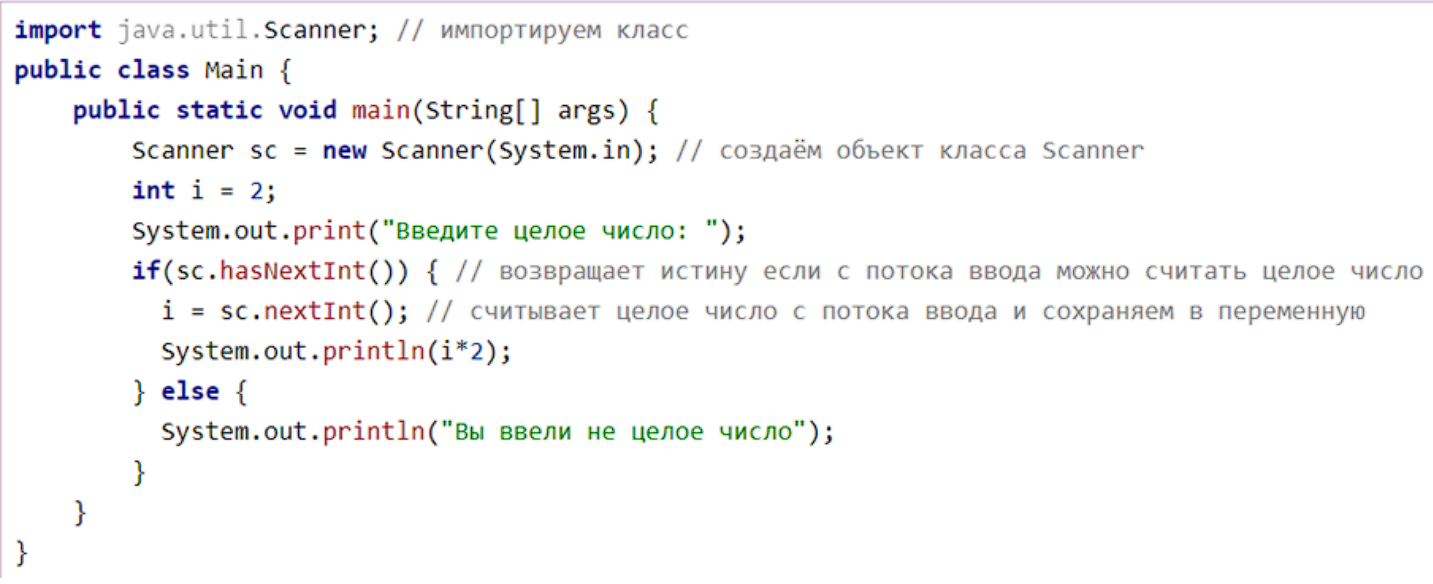
**Размерность массивов** – количество индексов, требуемое для однозначной адресации всех элементов.

Применение максимально обширно, можно хранить в них функции, можно хранить данные, упорядочивать данные.

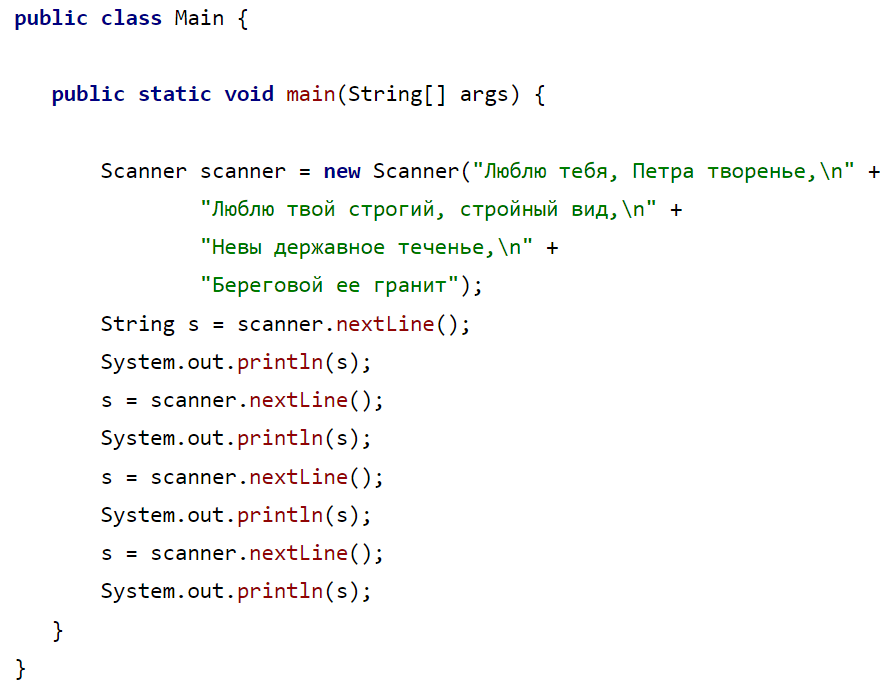
# Класс Scanner и его использование для чтения стандартного потока ввода, конструктор класса Scanner

Класс Scanner предоставляет удобные методы для чтения входных значений различных типов. Объект Scanner можно настроить, таким образом, чтобы читать входные данные из различных источников, включая значения опечаток пользователей на клавиатуре. Например, из строки, из файла, из консоли. Далее он распознает эту информацию и обрабатывает нужным образом.

Стандартный поток ввода (клавиатура) в Java представлен объектом — System.in. А стандартный поток вывода (дисплей) — уже знакомым вам объектом System.out.



Сам конструктор может принимать и, например строку, с которой он будет работать.



# Методы класса Scanner nextLine(), nextInt(), hasNextInt(), hasNextLine() и их использование для чтения ввода пользователя с клавиатуры

Метод nextLine() обращается к источнику данных (нашему тексту с четверостишием), находит там следующую строку, которую он еще не считывал (в нашем случае — первую) и возвращает ее. После чего мы выводим ее на консоль:

Метод nextInt() считывает и возвращает введенное число. В нашей программе он используется для того, чтобы присвоить значение переменной number.

hasNextInt() — метод проверяет, является ли следующая порция введенных данных числом, или нет (возвращает, соответственно, true или false).

hasNextLine() — проверяет, является ли следующая порция данных строкой.

# Виды типов данных в Джава. Примитивные типы данных, объявление и присваивание переменных. Константы в Джава: объявление и использование константы

Приложение в Java состоит из классов и объектов. Рассмотрим, какими бывают Java переменные:

* переменные объекта; (создать объект new и вызвать переменную)
* локальные переменные;
* переменные класса. (static)

Примитивные типы данных — это основные типы данных, которые существуют в языке Java. Они используются для представления простых значений, таких как числа, символы и логические значения.

* Boolean — представляет значение true или false.
* int — представляет 32-битное целое число.
* long — представляет 64-битное целое число.
* float — представляет 32-битное число с плавающей запятой одинарной точности.
* double — представляет 64-битное число с плавающей запятой двойной точности.

В Java константы — это переменные, обозначенные специальным словом — final:

public final double Pi = 3.14;

# Виды типов данных в Джава. Объектные типы данных

Есть целочисленные, с плавающей точкой, логические и символьные примитивные типы данных.

Объектные(ссылочные) типы данных: экземпляры классов, массив

* Ссылочный тип данных (Reference) используется для хранения ссылок на объекты, а не фактического значения объекта.
* Ссылочная переменная содержит адрес памяти объекта в Java, наиболее распространенным ссылочным типом данных является объект.
* Массив — это набор похожих типов данных в Java. Он может быть любого типа, например, int[], String[] или Object[].
* Строка — это последовательность символов. Это объект класса String в Java.
* Объект — это экземпляр класса, базовая единица данных, которую можно использовать для хранения данных и поведения в форме методов и переменных.

Величины ссылочного типа - это ссылки на объекты. Переменная, имеющая ссылочный тип, содержит ссылку на объект.

# Объявление и использование бестиповых переменных в Джава

В Java, объявление и использование бестиповых переменных происходит с использованием ключевого слова var. Это позволяет компилятору самостоятельно определить тип переменной на основе присвоенного значения.

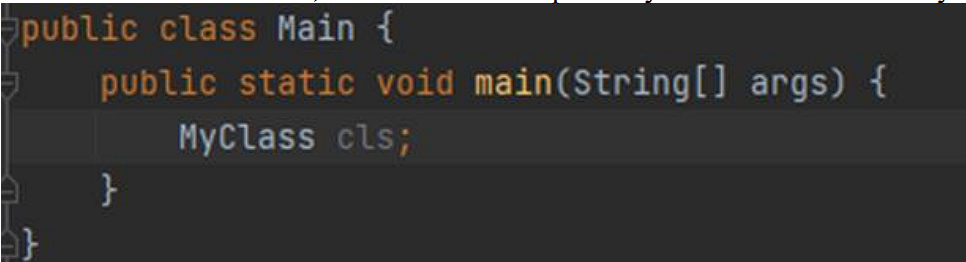
Пример объявления и использования бестиповых переменных в Java:

var number = 10; // компилятор определит тип переменной как int

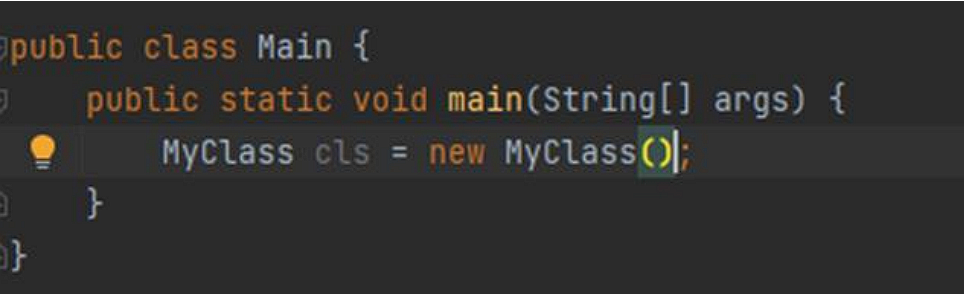
var name = "John"; // компилятор определит тип переменной как String

# Объявление переменных типа класс и их инициализация

Допустим у нас есть класс MyClass Если мы напишем так, то мы объявим переменную ссылочного типа MyClass



Если мы напишем так то инициализируем



# Массивы в Джава, как объектные типы данных, контроль доступа за выход за границы массива. Объявление и инициализация массивов, длина массива, получение доступа к элементу массива

Выход за границы массива За выходом за границы массивы следит интерпретатор. Если случился выход за границы массива, то вылетит исключение ArrayIndexOutOfBoundsException.

а) Объявление:

1) typeData Name [] = new typeData[length]; - стиль Java

2) typeData []Name = new typeData[length]; - стиль из С++

б) Инициализация:

1) int c [] = new int[2];

c[0] = 1;

c[1] = 2;

2) int c [] = new int [] {1, 2, 3, 4};

3) int c [] = {1, 2, 3, 4};

в) Длина массива

1. int length1 = с.length; // Длина массива с = 4

# Способы объявления массивов в Джава, использование операции new для выделения памяти для элементов массива. Объявление с инициализацией, объявление массива определенного размера без инициализации.

Объявление с использованием операции new:

тип\_данных[] имя\_массива = new тип\_данных[размер];

int[] numbers = new int[5];

В этом примере мы объявляем массив numbers, который может содержать 5 элементов типа int. Память для элементов массива выделяется с помощью операции new.

Объявление с инициализацией:

String[] fruits = {"яблоко", "банан", "апельсин"};

В этом примере мы объявляем массив fruits и инициализируем его тремя строковыми значениями.

Объявление массива определенного размера без инициализации:

double[] temperatures = new double[10];

В этом примере мы объявляем массив temperatures, который может содержать 10 элементов типа double. Элементы массива не инициализируются при объявлении и по умолчанию получают значения по умолчанию для своего типа данных (ноль для числовых типов, false для boolean, null для ссылочных типов).

# Инициализация полей класса и локальных переменных (отличие), инициализатор и статический инициализатор (когда вызывается).

В Java есть разница между инициализацией полей класса и локальных переменных.

**Инициализация полей класса:**

Поля класса являются переменными, объявленными внутри класса, но вне методов.

Поля класса могут быть инициализированы непосредственно при их объявлении или в конструкторе класса.

Поля класса имеют значения по умолчанию, если они не были явно инициализированы или не имеют конструктора, который бы их инициализировал.

**Инициализация локальных переменных:**

Локальные переменные являются переменными, объявленными внутри методов или блоков кода.

Локальные переменные должны быть инициализированы перед использованием.

Локальные переменные не имеют значений по умолчанию и должны быть явно инициализированы до первого использования.

Инициализаторы и статические инициализаторы - это способы инициализации полей класса и статических переменных.

**Инициализатор:**

Инициализатор - это блок кода, который используется для инициализации полей класса.

Инициализатор не имеет имени и заключен в фигурные скобки {}.

Инициализатор выполняется каждый раз перед вызовом конструктора класса.

Инициализатор может быть использован для выполнения сложных инициализаций или общей инициализации для нескольких конструкторов.

**Статический инициализатор:**

Статический инициализатор - это блок кода, который используется для инициализации статических полей или выполнения других статических операций.

Статический инициализатор имеет ключевое слово static и заключен в фигурные скобки {}.

Статический инициализатор выполняется при первой загрузке класса и выполняется только один раз.

**Примеры:**



# Циклические конструкции в Java. Использование циклов для работы с массивами. Использование итераторов для обработки массивов. Использование итераторов для работы с коллекциями

В Java существует несколько видов циклических конструкций, которые можно использовать для работы с массивами и коллекциями.

1. Цикл for для работы с массивами:

int[] array = {1, 2, 3, 4, 5};

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

System.out.println(array[i]);

}

2. Цикл for-each для работы с массивами:

int[] array = {1, 2, 3, 4, 5};

for (int num : array) {

System.out.println(num);

}

3. while - цикл с предусловием:

int i = 0;

while (i < 10) {

// Действия, выполняемые в цикле

i++;

4. do-while - цикл с поступлением

int i = 0;

do {

i++;

// Действия, выполняемые в цикле

while (i < 10)

5. Использование итераторов для обработки массивов:

import java.util.Iterator;

import java.util.Arrays;

int[] array = {1, 2, 3, 4, 5};

Iterator<Integer> iterator = Arrays.stream(array).iterator();

while (iterator.hasNext()) {

System.out.println(iterator.next());

}

6. Использование итераторов для работы с коллекциями:

import java.util.ArrayList;

import java.util.Iterator;

ArrayList<String> list = new ArrayList<>();

list.add("one");

list.add("two");

list.add("three");

Iterator<String> iterator = list.iterator();

while (iterator.hasNext()) {

System.out.println(iterator.next());

}

# Статические поля и методы. Класс Math, его основные методы.

Статические методы и переменные, используя модификатор static. Это связывает метод или переменную с классом, а не с объектом этого класса. Статические методы иногда называют методами класса и статические переменные иногда называют переменные класса.

Статическая переменная

Каждый объект имеет свое собственное пространство данных, но если переменная объявлена как статическая, то существует только одна копия переменной для всех объектов

Статические методы не могут ссылаться на переменные экземпляра, поскольку пока объект не существует, то переменных экземпляра тоже не существует. Тем не менее, статический метод может ссылаться на статические переменные или локальные переменные. Статические методы и статические переменные часто работают вместе.

Класс Math содержит множество математических функций, все они возвращают тип double и берут аргументы типа double. Все методы и поля класса Math являются статическими.

* Math.abs(x) - возвращает абсолютное значение числа x
* Math.sin(x), Math.cos(x), Math.tan(x) - тригонометрические функции - синус, косинус и тангенс угла x (значение угла в радианах)
* Math.pow(x, y) - возведение числа x в степень y
* Math.sqrt(x) - возвращает квадратный корень числа x
* Math.random() - возвращает случайное число в диапазоне от 0.0 (включительно) до 1.0 (исключительно)
* Math.max(x, y) - возвращает наибольшее из двух чисел x и y

# Понятие перечисления. Состав и приемы использования в ООП программах на Jаva

Перечисление — это тип данных, который представляет собой набор констант с ограниченными возможностями.

Для создания перечисления служит ключевое слово enum.

enum Season {winter, spring, summer, fall};

Тема 2. Реализация ООП в Java.

# Понятие класса. Определение, инициализация. Модификаторы доступа. Константы и переменные. Объявление классов.

**Класс** - это шаблонная конструкция, которая позволяет описать в программе объект, его свойства (атрибуты или поля класса) и поведение (методы класса).

public class Cat {

String name;

int age;

}

**Инициализация** - это когда мы впервые задаем переменной какое-либо значение. Например мы объявили переменную int a;. Мы ее не инициализировали - другими словами, не задали начальное, стартовое значение. Если попытаться вывести её значение мы получим ошибку "переменная не инициализирована".

**Модификаторы доступа** - ключевые слова, которые регулируют уровни доступа к разным частям программы

а) Private

Модификатор доступа private ограничивает доступ из вне к свойствам и методам класса.

Данные и методы с таким модификатором нельзя наследовать.

б) Protected

Поля и методы с модификатором доступа protected будут видны в пределах одного пакета или в пределах классов наследников

в) package visible/default

Если у переменной, метода, класса и т.д. нет никакого модификатора, Java по умолчанию присваивает им модификатор default

г) Public

Модификатор public разрешает доступ к полям и методам класса из вне.

