**1. Введение в Java**

**1.1 Что такое Java**

На сегодняшний момент язык Java является одним из самых распространенных и популярных языков программирования. Первая версия языка появилась еще в 1996 году в недрах компании Sun Microsystems, впоследствии поглощенной компанией Oracle. Java задумывался как универсальный язык программирования, который можно применять для различного рода задач. И к настоящему времени язык Java проделал большой путь, было издано множество различных версий. Текущей версией является Java 21, которая вышла в сентябре 2023 года. А Java превратилась из просто универсального языка в целую платформу и экосистему, которая объединяет различные технологии, используемые для целого ряда задач: от создания десктопных приложений до написания крупных веб-порталов и сервисов. Кроме того, язык Java активно применяется для создания программного обеспечения для множества устройств: обычных ПК, планшетов, смартфонов и мобильных телефонов и даже бытовой техники. Достаточно вспомнить популярность мобильной ОС Android, большинство программ для которой пишутся именно на Java.

**Особенности Java**

Ключевой особенностью языка Java является то, что его код сначала транслируется в специальный байт-код, независимый от платформы. А затем этот байт-код выполняется виртуальной машиной JVM (Java Virtual Machine). В этом плане Java отличается от стандартных интерпретируемых языков как PHP или Perl, код которых сразу же выполняется интерпретатором. В то же время Java не является и чисто компилируемым языком, как С или С++.

Подобная архитектура обеспечивает кроссплатформенность и аппаратную переносимость программ на Java, благодаря чему подобные программы без перекомпиляции могут выполняться на различных платформах - Windows, Linux, Mac OS и т.д. Для каждой из платформ может быть своя реализация виртуальной машины JVM, но каждая из них может выполнять один и тот же код.

Java является языком с Си-подобным синтаксисом и близок в этом отношении к C/C++ и C#. Поэтому, если вы знакомы с одним из этих языков, то овладеть Java будет легче.

Еще одной ключевой особенностью Java является то, что она поддерживает автоматическую сборку мусора. А это значит, что вам не надо освобождать вручную память от ранее использовавшихся объектов, как в С++, так как сборщик мусора это сделает автоматически за вас.

**Где использовать?**

Java – высокоуровневый язык программирования, изначально разработанный компанией Sun Microsystems и выпущенный в 1995 году. Работает на различных платформах, таких как Windows, Mac OS, а также различных версий UNIX.

Java – это язык программирования общего назначения, предназначенный для того, чтобы программисты могли писать один раз и запускать где угодно (WORA). Это означает, что скомпилированный код Java может работать на всех платформах, поддерживающих Java, без необходимости повторной компиляции. Важные области использования Java включают:

1) **Веб-программирование**: Java EE поддерживает разработку корпоративных веб-приложений, веб-сервисов и порталов, обеспечивая множество API и библиотек для серверных компонентов, баз данных и сетевых протоколов.

2) **Мобильная разработка**: Java является основным языком для создания приложений под операционную систему Android, используемую на мобильных устройствах.

3