**Переменная** – это часть памяти, которая может содержать значение данных, она имеет тип данных. Переменные обычно используются для хранения информации, необходимой вашей программе для выполнения своей работы. Это может быть информация любого типа, от текстов, кодов до цифр, временных результатов многоступенчатых расчетов и т. д.

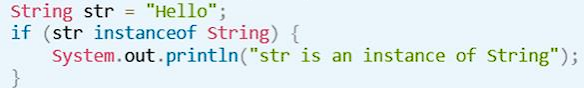
В Java имеется **несколько типов констант**, соответствующих его встроенным типам, а именно: константа null нулевого типа; логические константы true (истина) и false (ложь) типа boolean; символьные константы типа char, например, 'a' или '\t'; строковые константы класса, например, "Привет всем!"; целые константы типов int и long, например, 111 или -2L; плавающие константы типов float и double, например, 3.1415926.

# Получение информации о типе. Создание экземпляров классов. Вызов методов класса Объявление класса на Джава, пример объявления

**Рефлексия** – это получение информации о типах во время выполнения программы.

Java для получения информации о типе объекта или переменной можно использовать оператор instanceof или метод getClass(). Давайте рассмотрим оба способа.

Использование оператора instanceof:



Использование метода getClass():



Создание экземпляров класса

Для создания экземпляра класса в Java используется ключевое слово new, после которого идет вызов конструктора класса.

Пример создания экземпляра класса Car:

Car myCar = new Car("Toyota", "Camry", 2015, 15000);

Вызов методов класса:

После создания экземпляра класса вы можете вызвать его методы, используя оператор точки . и имя метода

myCar.drive(100); // Передача аргумента 100 в метод drive

Объявление класса в Java:



В этом примере:

- Класс называется Person.

- У класса есть два приватных поля: name (типа String) и age (типа int).

- Есть конструктор класса, который инициализирует поля name и age.

- Есть метод greet, который выводит приветствие с использованием имени и возраста.

# ООП в Java. Понятие объекта. Что представляет собой Java приложение с точки зрения ООП. Основные характеристики объектов в Java.

Объектно-ориентированное программирование (ООП) в Java основано на понятии объекта. Объект представляет собой экземпляр класса, который содержит данные (полей) и методы для работы с этими данными.

Java приложение с точки зрения ООП представляет собой набор взаимодействующих объектов, которые обмениваются сообщениями и выполняют определенные действия. Каждый объект в Java является экземпляром определенного класса, который определяет его структуру и поведение.

Основные характеристики объектов в Java включают:

* Наследование— механизм, который позволяет описать новый класс на основе существующего (родительского). При этом свойства и функциональность родительского класса заимствуются новым классом.
* Абстракция означает выделение главных, наиболее значимых характеристик предмета и наоборот — отбрасывание второстепенных, незначительных.
* Инкапсуляция означает ограничение доступа к данным и возможностям их изменения (мы прячем какие-то важные для нас данные, которые не хотим, чтобы кто-то меня).
* Полиморфизм — это возможность работать с несколькими типами так, будто это один и тот же тип. При этом поведение объектов будет разным в зависимости от типа, к которому они принадлежат.

# Конструкторы, назначение и использование. Конструктор с параметром, конструктор по умолчанию.

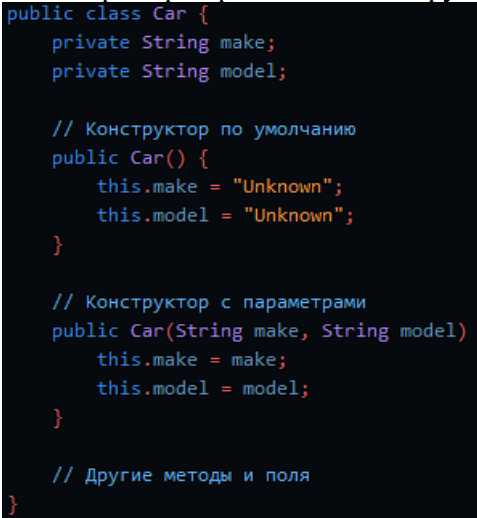
В языке программирования Java конструкторы используются для инициализации объектов при их создании. Конструктор представляет собой метод, который имеет то же имя, что и класс, в котором он определен, и не возвращает никакого значения. Основной целью конструктора является установка начальных значений полей объекта.

В работе с конструкторами в Java следует учитывать следующие особенности:

1. Конструктор по умолчанию: в Java каждый класс по умолчанию имеет конструктор без параметров, который называется конструктором по умолчанию. Если вы не определяете явно какие-либо конструкторы, компилятор Java автоматически добавит конструктор по умолчанию.

2. Конструктор с параметрами: помимо конструктора по умолчанию, класс может также иметь конструкторы с параметрами. Они предоставляют возможность передачи параметров при создании объектов, что позволяет легко инициализировать и настраивать объекты сразу при их создании.

Пример определения конструкторов в классе на Java:



# Конструкторы, назначение и использование. Вызов конструктора родительского класса, неявный вызов конструктора родительского класса, порядок инициализации экземпляра Java класса.

Конструкторы в Java - это специальные методы, которые используются для инициализации объектов.

Вызов конструктора родительского класса в Java осуществляется с помощью ключевого слова super.

public class SubClass extends SuperClass {

public SubClass() {

super(); // Вызов конструктора суперкласса

// Другие действия

}

}

Неявный вызов конструктора родительского класса:

Если конструктор суперкласса не вызывается явно в конструкторе подкласса, то будет вызван конструктор по умолчанию (без параметров) суперкласса.

Порядок инициализации экземпляра Java класса:

1. Выделение памяти для объекта.

2. Инициализация переменных экземпляра по умолчанию (нулевое значение для ссылочных типов, 0 для числовых типов и false для boolean).

3. Выполнение блока инициализации экземпляра (если он есть).

4. Вызов конструктора (явного или неявного) суперкласса.

5. Инициализация переменных экземпляра в соответствии с кодом конструктора.

# Использование языка UML для проектирования и документирования объектно-ориентированных программ. Основные UML диаграммы для отображения отношений между классами в ООП программах

**UML** — это язык графического описания для объектного моделирования в области разработки программного обеспечения, моделирования бизнес-процессов, системного проектирования и отображения организационных структур.

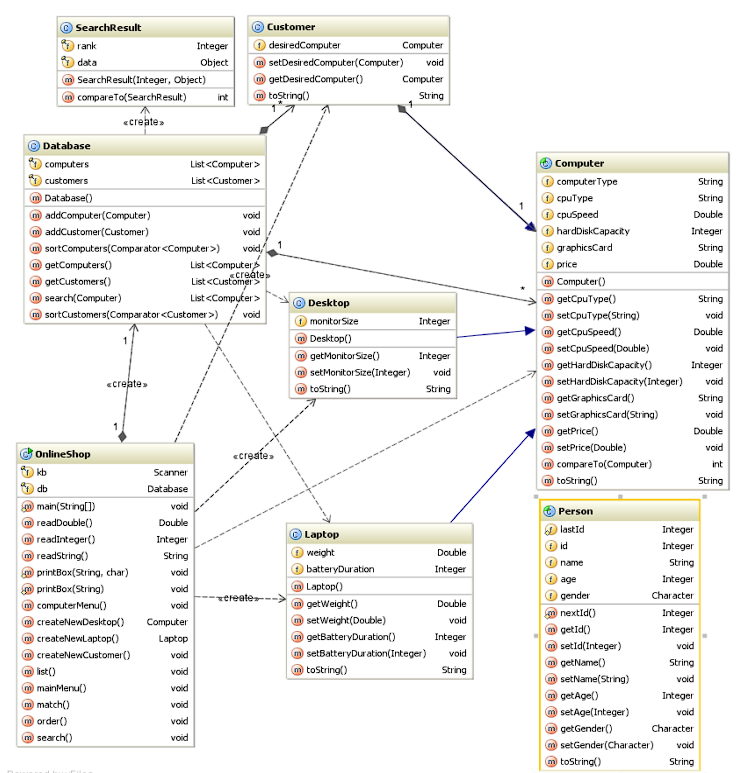
Использование UML-диаграмм позволит развить навыки продумывания архитектуры приложения и поможет выявить недостатки структуры классов на раннем этапе, а не когда вы уже потратите день на реализацию неправильной модели.

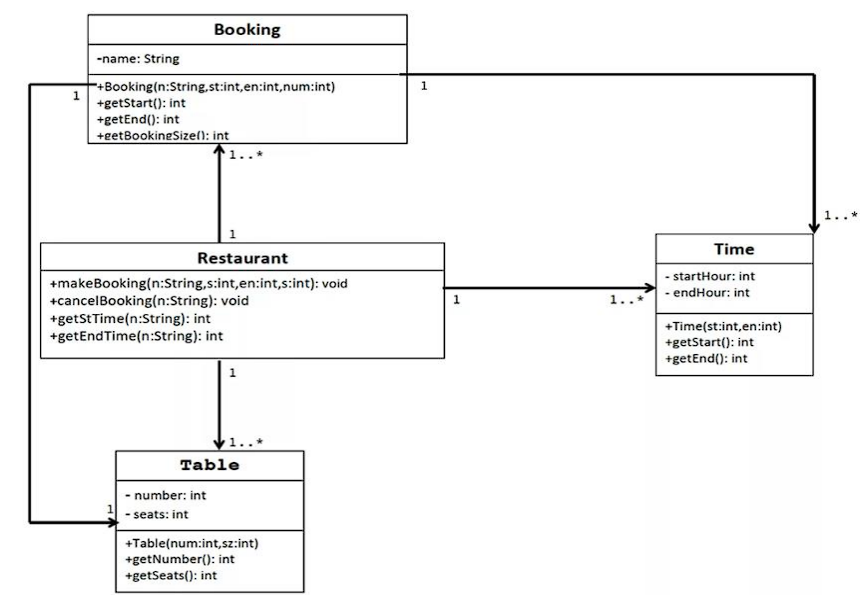
Существует два основных типа диаграмм UML: структурные диаграммы и поведенческие диаграммы (а внутри этих категорий имеется много других). Эти варианты существуют для представления многочисленных типов сценариев и диаграмм, которые используют разные типы людей.

**Структурные диаграммы** представляют статическую структуру программного обеспечения или системы, они также показывают различные уровни абстракции и реализации. Они используются, чтобы помочь визуализировать различные структуры, составляющие систему, например, базу данных или приложение. Они показывают иерархию компонентов или модулей и то, как они связаны и взаимодействуют между собой. Эти инструменты обеспечивают руководство работы и гарантируют, что все части системы функционируют так, как задумано по отношению ко всем остальным частям.

**Поведенческие диаграммы** Основное внимание здесь уделяется динамическим аспектам системы программного обеспечения или процесса. Эти диаграммы показывают функциональные возможности системы и демонстрируют, что должно происходить в моделируемой системе.







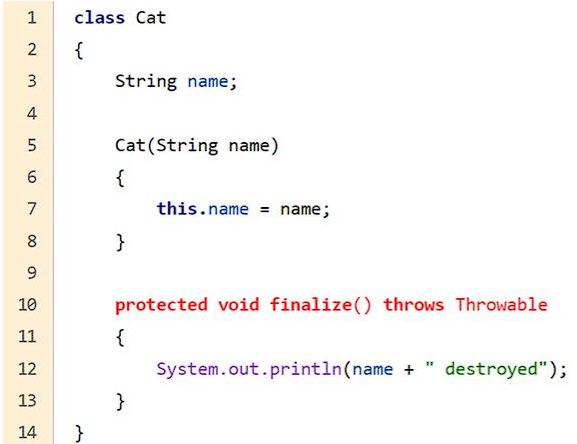
# Управление памятью в Java и C++, процесс освобождения памяти, занимаемой объектом. Метод finalize.

Управление памятью в Java и C++ осуществляется с помощью сборщика мусора (garbage collector) в Java и операторов new и delete в C++.

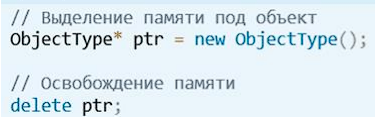
В Java сборщик мусора автоматически освобождает память, занимаемую объектами, которые больше не используются. Когда объект становится недоступным для программы, сборщик мусора автоматически освобождает память, занимаемую этим объектом.

В C++ разработчику нужно самостоятельно управлять памятью с помощью операторов new и delete. Оператор new выделяет память под объект, а оператор delete освобождает эту память после того, как объект больше не нужен.

Метод finalize() в Java позволяет разработчику указать дополнительные действия перед тем, как объект будет удален сборщиком мусора. Однако использование метода finalize() рекомендуется избегать, так как его вызов не гарантирован, а сам механизм сборки мусора может привести к нежелательным задержкам в работе программы. (Фактически этот метод – противоположность конструктору. В нем можно освобождать ресурсы, используемые объектом. )



В C++ нет эквивалента метода finalize(), поскольку управление памятью является ответственностью программиста, и он сам должен следить за освобождением памяти после использования объектов.



# Понятие рекурсии, виды рекурсии и ее использование. Реализация Рекурсивных алгоритмов в ООП программах

Рекурсия – это метод вызывает сам себя. Виды рекурсии: прямая рекурсия (метод вызывает сам себя), косвенная рекурсия (несколько методов вызывают друг друга), хвостовая рекурсия (вызов рекурсивного метода является последней операцией в методе).

# Оператор new. Понятие ссылки и указателя на объект. Реализация в С++ и Java. Время жизни объекта

Оператор new используется для создания нового объекта в Java. В Java **ссылка** – это переменная, которая указывает на объект в памяти. В C++ **указатель** – это переменная, которая содержит адрес объекта в памяти. Время жизни объекта определяется его областью видимости и способом выделения памяти.

Реализация:  
В С++  
int\* p = new int(5);

После завершения работы с объектом, память должна быть освобождена оператором delete.

delete p;

В Java

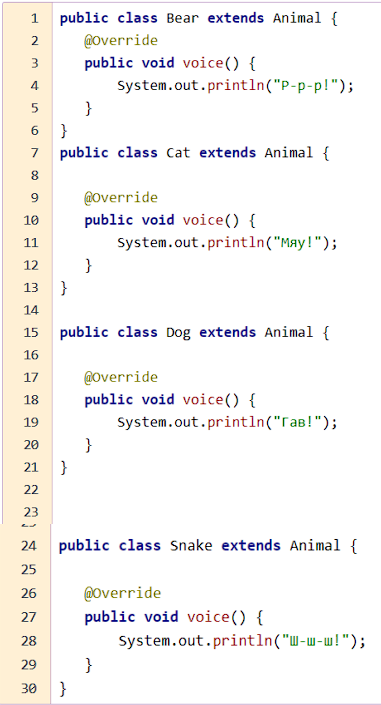
Integer i = new Integer(40);

# Переопределение методов в Java, абстрактные методы.

В Java переопределение методов позволяет классам наследникам изменять реализацию методов, унаследованных от их суперклассов. Это позволяет создавать более специфичные версии методов для конкретных классов.

- Для переопределения метода в подклассе нужно использовать аннотацию @Override над методом, который мы хотим переопределить.

- Подкласс должен иметь метод с тем же именем, параметрами и возвращаемым типом, как унаследованный метод.



**Абстрактный** **класс** - это **класс**, который нельзя реализовать. Он объявляется при помощи ключевого слова abstract, присутствующего в сигнатуре этого метода, и это ключевое слово обязательно должно быть включено в абстрактный класс



Абстрактные методы:

- Абстрактный метод не имеет тела (реализации) и должен быть объявлен с помощью ключевого слова abstract.

- Класс, содержащий хотя бы один абстрактный метод, должен быть объявлен как абстрактный с помощью ключевого слова abstract.

- Подкласс абстрактного класса должен реализовать (переопределить) все абстрактные методы или сам быть объявлен как абстрактный.

# Преобразование ссылочных типов в Java, instanceof (экземпляр класса).

В Java преобразование ссылочных типов осуществляется с помощью операторов приведения типов. Оператор приведения типов позволяет явно указать компилятору, что объект одного типа должен быть рассмотрен как объект другого типа.

Преобразование ссылочных типов можно осуществить двумя способами: неявное преобразование (upcasting) и явное преобразование (downcasting).

1. Неявное преобразование (upcasting):

Неявное преобразование возможно, когда происходит приведение объекта к более общему типу в иерархии наследования.

Например:

class Parent {}

class Child extends Parent {}

Child child = new Child();

Parent parent = child; // Неявное преобразование (upcasting)

2. Явное преобразование (downcasting):

Явное преобразование используется, когда нужно привести объект к более конкретному типу в иерархии наследования. Для этого используется оператор приведения типов:

class Parent {}

class Child extends Parent {}

Parent parent = new Child();

Child child = (Child) parent; // Явное преобразование (downcasting)

Оператор instanceof используется для проверки, является ли объект экземпляром определенного класса или интерфейса. Например:

Parent parent = new Child();

if (parent instanceof Child) {

System.out.println("parent is an instance of Child");

}

https://javarush.com/groups/posts/2018-kak-rabotaet-operator-instanceof

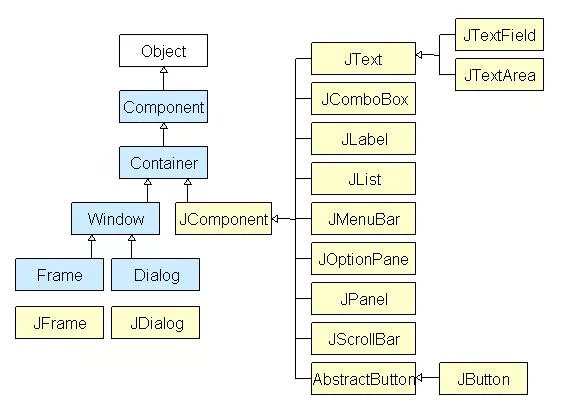
# Графическая подсистема. Основы AWT, Swing components. Событийная модель при программировании GUI в ООП программах

Графическая подсистема в Java предоставляет два основных набора инструментов для создания графических пользовательских интерфейсов (GUI): AWT (Abstract Window Toolkit) и Swing.

AWT была первой библиотекой для создания GUI в Java, но она имеет ограниченную функциональность и внешний вид зависит от операционной системы. Swing была разработана для полной замены AWT и предоставляет более широкие возможности для создания настраиваемого и кросс-платформенного пользовательского интерфейса.

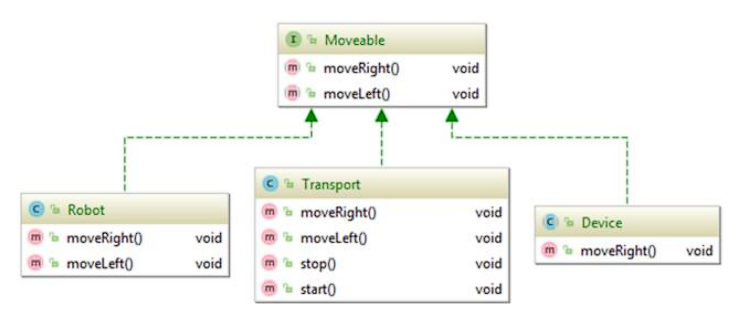
В AWT и Swing используются компоненты (кнопки, текстовые поля, панели и т.д.), которые могут быть использованы для создания интерфейса приложения. Эти компоненты могут быть настроены и оформлены с помощью различных методов и свойств.

Событийная модель при программировании GUI в ООП программах в Java основана на обработке событий. Когда пользователь взаимодействует с графическим интерфейсом (например, кликает на кнопку или вводит текст), это генерирует событие. Для обработки этих событий в Java используется слушатель (listener) - объект, который реагирует на определенные события.



# Использование языка UML для проектирования и документирования объектно-ориентированных программ. Основные UML диаграммы для отображения отношений между классами в ООП программах

Преимущества интерфейсов Существуют по крайней мере три веские причины использовать интерфейсы: • они используется для достижения абстракции. • Благодаря интерфейсам мы можем поддерживать механизм множественного наследования. • они использовать для достижения слабой связанности кода (low coupling code) UML диаграмма трех классов и одного интерфейса. Как видно из отношений на схеме, все три класса реализуют общий интерфейс.



# ООП в Java. Понятие объекта. Что представляет собой Java приложение с точки зрения ООП. Основные характеристики объектов в Java.

В объектно-ориентированном программировании (ООП), объект представляет собой конкретный экземпляр класса, который содержит данные (поля) и методы для работы с этими данными. В Java приложение с точки зрения ООП представляет собой набор взаимодействующих объектов, каждый из которых выполняет определенную функцию.

Основные характеристики объектов в Java:

* Наследование— механизм, который позволяет описать новый класс на основе существующего (родительского). При этом свойства и функциональность родительского класса заимствуются новым классом.
* Абстракция означает выделение главных, наиболее значимых характеристик предмета и наоборот — отбрасывание второстепенных, незначительных.
* Инкапсуляция означает ограничение доступа к данным и возможностям их изменения (мы прячем какие-то важные для нас данные, которые не хотим, чтобы кто-то меня).
* Полиморфизм — это возможность работать с несколькими типами так, будто это один и тот же тип. При этом поведение объектов будет разным в зависимости от типа, к которому они принадлежат.

# Модификатор доступа или видимости в Джава, виды и использование. Использования this для доступа к компонентам класса.

**Модификаторы** – ключевые слова, которые Вы добавляете при инициализации для изменения значений. Язык Java имеет широкий спектр модификаторов, основные из них:

• модификаторы доступа;

• модификаторы класса, метода, переменной и потока, используемые не для доступа.

Чтобы использовать модификатор в Java, нужно включить его ключевое слово в определение класса, метода или переменной. Модификатор должен быть впереди остальной части оператора.

Java предоставляет ряд модификаторов доступа, чтобы задать уровни доступа для классов, переменных, методов и конструкторов. Существует четыре доступа:

• Видимый в пакете (стоит по умолчанию и модификатор не требуются).

• Видимый только для класса (private).

• Видимый для всех (public).

• Видимый для пакета и всех подклассов (protected).

**Модификатор доступа по умолчанию** – означает, что мы явно не объявляем модификатор доступа в Java для класса, поля, метода и т.д.

Переменная или метод, объявленные без модификатора контроля доступа доступны для любого другого класса в том же пакете. Поля в интерфейсе неявно являются public, static, final, а методы в интерфейсе по умолчанию являются public.

**Модификатор private** – методы, переменные и конструкторы, которые объявлены как private в Java могут быть доступны только в пределах самого объявленного класса.

Модификатор доступа private является наиболее ограничивающим уровнем доступа. **Класс и интерфейсы не могут быть private.** Переменные, объявленные как private, могут быть доступны вне класса, если получающие их открытые (public) методы присутствуют в классе (ниже смотрите пример и пояснения).

Использование модификатора private в Java является основным способом, чтобы скрыть данные.

**Модификатор public** – класс, метод, конструктор, интерфейс и т.д. объявленные как public могут быть доступны из любого другого класса. Поэтому поля, методы, блоки, объявленные внутри public класса могут быть доступны из любого класса, принадлежащего к "вселенной" Java.

Тем не менее, если к public классу в другом пакете мы пытаемся получить доступ, то public класс приходится импортировать.

Благодаря наследованию классов, в Java все публичные (public) методы и переменные класса наследуются его подклассами.

**Модификатор protected** – переменные, методы и конструкторы, которые объявляются как protected в суперклассе, могут быть доступны только для подклассов в другом пакете или для любого класса в пакете класса protected.

Модификатор доступа protected в Java не может быть применен к классу и интерфейсам. Методы и поля могут быть объявлены как protected, однако методы и поля в интерфейсе не могут быть объявлены как protected.

Доступ protected дает подклассу возможность использовать вспомогательный метод или переменную, предотвращая неродственный класс от попыток использовать их.

**Модификатор static** – применяется для создания методов и переменных класса.

Ключевое слово **static** используется для создания переменных, которые будут существовать независимо от каких-либо экземпляров, созданных для класса. Только одна копия переменной static в Java существует вне зависимости от количества экземпляров класса. Статические переменные также известны как переменные класса. В Java локальные переменные не могут быть объявлены статическими (static).

Ключевое слово **static** используется для создания методов, которые будут существовать независимо от каких-либо экземпляров, созданных для класса.

В Java статические методы или методы static не используют какие-либо переменные экземпляра любого объекта класса, они определены. Методы static принимают все данные из параметров и что-то из этих параметров вычисляется без ссылки на переменные.

Переменные и методы класса могут быть доступны с использованием имени класса, за которым следует точка и имя переменной или метода.

Модификатор **final** – используется для завершения реализации классов, методов и переменных.

Переменная final может быть инициализирована только один раз. Ссылочная переменная, объявленная как final, никогда не может быть назначен для обозначения другого объекта. Однако данные внутри объекта могут быть изменены. Таким образом, состояние объекта может быть изменено, но не ссылки. С переменными в Java модификатор final часто используется со static, чтобы сделать константой переменную класса.

Метод final не может быть переопределен любым подклассом. Как упоминалось ранее, в Java модификатор final предотвращает метод от изменений в подклассе. Главным намерение сделать метод final будет то, что содержание метода не должно быть изменено стороне.

Основная цель в Java использования класса объявленного в качестве final заключается в предотвращении класс от быть подклассом. Если класс помечается как final, то ни один класс не может наследовать любую функцию из класса final.

Модификатор **abstract** – используется для создания абстрактных классов и методов.

Класс abstract не может создать экземпляр. Если класс объявлен как abstract, то единственная цель для него быть расширенным.

Класс не может быть одновременно abstract и final, так как класс final не может быть расширенным. Если класс содержит абстрактные методы, то он должен быть объявлен как abstract. В противном случае будет сгенерирована ошибка компиляции. Класс abstract может содержать как абстрактные методы, а также и обычные.

Метод abstract является методом, объявленным с любой реализацией. Тело метода (реализация) обеспечивается подклассом. Методы abstract никогда не могут быть final или strict.

Любой класс, который расширяет абстрактный класс должен реализовать все абстрактные методы суперкласса, если подкласс не является абстрактным классом. Если класс в Java содержит один или несколько абстрактных методов, то класс должен быть объявлен как abstract. Абстрактный класс не обязан содержать абстрактные методы. Абстрактный метод заканчивается точкой с запятой. Пример: public abstract sample();

**Модификатор synchronized** – используются в Java для потоков.

Ключевое слово synchronized используется для указания того, что метод может быть доступен только одним потоком одновременно. В Java модификатор synchronized может быть применен с любым из четырех модификаторов уровня доступа.

Переменная экземпляра отмеченная как **transient** указывает виртуальной машине Java (JVM), чтобы пропустить определенную переменную при сериализации объекта, содержащего её. Этот модификатор включён в оператор, что создает переменную, предшествующего класса или типа данных переменной.

Модификатор volatile – используются в Java для потоков.

В Java модификатор volatile используется, чтобы позволить знать JVM, что поток доступа к переменной всегда должен объединять свою собственную копию переменной с главной копией в памяти. Доступ к переменной volatile синхронизирует все кэшированные скопированные переменные в оперативной памяти. Volatile может быть применен только к переменным экземпляра, которые имеют тип объект или private. Ссылка на объект volatile может быть null.

**this** Ключевое слово является ссылкой на текущий объект.

Другой способ подумать об этом заключается в том, что this ключевое слово похоже на личное местоимение, которое вы используете для обозначения себя.

Думайте о this как о простом способе для типа сказать "my".

# Чем отличаются static-метод класса от обычного метода класса. Можно ли вызвать static-метод внутри обычного метода?

Static-метод класса отличается от обычного метода тем, что обычный метод класса работает от объекта класса, а static-метод от всего класса.

Static-метод может быть вызван без создания экземпляра класса.

Обычный метод класса, напротив, принадлежит конкретному экземпляру класса и имеет доступ к его полям и методам.

Да, можно вызвать static-метод внутри обычного метода класса, но это делается через имя класса, а не через экземпляр класса.

# Объявление и использование методов, объявленных с модификатором public static. Как вызвать обычный метод класса внутри static-метода?

Методы, объявленные с модификатором public static, являются методами класса и могут быть вызваны без создания экземпляра данного класса.

Если внутри static-метода нужно вызвать обычный метод класса, можно сделать это, создав экземпляр класса и вызвав метод через этот экземпляр

Например:

public class MyClass {

public void regularMethod() {

System.out.println("This is a regular method");

}

public static void staticMethod() {

MyClass myObject = new MyClass();

myObject.regularMethod();

}

}

# Синтаксис объявления методов в Джава, тип возвращаемого значения, формальные параметры и аргументы. Методы с пустым списком параметров

Методы могут возвращать или не возвращать значения, могут вызываться с указанием параметров или без.

Тип возвращаемых данных указывают при объявлении метода -- перед его именем (void, int, string и тд.). В заголовке метода сначала идут модификаторы, определяющие, на каких условиях он доступен для вызова. Вернёмся к заголовку: после модификаторов - возвращаемый тип, затем идет имя метода, в скобках - параметры. B Java параметры и аргументы отличаются по своему назначению и использованию, хотя синтаксически они выглядят одинаково.

**Формальные параметры** - это переменные, объявленные в объявлении метода или конструктора, которые указывают на тип данных, которые метод ожидает получить.

Фактические параметры, или **аргументы**, - это конкретные значения, которые передаются в метод при его вызове.

Методы с пустым списком параметров просто не принимают параметров на вход, совершая все действия, не используя входящих параметров.

# Стандартные методы класса сеттеры и геттеры, синтаксис и их назначение?

Методы геттеры возвращают текущее значение переменной (получает значение). Методы сеттеры позволяют получить доступ к закрытому полю класса извне и изменить его значение (устанавливает значение).

String getName(){ return this.name; }

void getName(string name) { this.name = name; }

# Может ли быть поле данных класса объявлено как с модификатором static и final одновременно и что это означает?

Да, поле данных класса может быть объявлена как static и final одновременно. Это означает, что это поле является статическим (принадлежит классу, а не экземпляру) и неизменным (не может быть переопределено или изменено после инициализации).

Когда поле данных класса объявлена как static и final, оно называется "константой класса". Константы класса обычно используются для хранения значений, которые должны быть доступны для всех экземпляров класса, но не могут быть изменены. Например, часто константы класса используются для хранения математических констант (например, число π), кодов ошибок или других значений, которые не должны меняться в течение выполнения программы.

Тема 3. Реализация наследования в программах на Джаве

# Наследование, виды наследования и его реализация в Java и C++

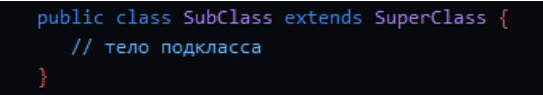
Наследование — механизм, который позволяет описать новый класс на основе существующего (родительского). При этом свойства и функциональность родительского класса заимствуются новым классом.

Существуют два основных вида наследования - одиночное и множественное. Одиночное наследование означает, что подкласс наследует от одного базового класса, а множественное наследование означает, что подкласс наследует от нескольких базовых классов (реализуется только через интерфейсы).

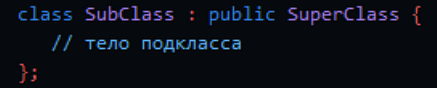
Реализация:

1. Наследование в одиночном (или простом) виде

Java:



C++



1. Множественное наследование (только в C++): В C++ разрешено множественное наследование, когда один подкласс наследует свойства и методы сразу от нескольких суперклассов

# Расширение классов. Порядок создания экземпляра дочернего класса.

В Java расширение классов осуществляется с помощью ключевого слова extends. При этом дочерний класс (или подкласс) наследует поля и методы родительского класса (или суперкласса) и может добавлять свои собственные поля и методы.

Порядок создания экземпляра дочернего класса включает в себя следующие шаги:

1. Выбор конструктора: При создании экземпляра дочернего класса сначала вызывается конструктор этого класса. Если в дочернем классе не определен явно конструктор, то будет вызван конструктор по умолчанию.

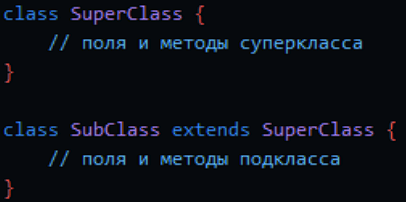
2. Вызов конструктора суперкласса: Первым действием в конструкторе дочернего класса должен быть вызов конструктора суперкласса с помощью ключевого слова super. Это позволяет инициализировать части объекта, которые наследуются от суперкласса.

3. Выполнение кода конструктора дочернего класса: После вызова конструктора суперкласса выполняется код конструктора дочернего класса, который может инициализировать собственные поля и выполнять другие необходимые действия.

# Наследование в Джава. Вид наследования и синтаксис Ключевое слово extends

Наследование в Java позволяет одному классу (подклассу) наследовать свойства и методы другого класса (родительского класса). Подкласс может использовать все свойства и методы родительского класса, а также расширять их или переопределять.

Реализация:



# Что означает перегрузка метода в Java (overload) и переопределение метода в Java (override)? В чем разница?

Перегрузка метода (overload) – это создание нескольких методов с одинаковыми именами, но разными параметрами (сигнатурами). Переопределение метода (override) – это создание новой реализации метода из родительского класса в дочернем классе.

Перегрузка методов позволяет упростить код и увеличить его читаемость, так как одно имя метода может выполнять разные функции, в зависимости от переданных аргументов.

Переопределение метода говорит о том, что родительский метод переопределен в наследнике. Компилятор при наличии такой аннотации проверяет, не нарушены ли правила переопределения.

Проще понять на примере. Допустим, вы создаете класс Animal с методом voice(). Он нужен для того, чтобы животное могло подать голос:

public class Animal {

   public void voice() {

       System.out.println("Говори!");

   }

}

И сразу возникает проблема — все животные издают разные звуки. Можно создать для каждого отдельный метод. Например, у кошки это будет voiceCat(), а у собаки — voiceDog(). Но представьте, сколько строк кода придётся написать, чтобы дать возможность всем животным подать голос?

Здесь на помощь и приходит механизм переопределения в Java. Он позволяет заменить реализацию в классе-наследнике. Посмотрим на примере кошки и собаки:

public class Cat extends Animal {

   @Override

   public void voice() {

       System.out.println("Мяу!");

   }

}

public class Dog extends Animal {

   @Override

   public void voice() {

       System.out.println("Гав!");

   }

}

В выводе отобразится сначала «Мяу», а затем — «Гав». Чтобы добиться такого результата, нужно:

# Абстрактные классы в Джава и абстрактные методы класса. Вложенные и анонимные классы.

Абстракция - это принцип ООП, согласно которому при проектировании классов и создании объектов необходимо выделять только главные свойства сущности, и отбрасывать второстепенные.

Абстрактным называется такой класс, экземпляр которого нельзя создать сам по себе - он служит основой для других классов.

В Java абстрактный класс объявляется с использованием ключевого слова abstract. Абстрактный класс может содержать как обычные методы, так и абстрактные методы. Абстрактный метод не имеет тела и должен быть переопределен в подклассе. Объект абстрактного класса не может быть создан, но он может быть использован в качестве типа ссылки.

abstract class Shape {

abstract void draw();

void display() {

System.out.println("Displaying shape");

}

}

Вложенные классы в Java - это классы, определенные внутри другого класса. Они могут быть статическими или нестатическими. Нестатический вложенный класс имеет доступ к членам внешнего класса, в то время как статический вложенный класс может быть создан без создания экземпляра внешнего класса.

class Outer {

private int outerVar;

class Inner {

void display() {

System.out.println("Value of outerVar: " + outerVar);

}

}

}

Анонимные классы – это классы без имени, которые создаются на месте их использования. Они обычно используются для реализации интерфейсов или абстрактных классов.

interface Greeting {

void greet();

}

public class Main {

public static void main(String[] args) {

Greeting greeting = new Greeting() {

@Override

public void greet() {

System.out.println("Hello!");

}

};

greeting.greet();

}

}

Ключевые особенности анонимных классов:

1. Отсутствие имени: Анонимные классы не имеют названия. Они создаются и использованы на месте, где они нужны, что делает их удобными для обработки небольших кусков кода без необходимости создания отдельного класса.

2. Использование для интерфейсов и подклассов: Анонимные классы обычно используются для реализации интерфейсов или создания подклассов существующих классов без явного создания нового класса.

# Виды наследования в Джава, использование интерфейсов для реализации наследования

* Одиночное наследование.
* Множественное наследование (используются интерфейсы)

Java не поддерживает множественное наследование классов, она поддерживает множественное наследование через интерфейсы. Класс может реализовать несколько интерфейсов, что позволяет ему наследовать поведение от нескольких источников.

// Определение интерфейса

interface Sound {

void makeSound();

}

// Определение другого интерфейса

interface Move {

void move();

}

// Определение класса, который реализует оба интерфейса

class Dog implements Sound, Move {

public void makeSound() {

System.out.println("Собака лает");

}

public void move() {

System.out.println("Собака бежит");

}

}

# Что наследуется при реализации наследования в Джава (какие компоненты класса), а что нет?

При реализации наследования в Java наследуются следующие компоненты класса:

* Поля (переменные экземпляра).
* Методы.
* Вложенные классы.

При этом не наследуются:

* Конструкторы.
* Статические переменные и методы.
* Приватные переменные и методы (они не доступны в подклассе).
* Финальные методы (они не могут быть переопределены).

Также, при наследовании подкласс не может наследовать от более чем одного класса, так как Java не поддерживает множественное наследование классов.

# К каким методам и полям базового класса производный класс имеет доступ (даже если базовый класс находится в другом пакете), а каким нет? Область видимости полей и данных из производного класса

Производный класс (подкласс) имеет доступ к следующим компонентам базового класса (родительского класса), даже если базовый класс находится в другом пакете:

* Публичные методы и поля.
* Защищенные методы и поля.
* Пакетные методы и поля, если производный класс находится в том же пакете, что и базовый класс.

Производный класс не имеет доступа к приватным методам и полям базового класса, даже если они находятся в том же пакете.

Область видимости полей и методов из производного класса ограничена их модификаторами доступа. Если поле или метод в базовом классе объявлено с модификатором private, то оно недоступно из производного класса. Если оно объявлено с модификатором protected, public или default (package-private), то оно будет доступно из производного класса в зависимости от области видимости производного класса.

# Класс Object, его методы, их назначение. Иерархия классов в Java.

Класс Object является корневым классом для всех классов в Java. Он определяет общие методы, которые наследуются всеми объектами в Java.

Некоторые из методов класса Object включают:

1. equals(Object obj) - используется для сравнения объектов на равенство.

2. hashCode() - возвращает хэш-код объекта, который используется в хэш-таблицах.

3. toString() - возвращает строковое представление объекта.

4. getClass() - возвращает класс объекта в виде объекта типа Class.

5. clone() - создает и возвращает копию объекта.

6. finalize() - вызывается перед удалением объектом сборщиком мусора.

Иерархия классов в Java строится вокруг класса Object. Все классы являются потомками класса Object, либо непосредственно (если они не наследуют от другого класса), либо косвенно через цепочку наследования.

# Наследование. Использование ключевых слов this и super. Пример использования в языках Си++ и Java

Наследование - это механизм в объектно-ориентированных языках программирования, который позволяет создавать новые классы на основе уже существующих классов. При использовании наследования, новый класс, называемый подклассом или производным классом, наследует свойства и методы от другого класса, называемого суперклассом или базовым классом.

Ключевое слово this используется для ссылки на текущий объект, а ключевое слово super используется для вызова конструктора или метода родительского класса.

Пример использования в языке C++:

class Animal {

public:

void eat() {

cout << "Animal is eating" << endl;

}

};

class Dog : public Animal {

public:

void bark() {

cout << "Dog is barking" << endl;

this->eat(); // обращение к методу eat текущего объекта

}

};

Пример использования в языке Java:

class Animal {

public void eat() {

System.out.println("Animal is eating");

}

}

class Dog extends Animal {

public void bark() {

System.out.println("Dog is barking");

this.eat(); // обращение к методу eat текущего объекта

}

}

# Паттерны проектирования программ. Паттерн Фабрика.

Паттерн Фабрика (Factory) – это порождающий паттерн проектирования, который предоставляет интерфейс для создания экземпляра объекта в суперклассе, но оставляет выбор типа создаваемого объекта подклассам.

Абстрактная фабрика – это порождающий паттерн проектирования, который позволяет создавать семейства связанных объектов, не привязываясь к конкретным классам создаваемых объектов.

Абстрактная фабрика используется:

• Когда система не должна зависеть от способа создания и компоновки новых объектов

• Когда создаваемые объекты должны использоваться вместе и являются взаимосвязанными

# Паттерны проектирования программ. Паттерн Фабричный метод.

Паттерн Фабричный метод (Factory Method) – это порождающий паттерн проектирования, который определяет интерфейс для создания объекта, но оставляет выбор типа создаваемого объекта подклассам.

# Расширение классов в Джава. Переопределение методов. Сокрытие полей данных.

Расширение классов, переопределение методов и сокрытие полей данных - это ключевые концепции объектно-ориентированного программирования (ООП) в Java, позволяющие создавать гибкие и удобные в использовании классы.

**Расширение классов**: В Java, ключевое слово extends используется для расширения (или наследования) классов. Один класс может наследовать свойства и методы другого класса.

java

Copy code

class Parent {

// Родительский класс

}

class Child extends Parent {

// Дочерний класс, наследует от Parent

}

Переопределение методов (Method Overriding): Дочерний класс имеет возможность переопределить методы своего родительского класса, если эти методы не являются final. Это позволяет дочернему классу предоставить свою реализацию метода с тем же именем и сигнатурой.

java

Copy code

class Parent {

void display() {

System.out.println("Метод display() в родительском классе");

}

}

class Child extends Parent {

@Override

void display() {

System.out.println("Метод display() в дочернем классе");

}

}

Сокрытие полей данных (Data Hiding):

Используя модификаторы доступа (public, private, protected), можно контролировать доступ к полям данных в классе. Ключевое слово private используется для скрытия полей от прямого доступа извне класса.

java

Copy code

class MyClass {

private int myPrivateField; // Скрытое приватное поле

public int getMyPrivateField() {

return myPrivateField;

}

public void setMyPrivateField(int value) {

myPrivateField = value;

}

}

Таким образом, расширение классов позволяет создавать иерархию классов, переопределение методов дает возможность дочернему классу предоставить свою реализацию, а сокрытие полей данных обеспечивает контроль доступа к ним, предоставляя методы доступа к данным (get и set методы). Эти концепции позволяют создавать более гибкие, модульные и удобные для использования классы в Java.

# Паттерны проектирования программ. Паттерн Observer и модель MVC

Паттерн Observer (наблюдатель) – это поведенческий паттерн проектирования, который позволяет создать механизм подписки и уведомления об изменениях в объектах. Модель MVC (Model-View-Controller) – это архитектурный шаблон проектирования, который разделяет приложение на три компонента: модель (хранение данных), представление (отображение данных) и контроллер (управление данными и представлением).

Model. Первая компонента/модуль — так называемая модель. Она содержит всю бизнес-логику приложения.

View. Вторая часть системы — вид. Данный модуль отвечает за отображение данных пользователю. Все, что видит пользователь, генерируется видом.

Controller. Третьим звеном данной цепи является контроллер. В нем хранится код, который отвечает за обработку действий пользователя (любое действие пользователя в системе обрабатывается в контроллере).

Тема 4. Полиморфизм в Джава. Работа со строками. Интерфейсы.

# Интерфейсы. Общий синтаксис и расширение. Пустые интерфейсы. Реализация и применение. Сравнение с абстрактными классами.

Интерфейсы в Java представляют собой абстрактные типы данных, которые содержат только сигнатуры методов, но не их реализацию. Интерфейсы могут содержать как абстрактные методы, так и константы. Пустой интерфейс – это интерфейс без объявленных методов. Основное различие между интерфейсом и абстрактным классом заключается в том, что интерфейс не может иметь состояние, тогда как абстрактный класс может иметь состояние с переменными экземпляра.

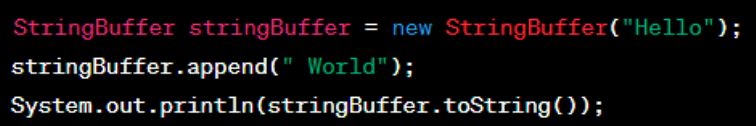
# Обработка строк в Java. Класс StringBuffer. Класс StringBuilder

В Java классы StringBuffer и StringBuilder предназначены для работы с изменяемыми строками, в отличие от класса String, который является неизменяемым. Оба класса предоставляют методы для изменения содержимого строки без создания новых объектов. Однако есть некоторые различия между ними. **StringBuffer**:

• StringBuffer является потокобезопасным (thread-safe) классом, что означает, что его методы синхронизированы и могут быть безопасно использованы в многопоточных приложениях.

• Из-за синхронизации StringBuffer может быть менее эффективным в однопоточных приложениях по сравнению с StringBuilder.

• Пример использования StringBuffer:

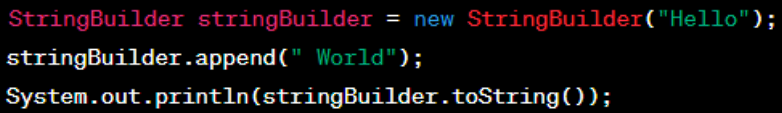


**StringBuilder**:

• StringBuilder представляет собой несинхронизированный аналог StringBuffer.

• По сравнению с StringBuffer, StringBuilder может обеспечивать более высокую производительность в однопоточных приложениях.

• Пример использования StringBuilder:



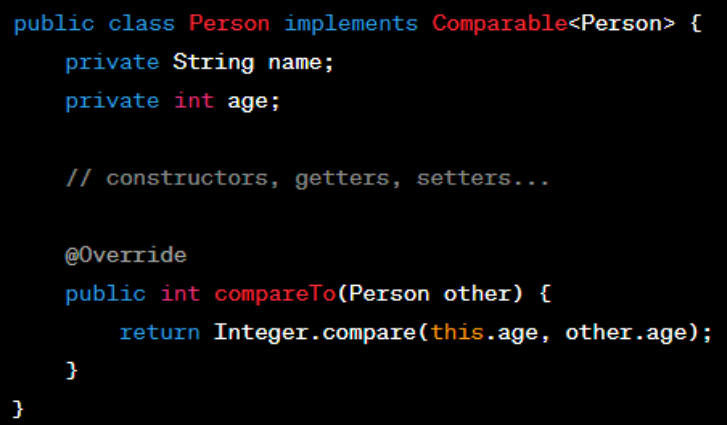
# Работа со строками в Java, строковый кэш. Операция конкатенации строк

# Интерфейс Comparable и Comparator. Использование интерфейсных ссылок для написания обобщенных алгоритмов

Comparable и Comparator в Java предоставляют способы сравнения объектов для упорядочивания. Эти интерфейсы особенно полезны при сортировке коллекций объектов.

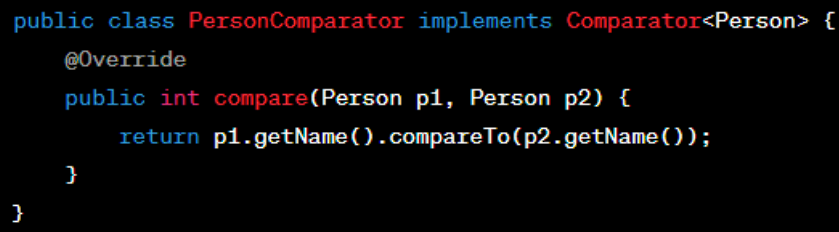
1. **Comparable** интерфейс:

• Реализуя интерфейс Comparable, объекты становятся сравнимыми с помощью их собственного метода compareTo.



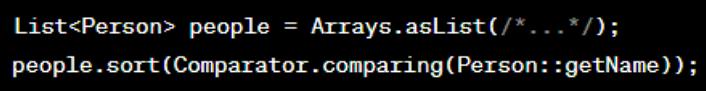
2. **Comparator** интерфейс:

• Comparator предоставляет более гибкий подход, позволяя определить сравнение объектов вне их класса.



3. Использование интерфейсных ссылок для написания обобщенных алгоритмов:

С использованием интерфейсных ссылок и лямбда-выражений можно создавать обобщенные алгоритмы для сортировки и сравнения.



Здесь мы создаём список людей (List people), а затем сортируем его методом sort. В метод мы передаем компаратор, который сортирует людей по методу getName класса Person.

# Понятие сортировки массивов. Сортировка пузырьком. Сортировка вставками. Использование полиморфизма (ООП) для программирования алгоритмов сортировок в массивах и коллекциях

Сортировка массивов – это процесс упорядочивания элементов массива по определенному критерию.

Сортировка пузырьком - это простой алгоритм сортировки, который многократно проходит через массив, сравнивая пары соседних элементов и меняя их местами, если они находятся в неправильном порядке. Этот процесс повторяется до тех пор, пока массив не будет отсортирован.

Пример:

public class BubbleSort {

public static void bubbleSort(int[] arr) {

int n = arr.length;

for (int i = 0; i < n-1; i++) {

for (int j = 0; j < n-i-1; j++) {

if (arr[j] > arr[j+1]) {

// меняем местами arr[j] и arr[j+1]

int temp = arr[j];

arr[j] = arr[j+1];

arr[j+1] = temp;

}

}

}

}

public static void main(String[] args) {

int[] arr = {64, 34, 25, 12, 22, 11, 90};

bubbleSort(arr);

System.out.println("Отсортированный массив:");

for (int i : arr) {

System.out.print(i + " ");

}

}

}

Сортировка вставками – это алгоритм сортировки, который проходит по массиву и вставляет каждый элемент на правильное место в отсортированной части массива. Этот процесс повторяется до тех пор, пока весь массив не будет отсортирован.

Пример:

public class InsertionSort {

public static void insertionSort(int[] arr) {

int n = arr.length;

for (int i = 1; i < n; i++) {

int key = arr[i];

int j = i - 1;

while (j >= 0 && arr[j] > key) {

arr[j + 1] = arr[j];

j = j - 1;

}

arr[j + 1] = key;

}

}

public static void main(String[] args) {

int[] arr = {64, 34, 25, 12, 22, 11, 90};

insertionSort(arr);

System.out.println("Отсортированный массив:");

for (int i : arr) {

System.out.print(i + " ");

}

}

}

Использование полиморфизма в ООП для программирования алгоритмов сортировок в массивах и коллекциях позволяет создать общий интерфейс или абстрактный класс для всех алгоритмов сортировки, после чего реализовать конкретные алгоритмы сортировки как подклассы этого интерфейса или класса. Это позволяет легко заменять один алгоритм сортировки другим без изменения основного кода.

Пример:

interface SortAlgorithm {

void sort(int[] arr);

}

class BubbleSort implements SortAlgorithm {

@Override

public void sort(int[] arr) {

// Реализация сортировки пузырьком

}

}

class InsertionSort implements SortAlgorithm {

@Override

public void sort(int[] arr) {

// Реализация сортировки вставками

}

}

public class Main {

public static void main(String[] args) {

SortAlgorithm algorithm = new BubbleSort(); // Использование полиморфизма

int[] arr = {64, 34, 25, 12, 22, 11, 90};

algorithm.sort(arr);

System.out.println("Отсортированный массив:");

for (int i : arr) {

System.out.print(i + " ");

}

}

}

# Понятие поиска в массивах. Последовательный поиск. Сортировка методом прямого выбора. Использование полиморфизма (ООП) для программирования алгоритмов поиска в массивах и коллекциях

**Последовательный поиск** – это прямой способ поиска элемента в коллекции. Этот тип поиска использует цикл, чтобы просмотреть каждый элемент по одному и посмотреть, совпадает ли он с элементом, который мы ищем. Механизм поиска движется в определенной последовательности, отсюда и название последовательного поиска. В случае, если данные отсортированы, мы можем сократить количество запросов. Мы можем остановить поиск, когда превысим целевой элемент. Это возможно только в том случае, если целевой элемент отсутствует

**Сортировка прямым выбором**

Алгоритм сортировки прямым выбором в некотором смысле противоположен сортировке прямыми включениями. При прямом включении на каждом шаге рассматривается только один очередной элемент входной последовательности и все элементы готовой последовательности для нахождения места включения. При прямом выборе для поиска одного элемента с наименьшим ключом просматриваются все элементы входной последовательности и найденный элемент помещается как очередной элемент в конец готовой последовательности.

Полиморфизм – это способность программы идентично использовать объекты с одинаковым интерфейсом без информации о конкретном типе этого объекта.

# Объявление и инициализация переменных типа String. Операция конкатенации строк и ее использование

В Java переменные типа String объявляются как любые другие переменные, но у них есть специфический тип данных String. Вот несколько способов объявления и инициализации переменных типа String: Объявление и инициализация переменных типа String:

Используя литералы строк:

String myString = "Привет, мир!";

С помощью конструктора:

String anotherString = new String("Это другая строка");

Операция конкатенации строк:

Конкатенация строк означает объединение строк в одну строку. В Java для конкатенации строк используется оператор +. Этот оператор соединяет две строки вместе.

String firstName = "John";

String lastName = "Doe";

String fullName = firstName + " " + lastName; // Конкатенация строк

Также в Java есть метод concat, который позволяет конкатенировать строки:

String str1 = "Hello";

String str2 = "World";

String result = str1.concat(str2); // Использование метода concat для конкатенации

Использование конкатенации строк:

Конкатенация строк часто используется для формирования более сложных строковых значений, например, при создании сообщений для вывода или формировании запросов в базу данных.

int age = 25;

String message = "Мой возраст: " + age + " лет"; System.out.println(message); // Выведет: "Мой возраст: 25 лет"

Такие операции могут быть полезны при формировании строковых выражений для сообщений пользователю или при создании строк для работы с данными в приложениях.

# При создании объектов строк с помощью класса StringBuffer, например StringBuffer strBuffer = new StringBuffer(str) можно ли использовать операцию конкатенации строк или необходимо использовать методы класса 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.

• StringBuffer() — создаст пустой (не содержащий символов) объект

• StringBuffer(String str) — создаст объект на основе переменной str (содержащий все символы str в той же последовательности)

Конкатенация строк через StringBuffer в Java выполняется с помощью метода append.

strBuffer += "sd"; так нельзя

Вообще метод append в классе StringBuffer перегружен таким образом, что может принимать в себя практически любой тип данных.

Метод append возвращает объект, на котором был вызван (как и многие другие методы), что позволяет вызывать его “цепочкой”.

Для работы со строками у класса StringBuffer есть ряд методов.

Перечислим основные:

• delete(int start, int end) — удаляет подстроку символов начиная с позиции start, заканчивая end

• deleteCharAt(int index) — удаляет символ в позиции index

• insert(int offset, String str) — вставляет строку str в позицию offset. Метод insert также перегружен и может принимать различные аргументы

• replace(int start, int end, String str) — заменит все символы начиная с позиции start до позиции end на str

• reverse() — меняет порядок всех символов на противоположный

Преимущества:

1. Как уже сказано, StringBuffer — изменяемый класс, поэтому при работе с ним не возникает такого же количества мусора в памяти, как со String. Поэтому если над строками проводится много модификаций, лучше использовать StringBuffer.

2. StringBuffer — потокобезопасный класс. Его методы синхронизированы, а экземпляры могут быть использованы несколькими потоками одновременно.

Недостатки:

С одной стороны, потокобезопасность — преимущество класса, а другой — недостаток. Синхронизированные методы работают медленнее не синхронизированных.

# Объявление и инициализация массива строк. Организация просмотра элементов массива

Общая форма объявления и выделение памяти для одномерного массива строк:

String[] arrayName = new String[size];

Инициализация одномерного массива строк точно такая же как инициализация одномерного массива любого другого типа. String[] M = {

"Sunday",

"Monday",

"Tuesday",

"Wednesday",

"Thursday",

"Friday",

"Saturday"

};

String s;

s = M[2]; // s = "Tuesday"

s = M[4]; // s = "Thursday"

Просмотр элементов массива:

Чтобы просмотреть элементы массива строк, можно использовать цикл for или цикл foreach:

Цикл for:

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

System.out.println(M[i]);

}

Цикл foreach (цикл for-each):

for (String m: M) {

System.out.println(m);

}

# Понятие и объявление интерфейсов в Джава. Может ли один класс реализовывать несколько интерфейсов?

Интерфейс в языке программирования Java - это абстрактный тип, который используется для указания поведения, которое должны реализовывать классы. Да, в Java один класс может реализовывать несколько интерфейсов. Для этого класс объявляется с помощью ключевого слова "implements".

public class MyClass implements Interface1, Interface2 {

// Код класса

}

# Что входит в состав интерфейса (какие компоненты может содержать интерфейс)? Может ли интерфейс наследоваться от другого интерфейса?

Состав интерфейса может включать в себя следующие компоненты:

* Методы без реализации – это обязательный элемент интерфейса, описывающий методы, которые должны быть реализованы в классе, использующем интерфейс.
* Константы - интерфейс может содержать константы, которые используются в классах, реализующих интерфейс.
* Дефолтные методы - начиная с Java 8, интерфейсы могут содержать методы с реализацией по умолчанию.
* Статические методы - также начиная с Java 8, интерфейсы могут содержать статические методы.

Интерфейс может наследоваться от другого интерфейса. В этом случае дочерний интерфейс будет содержать методы и константы родительского интерфейса, а класс, реализующий дочерний интерфейс, должен реализовать как методы дочернего интерфейса, так и методы родительского интерфейса.

# Интерфейсные ссылки и их использование в Джава

В Java интерфейсные ссылки представляют ссылки на объекты, которые могут быть полезны для реализации полиморфизма. Когда объект класса реализует определенный интерфейс, ссылка на этот объект также может быть типизирована, как интерфейс, что позволяет использовать объект, как если бы он был этим интерфейсом.

# Интерфейс Camparable, назначение, его методы и использование в Джава

Comparable — это интерфейс, входящий в пакет java.lang и используемый для сортировки классов на основе их естественного порядка. Интерфейс Comparable должен быть реализован в классе, который будет использоваться для сортировки. Этот класс можно сортировать на основе отдельных атрибутов, таких как идентификатор, имя, отдел и так далее.

Класс, реализующий интерфейс Comparable, сравнивает себя с другими объектами. Реализованный класс предлагает пользовательскую реализацию int CompareTo(T var1) для пользовательской сортировки.

Метод int CompareTo(T var1) должен быть переопределен таким образом, чтобы:

* Он должен возвращать целое положительное значение Positive(+ve), если этот объект больше объекта сравнения.
* Он должен возвращать целое отрицательное значение Negative(-ve), если этот объект меньше объекта сравнения.
* Он должен вернуть Zero(0), если этот и сравниваемый объект равны.

# Какое значение возвращает вызов метода object1.compareTo(object2), который сравнивает 2 объекта obj1 и obj2 в зависимости от объектов?

Метод object1.compareTo(object2) возвращает:

- отрицательное значение, если object1 меньше object2

- нуль, если object1 равен object2

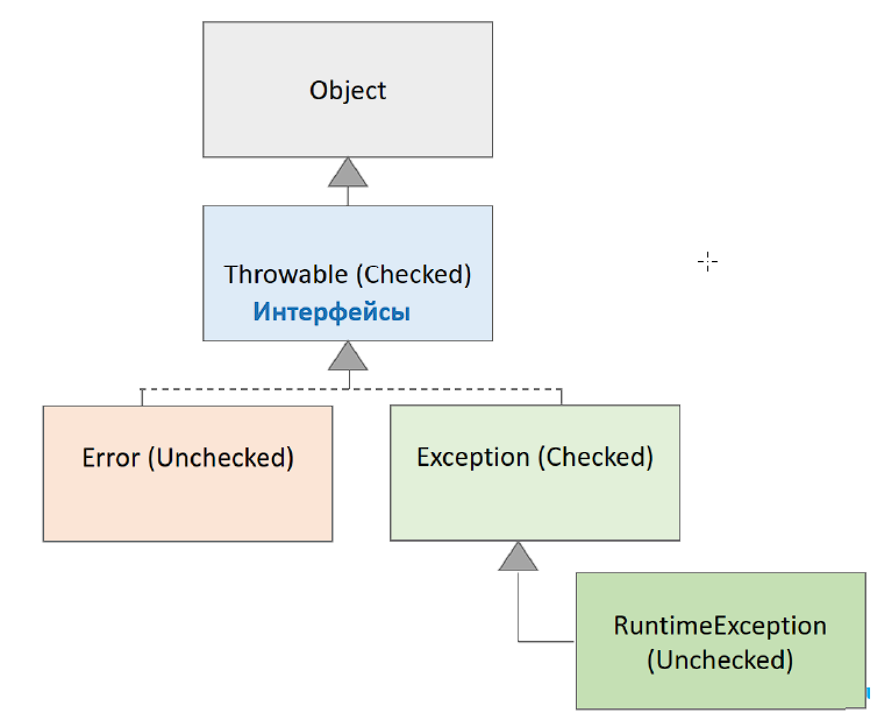
- положительное значение, если object1 больше object2.

Тема 5. Основные принципы и типы исключительных ситуаций.

# Понятие исключительной ситуации, причины возникновения, механизм обработки. Классификация исключений. Исключения, классификация и использование исключений. Генерация (порождение исключений).

Исключительная ситуация (исключение) – это специальный объект, описывающий исключительную ситуацию, которая возникла в некоторой части программного кода.

Причины возникновения исключительных ситуаций могут быть разными - ошибки в программе, некорректные данные, недоступность ресурсов и т.д.



Механизм обработки исключений предусматривает возможность перехвата и обработки исключения в коде программы. Для этого используются конструкции try-catch (попробовать-поймать), которые позволяют либо обработать исключение в текущем блоке кода, либо передать его выше по стеку вызовов.

Исключения можно классифицировать по типу ошибки, которая их вызвала (например, деление на ноль, отсутствие файла и т.д.), по уровню критичности (например, легкие, средние, критические) и по способу обработки (контролируемые исключения, которые обрабатываются программой, и неконтролируемые, которые не предполагают обработку в коде).

Исключения используются для обработки ошибок и некорректных ситуаций в программе. Они позволяют более гибко и безопасно управлять потенциальными проблемами, предотвращать аварийное завершение программы и предоставлять информацию об ошибке для последующей диагностики и устранения.

Генерация (порождение) исключений происходит в момент возникновения ошибки или некорректной ситуации в программе. Для этого используется оператор throw, который позволяет выбросить объект-исключение с указанием типа ошибки и дополнительной информации о ней.

# Служебное слово throw и его использование при определении методов. В каком случае программа должна использовать оператор throw?

Служебное слово "throw" в Java используется для генерации исключений. Оператор throw используется для явного выброса исключения внутри метода. Это позволяет программе обрабатывать ошибочные ситуации, например, когда метод получает некорректные данные или встречает другое непредвиденное состояние.

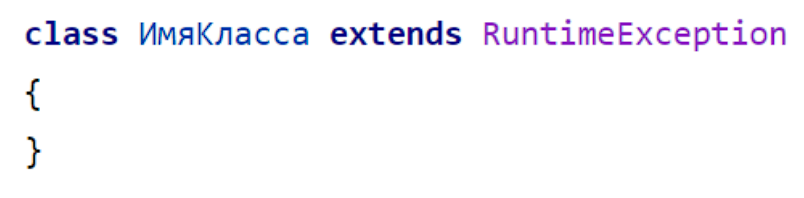
Программа должна использовать оператор throw в случае, когда происходит ошибка или непредвиденное состояние, которое нужно обработать. Например, если метод должен принимать только положительные значения, он может использовать оператор throw для генерации исключения, если ему передается отрицательное значение. Это позволяет программе контролировать поток выполнения и обрабатывать ошибки и исключения в соответствии с логикой программы.

# Создание собственных классов исключений

Существует возможность написать собственные исключения для обработки тех или иных ошибок, если нам недостаточно уже существующих библиотек. Для этого мы просто создаем класс-наследник от класса Исключения.

При создании собственных исключений следует учитывать два правила:

* Название нашего класса должно оканчиваться на «Exception»
* Класс должен содержать конструктор со строковой переменной, описывающей детали проблемы Исключения. В конструкторе вызывается супер-конструктор с передачей сообщения.

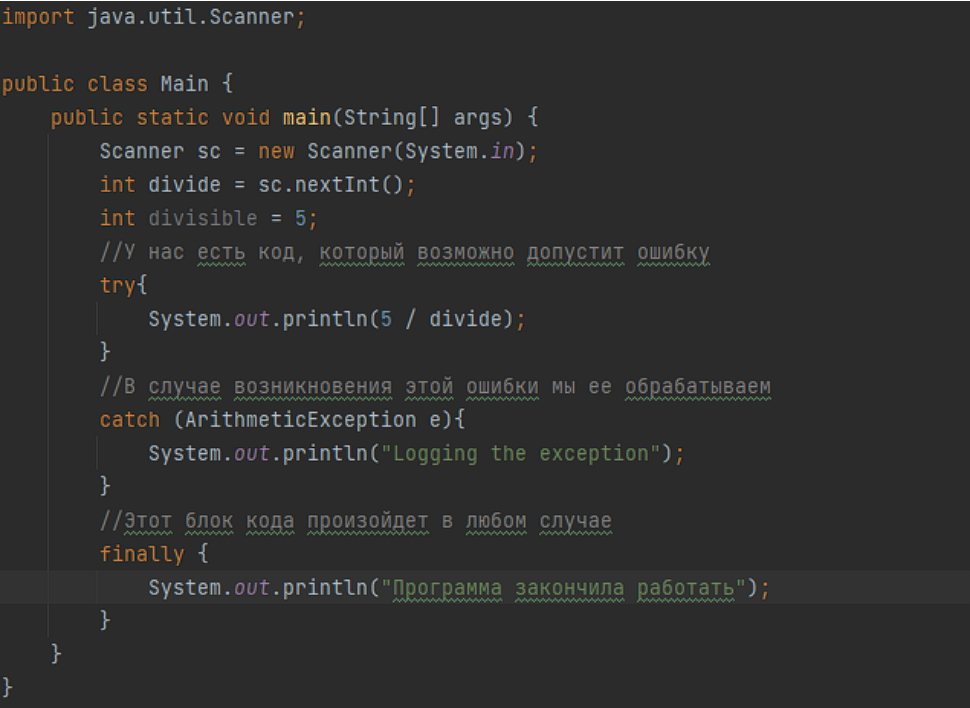


# Блок try/catch/finally, его предназначение и особенности

Блок try - содержит код, который может вызвать исключение. Код внутри блока try выполняется, и если возникает исключение, выполнение кода внутри блока try прерывается, и контроль передается обработчику исключений.

Блок catch - используется для перехвата и обработки возникших исключений. Если в блоке try произошло исключение, выполнение кода передается в блок catch, соответствующий типу исключения.

Блок finally - содержит код, который должен быть выполнен в любом случае, независимо от того, произошло исключение или нет. Этот блок часто используется для освобождения ресурсов, таких как закрытие файлов или соединений с базой данных, независимо от исключения.



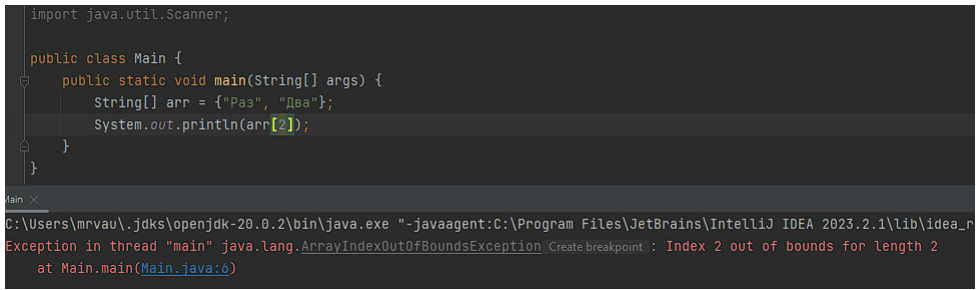
# В Java все исключения делятся на два основных типа. Что это за типы и какие виды ошибок ни обрабатывают?

В Java все исключения делятся на два основных типа: проверяемые исключения (checked exceptions) и непроверяемые исключения (unchecked exceptions).

* Проверяемые исключения (checked exceptions) - это исключения, которые должны быть обработаны в коде с помощью блока try-catch или объявлены в сигнатуре метода с помощью ключевого слова throws. Например, IOException, SQLException, FileNotFoundException и т.д. Обработка этих исключений обязательна, иначе код не будет компилироваться.
* Непроверяемые исключения (unchecked exceptions) - это исключения, которые не требуют обязательной обработки в коде. К ним относятся классы RuntimeException и его подклассы, такие как NullPointerException, ArrayIndexOutOfBoundsException, ArithmeticException и другие. Обработка этих исключений не является обязательной, но их можно обрабатывать, если это необходимо.
* Необрабатываемые ошибки (errors), такие как OutOfMemoryError, StackOverflowError и другие, обычно не обрабатываются в коде, так как они указывают на серьезные проблемы в работе приложения, которые обычно не могут быть корректно восстановлены.

# Код ниже вызовет ошибку: Exception <...> java.lang.ArrayIndexOutOfBoundsException: 4: Что она означает?

ArrayIndexOutOfBoundsException – это исключение, появляющееся во время выполнения. Оно возникает тогда, когда мы пытаемся обратиться к элементу массива по отрицательному или превышающему размер массива индексу.



Попытались обратиться к массиву по индексу, которого в массиве нет, все.

# Контролируемые исключения (checked) и неконтролируемые исключения (unchecked) и ошибки, которые они обрабатывают

Контролируемые исключения представляют собой ошибки, которые можно и нужно обрабатывать в программе, к этому типу относятся все потомки класса Exception (но не RuntimeException).

Обработка исключения может быть произведена с помощью операторов try…catch, либо передана внешней части программы. Например, метод может передавать возникшие в нём исключения выше по иерархии вызовов, сам его не обрабатывая.

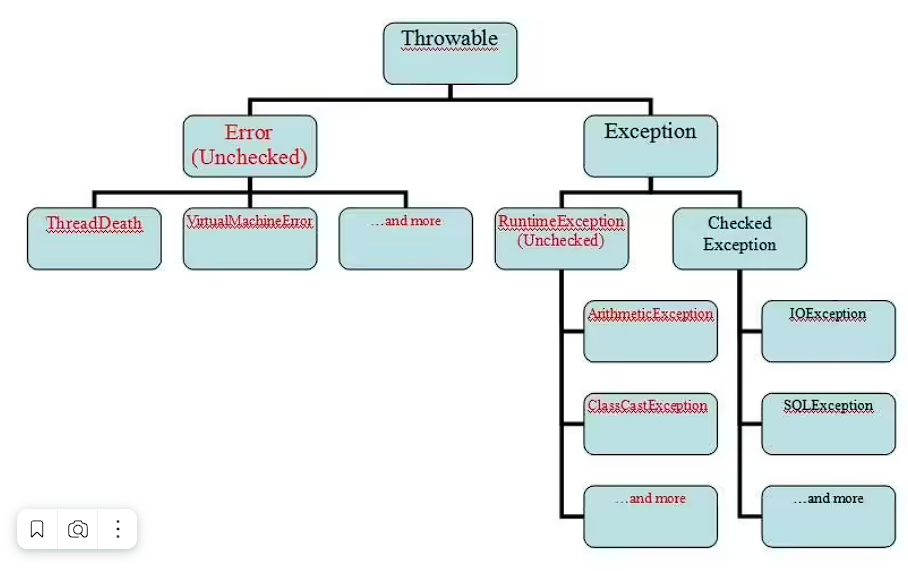
Неконтролируемые исключения не требуют обязательной обработки, однако, при желании, можно обрабатывать исключения класса RuntimeException.

Ошибки (Errors):

- Ошибки представляют собой серьезные проблемы, которые обычно выходят за рамки возможностей программиста для обработки. Они обычно возникают из-за проблем виртуальной машины Java или окружающей системы.

- Примеры ошибок: VirtualMachineError, ThreadDeath.

- Обработка ошибок обычно не выполняется, поскольку они указывают на серьезные проблемы, которые требуют вмешательства системного администратора или разработчика более высокого уровня.



# Как реализуется принципы ООП в Java при создании исключений? Порядок выполнения операторов при обработке блока блока try...catch

В Java принципы объектно-ориентированного программирования (ООП) применяются при создании и обработке исключений. Вот как это происходит:

1. Наследование:

class MyCustomException extends Exception {

// Конструкторы и другие методы

}

2. Полиморфизм:

try {

// Код, который может вызвать исключение

} catch (IOException | SQLException ex) {

// Обработка исключения

}

3. Инкапсуляция:

class CustomException extends Exception {

public CustomException(String message) {

super(message);

}

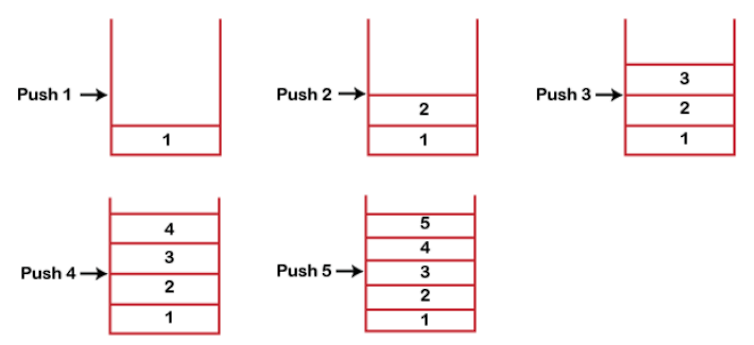
}

При использовании блока try...catch вначале выполняются все инструкции между операторами try и catch. Если в блоке try вдруг возникает исключение, то обычный порядок выполнения останавливается и переходит к инструкции catch.

Тема 6. Абстрактные типы данных Дженерики и использование контейнерных классов в Джава

# Абстрактный тип данных Stack (cтек) в Джава

Stack (стек) представляет собой структуру данных, работающую по принципу "последний вошел - первый вышел" (LIFO - Last In, First Out). Стек имеет один конец, куда мы можем добавлять элементы и извлекать их оттуда, в отличие от очереди, которая имеет два конца (спереди и сзади). Стек содержит только один указатель top (верхушка стека), указывающий на самый верхний элемент стека. Всякий раз, когда элемент добавляется в стек, он добавляется на вершину стека, и этот элемент может быть удален только из стека только сверху. Другими словами, стек можно определить как контейнер, в котором вставка и удаление элементов могут выполняться с одного конца, известного как вершина стека. Примеры стеков – стопка тарелок или книг.



**Общие операции, реализованные в стеке:**

• push(): добавляем элемент в стек. Если стек заполнен, возникает состояние переполнения.

• pop(): удаляем элемент из стека. Если стек пуст, это означает, что в стеке нет элементов, это состояние известно как состояние потери значимости.

• isEmpty(): определяет, пуст стек.

• isFull(): определяет, заполнен ли стек

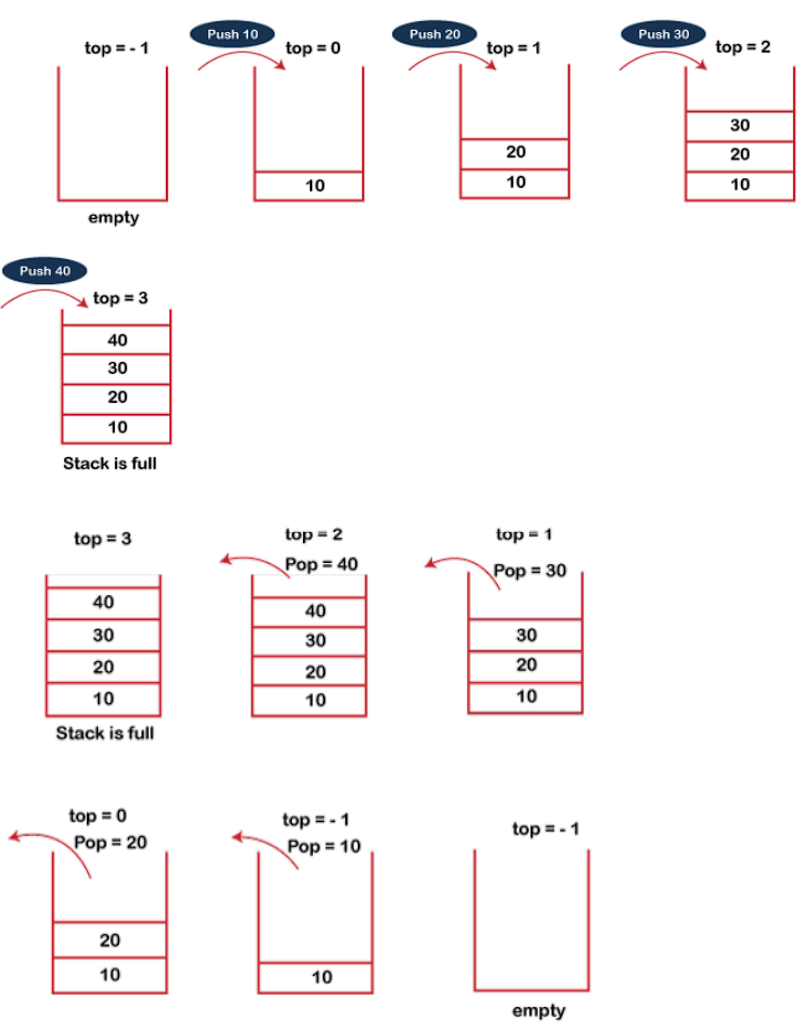
• peek(): возвращает элемент в заданной позиции.

• count(): возвращает общее количество элементов.

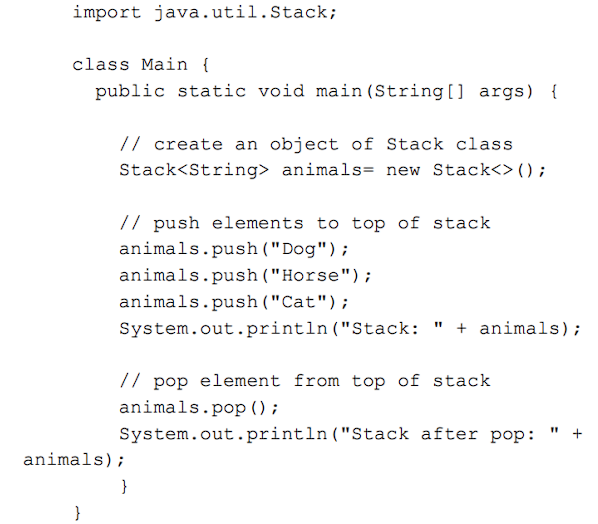
• change(): изменяет элемент в заданной позиции.

• display(): печатает все элементы, доступные в стеке.

**Операции pop и push**



**Реализация стека с помощью Stack**

****

# Абстрактный тип данных Queue (очередь) в Джава

Queue - упорядоченный список, который позволяет выполнять операции вставки на одном конце, называемом REAR , и операции удаления, которые выполняются на другом конце, называемом FRONT. Очередь работает по принципу «первый пришел — первый обслужен» ( FCFS, first come first served). Например: люди, стоящие в кассу магазина. Операция dequeue означает удаление элемента из начала очереди, а операция enqueue добавление элемента в конец очереди.



Виды очередей:

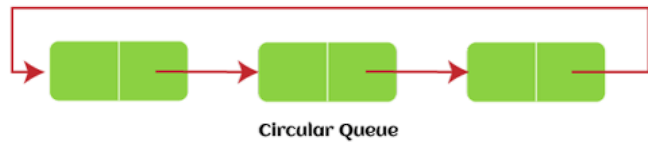
1. Простая очередь или линейная очередь

• Вставка элемента происходит с одного конца, а удаление — с другого.



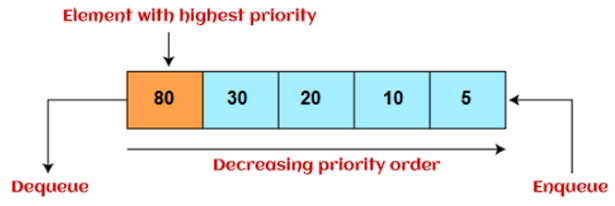
2. Циклическая очередь

• После последнего элемента очереди сразу идет первый или начальный элемент.



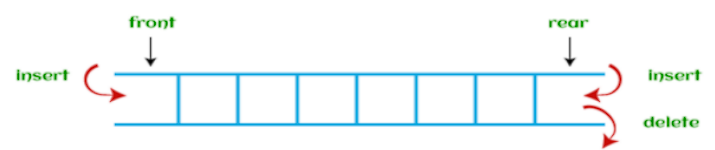
3. Очередь с приоритетами

• Это особый тип очереди, в которой элементы располагаются в зависимости от их приоритета. Это особый тип структуры данных очереди, в которой каждый элемент такой очереди имеет связанный с ним приоритет. Допустим, какие-то элементы встречаются с одинаковым приоритетом, тогда они будут располагаться по принципу FIFO.Вставка в такую очередь происходит на основе поступления элемента в соответствии с его приоритетом, а удаление в данной очереди происходит на основе приоритета.



4. Двусторонняя очередь или Дек (англ. Deque)

• В Deque или Double Ended Queue вставка и удаление могут выполняться с обоих концов очереди либо спереди, либо сзади. Это означает, что мы можем вставлять и удалять элементы как с переднего, так и с заднего конца очереди. Deque можно использовать и как стек, и как очередь, поскольку он позволяет выполнять операции вставки и удаления на обоих концах.



**Операции, выполняемые над очередью**

• Enqueue(): вставка элемента в конец очереди. Возвращает пустоту.

• Dequeue(): удаление из внешнего интерфейса очереди. Он также возвращает элемент, который был удален из внешнего интерфейса. Он возвращает целочисленное значение.

• Peek() Просмотр очереди: возвращает элемент, на который указывает передний указатель в очереди, но не удаляет его.

• isfFull() (Queue overflow):

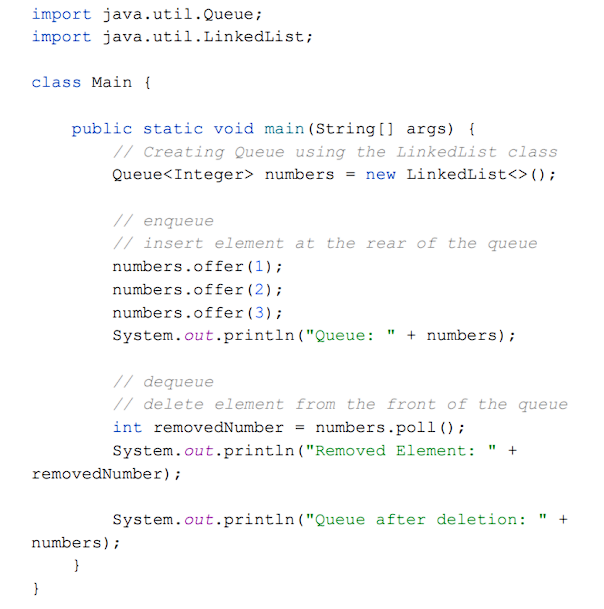
• isEmpty() (Queue underflow ):

**Способы реализации очереди**

• Реализация с использованием массива.

• Реализация с использованием связанного списка.

**Реализация очереди с помощью интерфейса Queue**

****

# Универсальные типы или обобщенные типы данных, для чего создаются?

Универсальный тип является одиночным элементом программирования, который используется для выполнения одинаковой функциональности для различных типов данных. При определении универсальных классов или процедур не нужно определять отдельную версию для каждого типа данных, для которых может потребоваться выполнение этой функциональности.

# Объявление обобщенного класса коллекции с параметризованным методом для обработки массива элементов коллекции на основе цикла for each (определение общего метода для отображения элементов массива)

# Что представляет из себя класс ArrayList и в каком случае используется

ArrayList - это класс в Java, который представляет собой динамический массив, который может изменять свой размер по мере необходимости. Он является частью стандартной библиотеки Java и находится в пакете java.util.

ArrayList часто используется в случаях, когда требуется хранить коллекцию объектов переменной длины. Он предоставляет удобные методы для добавления, удаления, доступа и изменения элементов в коллекции. ArrayList также автоматически увеличивает свой размер при необходимости, что делает его удобным для использования в ситуациях, когда заранее неизвестно, сколько элементов будет храниться.

# Обобщенное программирование. Понятие и использование дженериков в Java.

Обобщенное программирование (generic programming) – это методология разработки программ, которая позволяет создавать обобщенные (generic) структуры данных и алгоритмы, способные работать с различными типами данных. В Java обобщенное программирование реализуется с помощью дженериков (generics).

Дженерики в Java позволяют создавать классы, интерфейсы и методы, которые могут принимать параметры типа (type parameters), то есть параметры, которые представляют собой типы данных. Это позволяет создавать обобщенные структуры данных, такие как списки, множества, карты и т.д., которые могут работать с любым типом данных.

public class Box {

private T content;

public Box(T content) {

this.content = content;

}

public T getContent() {

return content;

}

}

// Создание объекта Box с типом String

Box <stringBox> = new Box<>("Hello World");

// Создание объекта Box с типом Integer

Box intBox = new Box<>(42);

// Получение содержимого объекта Box

String str = stringBox.getContent();

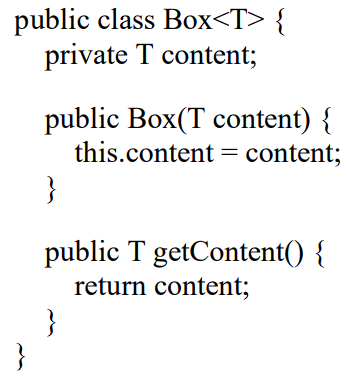
Integer i = intBox.getContent();

Преимущества использования дженериков в Java:

* + Безопасность типов: дженерики обеспечивают безопасность типов во время компиляции, что позволяет выявлять ошибки, связанные с типами данных на этапе компиляции.
  + Повышение читаемости кода: использование дженериков делает код более читаемым и понятным, так как не требуется явное приведение типов и повторное использование кода для различных типов данных.
  + Повышение производительности: дженерики позволяют избежать использования объектов типа Object и уменьшить необходимость в приведении типов, что может повысить производительность приложения.

# Параметризованные классы и методы. Их определение и использование

Параметризованные классы и методы - это классы и методы, которые могут работать с различными типами данных. В Java параметризованные классы и методы реализуются с помощью дженериков (generics). Параметризованный класс - это класс, который имеет один или несколько параметров типа (type parameter), которые заменяются на конкретный тип при создании экземпляра класса. Параметры типа указываются в угловых скобках после имени класса.



Параметризованный метод - это метод, который имеет один или несколько параметров типа, которые заменяются на конкретный тип при вызове метода. Параметры типа указываются в угловых скобках перед возвращаемым типом метода.

public static T getFirst(List list) {

if (list.isEmpty()) {

return null;

}

return list.get(0);

}

Использование параметризованных классов и методов позволяет создавать более гибкий и универсальный код, который может работать с различными типами данных. Это делает код более читаемым и удобным для работы.

# Стирание типов.

Стирание типов (type erasure) - это процесс, при котором параметры типа (type parameters) в дженериках (generics) заменяются на Object или их верхние ограничения (upper bounds) во время компиляции, а затем удаляются из скомпилированного байт-кода. Это делается для обеспечения обратной совместимости с кодом, написанным до появления дженериков в Java.



При вызове метода getContent() компилятор вставляет необходимые приведения типов, чтобы вернуть объект нужного типа.

Стирание типов может приводить к некоторым ограничениям при использовании дженериков. Например, нельзя создать массив параметризованного типа, потому что массивы в Java хранят информацию о типах элементов во время выполнения. Также нельзя создать экземпляр параметризованного типа во время выполнения, потому что информация о типе стирается во время компиляции.

# Понятие структуры данных список. Линейный список. Виды списков и их реализация на Java. Доступ к элементу структуры данных список. Использование списков. Трудоемкость операций со списками.

Список (List) - это структура данных, которая представляет собой упорядоченный набор элементов, каждый из которых имеет свой индекс. Список позволяет добавлять, удалять и изменять элементы внутри него.

Линейный список (Linked List) - это один из видов списков, который представляет собой последовательность элементов, каждый из которых содержит ссылку на следующий элемент. Первый элемент списка называется головой (head), а последний - хвостом (tail). Линейный список может быть однонаправленным (singly linked list), когда каждый элемент имеет ссылку только на следующий элемент, или двунаправленным (doubly linked list), когда каждый элемент имеет ссылки на следующий и предыдущий элементы.

В Java линейный список может быть реализован с помощью класса LinkedList, который представляет собой двунаправленный список. Класс ArrayList также может использоваться для реализации списка, но он представляет собой динамический массив, а не связанный список.

Доступ к элементу структуры данных списка осуществляется по его индексу. В линейном списке для доступа к элементу с определенным индексом необходимо пройти по всем элементам списка от головы до нужного элемента.

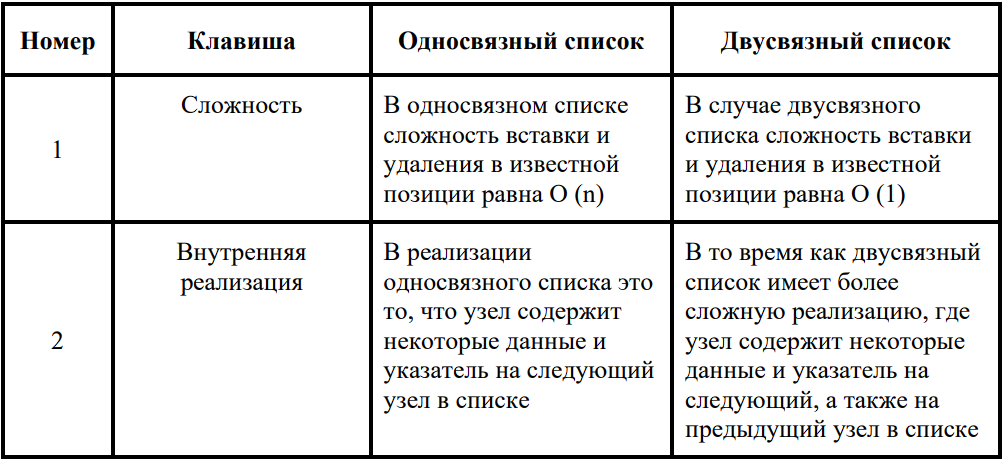
Списки широко используются в программировании для хранения и управления коллекциями данных, таких как списки задач, история действий пользователя, список контактов и т.д.

Трудоемкость операций со списками зависит от типа списка и используемой реализации. Например, добавление элемента в конец ArrayList имеет трудоемкость O(1), тогда как добавление элемента в конец LinkedList имеет трудоемкость O(n), потому что необходимо пройти по всем элементам списка до его конца. Удаление элемента из середины LinkedList также имеет трудоемкость O(n), потому что необходимо перестроить связи между элементами. При выборе реализации списка необходимо учитывать требования к производительности и особенности работы с данными.

# Односвязный и двусвязный список. Способы реализации на языке Джава

Как Односвязный список, так и двусвязный список являются реализацией связанного списка, в котором каждый элемент односвязного списка содержит некоторые данные и ссылку на следующий элемент, что позволяет сохранить структуру. С другой стороны, каждый узел в двусвязном списке также содержит ссылку на предыдущий узел.

Ниже приведены важные различия между односвязным списком и двусвязным списком.





Тема 7. Java Core. Дженерики (продолжение) и использование контейнерных классов Java Framework Collection

# Возможности Java Framework Collection. Контейнер ArrayList и его основные методы.

Класс java.util.ArrayList в коллекциях Java используется для хранения изменяемого размера массива элементов в списке. Это используется для хранения элементов и предоставляет эффективные методы для доступа, изменения и повторения элементов в списке.

Класс java.util.ArrayList обеспечивает быстрый произвольный доступ к своим элементам, но медленную вставку и удаление в произвольных позициях.

ArrayList - это динамический массив, который позволяет добавлять, удалять, изменять и получать элементы. Его основные методы включают:

1. add(E element) - добавляет элемент в конец списка.

2. add(int index, E element) - добавляет элемент по указанному индексу, сдвигая последующие элементы, если необходимо.

3. remove(int index) - удаляет элемент по указанному индексу.

4. set(int index, E element) - заменяет элемент по указанному индексу на новый элемент.

5. get(int index) - возвращает элемент по указанному индексу.

6. size() - возвращает размер списка.

ArrayList также автоматически расширяет свой размер по мере необходимости, что делает его удобной структурой данных для хранения и обработки коллекций объектов.

# Возможности Java Framework Collection. Контейнер LinkedList и его основные методы.

Класс java.util.LinkedList в Java Collection Framework наследует класс AbstractList и реализует интерфейсы List и Deque (двусвязный список). Он предоставляет эффективные методы для добавления, удаления и доступа к элементам в начале и конце списка.

Также этот Класс является реализацией интерфейса List, в которой для хранения элементов используется двусвязный список (doubly-linked list). Он обеспечивает быструю вставку и удаление в произвольных позициях, но медленный произвольный доступ к своим элементам.

Основные методы LinkedList включают:

1. add(E element) - добавляет элемент в конец списка.

2. add(int index, E element) - добавляет элемент по указанному индексу.

3. remove(int index) - удаляет элемент по указанному индексу.

4. get(int index) - возвращает элемент по указанному индексу.

5. set(int index, E element) - заменяет элемент по указанному индексу на новый элемент.

6. addFirst(E element) - добавляет элемент в начало списка.

7. addLast(E element) - добавляет элемент в конец списка.

8. removeFirst() - удаляет первый элемент списка.

9. removeLast() - удаляет последний элемент списка.

10. getFirst() - возвращает первый элемент списка.

11. getLast() - возвращает последний элемент списка.

LinkedList также предоставляет реализацию интерфейса Queue, что позволяет использовать его в качестве структуры данных для FIFO (первым вошел, первым вышел) очереди.

# Возможности Java Framework Collection. Интерфейс Map и его основные методы.

Map: коллекция пар ключ-значение (key-value), где каждый ключ уникален.

Основные методы интерфейса Map включают следующие:

1. void clear(): удаляет все пары "ключ-значение" из карты.

2. boolean containsKey(Object key): возвращает true, если карта содержит указанный ключ.

3. boolean containsValue(Object value): возвращает true, если карта содержит указанное значение.

4. Set<Map.Entry<K, V>> entrySet(): возвращает множество объектов Map.Entry, представляющих все пары "ключ-значение" в карте.

5. V get(Object key): возвращает значение, связанное с указанным ключом, или null, если ключ отсутствует в карте.

6. boolean isEmpty(): возвращает true, если карта не содержит ни одной пары "ключ-значение".

7. Set<K> keySet(): возвращает множество всех ключей в карте.

8. V put(K key, V value): добавляет указанное значение по указанному ключу в карту, заменяя предыдущее значение, связанное с этим ключом (если оно было).

9. void putAll(Map<? extends K, ? extends V> m): добавляет все пары "ключ-значение" из указанной карты в данную карту.

10. V remove(Object key): удаляет пару "ключ-значение", связанную с указанным ключом, из карты и возвращает удаленное значение.

11. int size(): возвращает количество пар "ключ-значение" в карте.

12. Collection<V> values(): возвращает коллекцию всех значений в карте.

# Возможности Java Framework Collection. Контейнер HashMap и его основные методы.

Класс java.util.HashMap в Java Collection Framework используется для хранения сопоставления ключей со значениями на карте. Он обеспечивает реализацию интерфейса java.util.Map на основе хеш-таблицы и позволяет хранить элементы в виде пар ключ-значение.

# Коллекция HashMap, принципы создания и методы работы с ней

HashMap в Java - это класс, реализующий интерфейс Map и предоставляющий структуру данных для хранения пар "ключ-значение". Он использует хеш-таблицу для хранения данных, что обеспечивает высокую производительность операций вставки, удаления и поиска элементов.

Принципы создания HashMap:

1. Создание экземпляра HashMap с указанием типов ключа и значения:

HashMap<String, Integer> map = new HashMap<>();

Методы работы с HashMap:

1. Добавление элемента в HashMap:

map.put("ключ", значение);

2. Получение значения по ключу:

Integer value = map.get("ключ");

3. Проверка наличия ключа в HashMap:

boolean containsKey = map.containsKey("ключ");

4. Удаление элемента по ключу:

map.remove("ключ");

5. Получение множества всех ключей:

Set<String> keySet = map.keySet();

6. Получение коллекции всех значений:

Collection<Integer> values = map.values();

7. Получение множества пар "ключ-значение":

Set<Map.Entry<String, Integer>> entrySet = map.entrySet();

8. Проверка наличия элементов в HashMap:

boolean isEmpty = map.isEmpty();

HashMap также поддерживает итерацию через цикл for-each для обхода всех элементов.

for (Map.Entry<String, Integer> entry : map.entrySet()) {

String key = entry.getKey();

Integer value = entry.getValue();

// Действия с ключом и значением

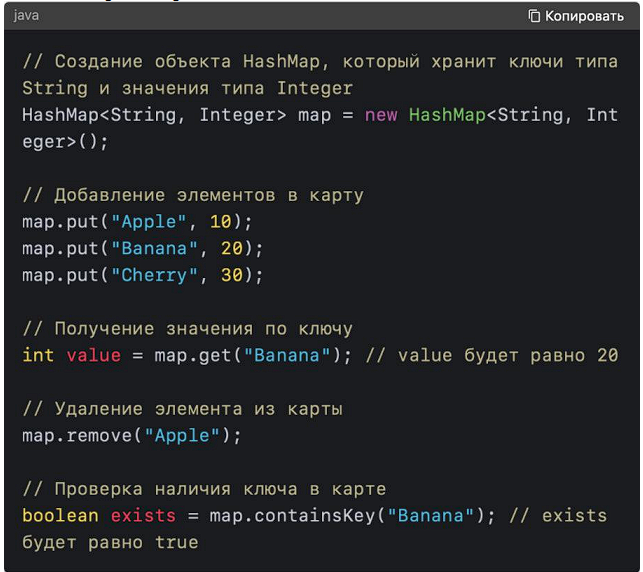
}

# Использование обобщенного класса HashMap, которая реализует интерфейс Map для хранения пар ключ-значение в разработке программ.

HashMap является обобщенным классом в Java, который реализует интерфейс Map. Это означает, что вы можете указать типы данных для ключей и значений при создании объекта HashMap.

Объект HashMap хранит данные в формате пары "ключ-значение". Это значит, что каждое значение в карте связано с уникальным ключом. Доступ к данным в карте осуществляется по ключу, что делает HashMap особенно полезным при работе с большими объемами данных.

Пример использования обобщенного класса HashMap:



HashMap также предоставляет методы такие как keySet(), который возвращает все ключи в карте, и values(), который возвращает все значения в карте.

Использование параметризованного HashMap (или любой другой коллекции Java) помогает обеспечить типобезопасность. Это означает, что вы можете уведомить компилятор о том, какие типы данных вы намереваетесь использовать, и получить ошибки во время компиляции, если вы случайно попытаетесь использовать неправильный тип данных.

# Возможности Java Framework Collection. Контейнер HashSet и его основные методы. Обобщенный класс HashSet класс коллекция, наследует свой функционал от класса AbstractSet, а также реализует интерфейс Set. Что он себя представляет?

Класс java.util.HashSet в Java Collection Framework используется для хранения коллекции уникальных элементов в set (наборе, можестве). Он обеспечивает реализацию интерфейса java.util.Set на основе хэш-таблицы.

Основные методы класса HashSet включают:

1. add(E e) - добавляет указанный элемент в этот набор, если он еще не присутствует.

2. remove(Object o) - удаляет указанный элемент из этого набора, если он присутствует.

3. clear() - удаляет все элементы из этого набора.

4. contains(Object o) - возвращает true, если этот набор содержит указанный элемент.

5. isEmpty() - проверяет, является ли этот набор пустым.

6. size() - возвращает количество элементов в этом наборе (его мощность).

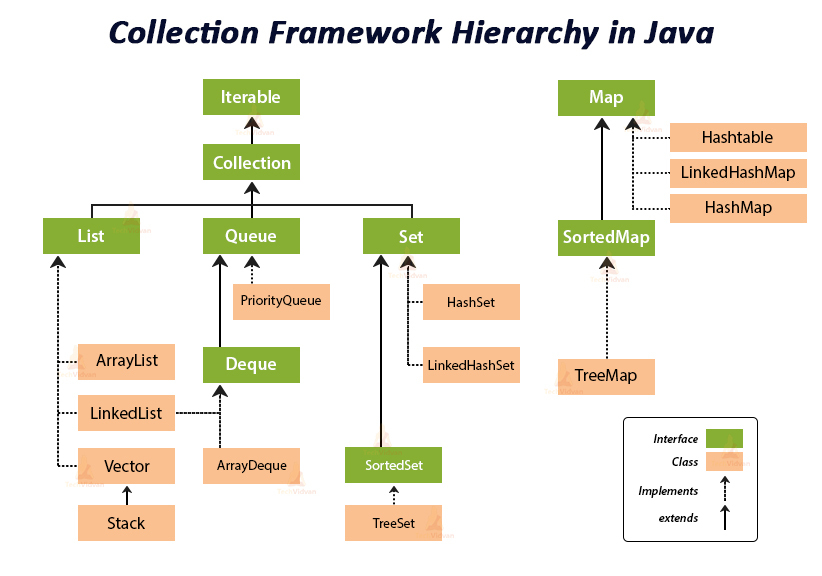
7. iterator() - возвращает итератор по элементам в этом наборе.

# Регулярные выражения и организация работы с ними в Java. Примеры

Регулярные выражения (regular expressions) представляют собой мощный инструмент для работы с текстом, позволяющий искать, заменять и анализировать подстроки в строках. В Java работа с регулярными выражениями осуществляется с использованием классов Pattern и Matcher из пакета java.util.regex.

https://javarush.com/groups/posts/regulyarnye-vyrazheniya-v-java

# Структура коллекций в Java Collection Framework. Иерархия интерфейсов



Основная иерархия интерфейсов:

1. \*\*Collection\*\* - это корневой интерфейс, который представляет общие методы для работы с коллекциями. Он имеет подинтерфейсы:

- \*\*List\*\* - представляет упорядоченную коллекцию, в которой могут быть дублирующиеся элементы.

- \*\*Set\*\* - представляет коллекцию, в которой каждый элемент уникален.

- \*\*Queue\*\* - представляет коллекцию, в которой элементы добавляются и удаляются в определенном порядке.

2. \*\*Map\*\* - представляет коллекцию пар ключ-значение, где каждый ключ уникален.

Интерфейсы List, Set и Queue имеют свои реализации в виде классов, таких как ArrayList, LinkedList, HashSet, TreeSet, PriorityQueue и другие. Интерфейс Map также имеет свои реализации, такие как HashMap, TreeMap, LinkedHashMap и т. д.

# Одним из ключевых методов интерфейса Collection является метод Iterator<E> iterator(). Что возвращает это метод?

Метод iterator() интерфейса Collection возвращает объект типа Iterator для перебора элементов коллекции. Интерфейс Iterator содержит методы для последовательного доступа к элементам коллекции, такие как hasNext() для проверки наличия следующего элемента и next() для получения следующего элемента в последовательности.

# Класс Pattern и его использование

Pattern Class – объект класса Pattern представляет скомпилированное представление регулярного выражения. В классе Pattern публичный конструктор не предусмотрен. Для создания шаблона, вам сперва необходимо вызвать один из представленных публичных статичных методов compile(), который далее произведет возврат объекта класса Pattern.

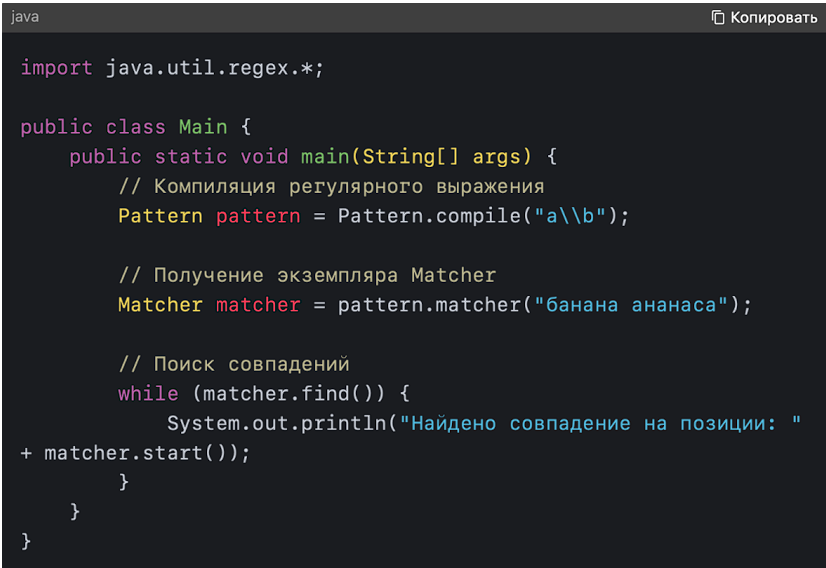
Основные методы класса Pattern включают:

- public static Pattern compile(String regex): этот метод используется для компиляции регулярного выражения в экземпляр класса Pattern.

- public Matcher matcher(CharSequence input): этот метод создает Matcher для данного входного символьного потока, который можно затем использовать для выполнения операций совпадения, поиска и замены.

- public static boolean matches(String regex, CharSequence input): этот метод является упрощенной формой использования регулярных выражений, который проверяет, соответствует ли вся входная последовательность данному регулярному выражению.

- public String pattern(): этот метод возвращает регулярное выражение, из которого был скомпилирован данный экземпляр Pattern.



# Класс Math и его использование

Класс Math содержит методы для того, чтобы выполнить основные числовые операции, такие как элементарный экспоненциал, логарифм, квадратный корень, и тригонометрические функции. В отличие от некоторых из числовых методов класса StrictMath, все реализации эквивалентных функций класса Math не определяются, чтобы возвратить поразрядное те же самые результаты.

* Класс Math используется Вычисление абсолютных значений (значений по модулю)
* Вычисление значений тригонометрических функций (синусов, косинусов и т.д.)
* Возведение в различные степени
* Извлечение корней различных степеней
* Генерация случайных чисел
* Округления

# Возможности Java Framework Collection. Интерфейс Map и его основные методы

Map: коллекция пар ключ-значение (key-value), где каждый ключ уникален.

Основные методы интерфейса Map включают следующие:

1. void clear(): удаляет все пары "ключ-значение" из карты.

2. boolean containsKey(Object key): возвращает true, если карта содержит указанный ключ.

3. boolean containsValue(Object value): возвращает true, если карта содержит указанное значение.

4. Set<Map.Entry<K, V>> entrySet(): возвращает множество объектов Map.Entry, представляющих все пары "ключ-значение" в карте.

5. V get(Object key): возвращает значение, связанное с указанным ключом, или null, если ключ отсутствует в карте.

6. boolean isEmpty(): возвращает true, если карта не содержит ни одной пары "ключ-значение".

7. Set<K> keySet(): возвращает множество всех ключей в карте.

8. V put(K key, V value): добавляет указанное значение по указанному ключу в карту, заменяя предыдущее значение, связанное с этим ключом (если оно было).

9. void putAll(Map<? extends K, ? extends V> m): добавляет все пары "ключ-значение" из указанной карты в данную карту.

10. V remove(Object key): удаляет пару "ключ-значение", связанную с указанным ключом, из карты и возвращает удаленное значение.

11. int size(): возвращает количество пар "ключ-значение" в карте.

12. Collection<V> values(): возвращает коллекцию всех значений в карте.

# Возможности Java Framework Collection. Контейнер Hashtable его основные методы.

Хеш-таблица - это структура данных, в которой данные хранятся в формате массива. Каждое значение данных имеет уникальное значение ключа. Если ключ известен, доступ к необходимым данным осуществляется очень быстро. Таким образом, операции вставки и поиска выполняются быстро независимо от размера данных. Хэш-таблица состоит из массива для хранения данных и хэширования для генерации индекса, в котором должен быть расположен элемент.

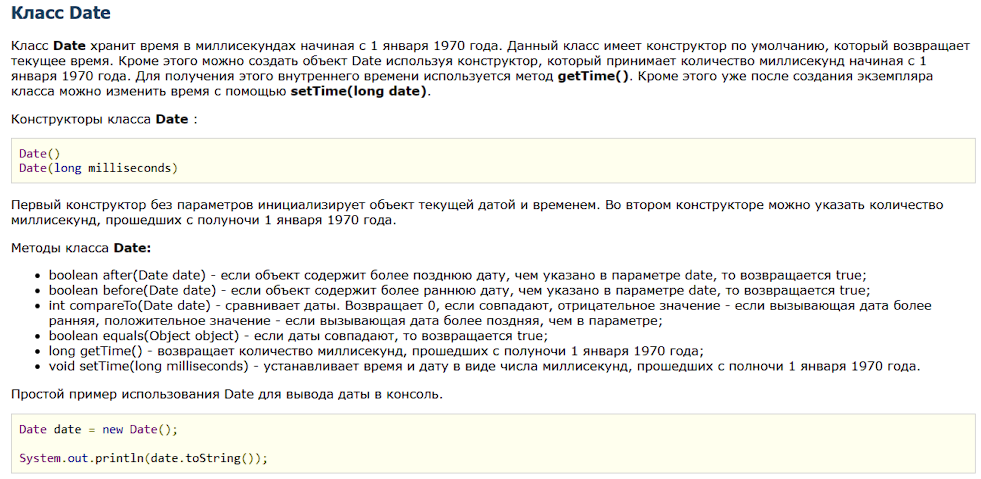
* Конкретный объект имеет определенный хэш-код.
* Два одинаковых объекта имеют одинаковые хэш-коды. Обратное неверно.
* Если два хэш-кода различаются, объекты определенно не равны.
* Разные объекты могут иметь одинаковый хэш-код. Это очень редкое событие вызывает столкновение. Хорошая хэш-функция минимизирует вероятность столкновений.

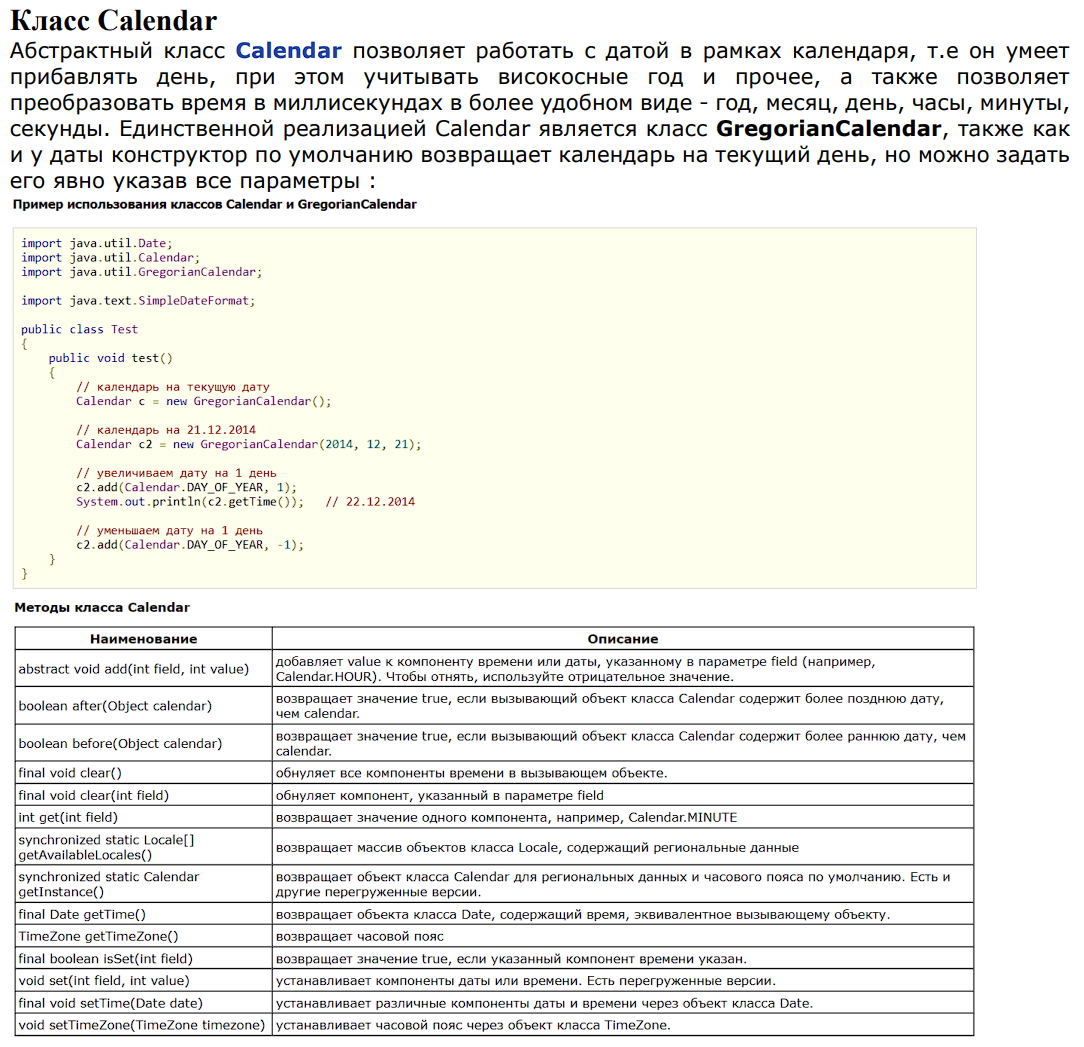
# Возможности Java Framework Collection. Интерфейс Iterator и Itarable.

<https://javarush.com/groups/posts/2308-korotko-o-glavnom---java-collections-framework>

Оба Iterator а также Iterable — это интерфейсы в Java, которые выглядят очень похожими и часто сбивают с толку новичков, но это две разные вещи. Короче говоря, если какой-либо класс реализует Iterable интерфейс, он получает возможность перебирать объект этого класса, используя Iterator.

# Работа с Датой и временем в Java. Примеры использования





Тема 8. Стандартные потоки ввода-вывода. Сериализация.

# Класс System. Стандартные поток ввода-вывода, предоставляемые Java. Работа со стандартами потоками вывода

В Java класс System предоставляет доступ к стандартным потокам ввода, вывода и ошибок. Эти потоки представлены следующими переменными:

- System.in: Представляет стандартный поток ввода. Этот поток обычно связан с клавиатурой.

- System.out: Представляет стандартный поток вывода. Этот поток обычно связан с консолью.

- System.err: Представляет стандартный поток ошибок. Этот поток также связан с консолью, но предназначен для вывода сообщений об ошибках.

Работа со стандартными потоками вывода в Java включает в себя вывод данных и сообщений в консоль с использованием методов, предоставляемых классом System:

- System.out.print(): Используется для вывода текста в стандартный поток вывода без перевода строки.

- System.out.println(): Используется для вывода текста в стандартный поток вывода с переводом строки.

- System.out.printf(): Используется для форматированного вывода данных в стандартный поток вывода с использованием форматов.

# Перегруженные методы out.println() класса System и их использование для вывода в консоль

* Обратиться к сущности, которая способна соединить наше приложение и консоль — System;
* Обратиться к потоку вывода на консоль — System.out;
* Вызвать метод, который записывает информацию на консоль — System.out.println;
* Передать текст, который необходимо записать — System.out.println(“Hello World!”);

# Класс Scanner. Ввод и вывод данных. Стандартные потоки ввода и вывода.

Класс Scanner используется для чтения ввода из стандартного потока или файлов. Он позволяет удобно читать различные типы данных из ввода. Стандартный поток ввода (клавиатура) в Java представлен объектом — System.in. А стандартный поток вывода (дисплей) — уже знакомым вам объектом System.out.

# Использование форматированного ввода-вывода в Java. Классы для форматирования вывода.

Для форматированного вывода в Java используются классы Formatter и String.format(). Они позволяют задавать формат вывода для различных типов данных.

# Понятие сериализации и ее использование в ООП программах. Использование интерфейса Serializable в программах на Джава

Сериализация — это процесс сохранения состояния объекта в последовательность байт. Механизм, который позволяет «записывать» полученный прогресс для будущей выгрузки.

Serializable - это маркерный интерфейс (не имеет элемента данных и метода). Он используется для “маркировки” классов java, чтобы объекты этих классов могли получить определенные возможности.

# Какие объекты можно сериализовать?

В Java можно сериализовать объекты, которые реализуют интерфейс Serializable. Этот интерфейс не определяет никаких методов, просто он служит указателем системе, что объект, реализующий его, может быть сериализован. Сериализация. Класс ObjectOutputStream.

# Какие методы определяет интерфейс Serializable?

Интерфейс Serializable обеспечивает автоматическую сериализацию, используя инструменты Java Object Serialization. Serializable не объявляет методов: он действует как маркер, сообщая инструментам Object Serialization, что класс Бина является сериализуемым. Пометка класса Serializable означает, что JVM сообщают, что уверены, что класс будет работать с сериализацией по умолчанию.

# Что означает понятие десериализация?

Десериализация — это процесс восстановления объекта из этих байт. Процедура выгрузки «зафиксированной» информации пользователем.

# Организация работы с файлами в Джава. Класс File, определенный в пакете java.io, не работает напрямую с потоками. В чем состоит его задача?

Класс File в Java предназначен для работы с файлами и директориями на файловой системе. Он предоставляет методы для создания, удаления, переименования файлов и директорий, а также для получения информации о файлах, такой как путь, размер, дата последнего изменения и другие атрибуты.

# При работе с объектом класса FileOutputStream происходит вызов метода FileOutputStream.write(), что в результате этого происходит?

Вызов метода write() объекта класса FileOutputStream приводит к записи данных в файл, на который указывает данный поток вывода (output stream). Метод write() принимает один байт данных или массив байтов и записывает их в файл.

Если вызывается метод FileOutputStream.write(byte[] b), то массив байтов b будет записан в файл. Если вызывается метод FileOutputStream.write(int b), то низший байт значения параметра b будет записан в файл.

Если файл уже существует, то вызов метода write() перезапишет существующие данные в файле. Однако можно использовать конструктор FileOutputStream с параметром, позволяющим добавлять данные к существующему файлу, если это необходимо.

При использовании метода write() также может возникнуть исключение IOException, поэтому необходимо обработать это исключение, например, с помощью конструкции try-catch или передать его выше в вызывающий код.

# Иерархия классов ввода вывода. Работа с файлами в Java. Работа с файлами. Сериализация объектов

Иерархия классов ввода-вывода в Java включает классы для работы с байтами (InputStream, OutputStream), символами (Reader, Writer) и файлами (FileInputStream, FileOutputStream, FileReader, FileWriter). Сериализация объектов выполняется с помощью классов ObjectInputStream и ObjectOutputStream.

**Сериализация** — это процесс сохранения состояния объекта в последовательность байт. Механизм, который позволяет «записывать» полученный прогресс для будущей выгрузки. **Десериализация** — это процесс восстановления объекта из этих байт. Процедура выгрузки «зафиксированной» информации пользователем.

В Java иерархия классов ввода-вывода предоставляет множество классов для работы с файлами, потоками данных, сериализацией и многими другими операциями ввода-вывода. Основные классы связанные с работой с файлами и сериализацией объектов включают в себя:

* java.io.File: Класс File представляет путь к файлу или директории и позволяет получать информацию о файлах, создавать и удалять файлы и директории, проверять их существование и многое другое.
* java.io.FileInputStream и java.io.FileOutputStream: Эти классы предоставляют функциональность для чтения из файла (FileInputStream) и записи в файл (FileOutputStream) соответственно.
* java.io.DataInputStream и java.io.DataOutputStream: Эти классы предоставляют методы для чтения и записи примитивных типов данных (int, double, boolean и т.д.) из/в поток ввода/вывода.
* java.io.ObjectInputStream и java.io.ObjectOutputStream: Эти классы позволяют сериализовать объекты (ObjectOutputStream) и десериализовать объекты (ObjectInputStream), то есть записывать объекты в файлы и считывать их обратно.
* java.io.Serializable: Интерфейс, который должен быть реализован классом, чтобы его объекты можно было сериализовать. Этот интерфейс не содержит методов, и служит только для присваивания классу маркера, что его объекты можно сериализовать.

Это лишь небольшая часть иерархии классов ввода-вывода в Java. Есть и другие классы для работы с различными типами данных и разными источниками данных (сетевые соединения, массивы байтов и т.д.). У каждого класса ввода-вывода есть свои особенности и применение, и выбор необходимого класса зависит от конкретной задачи.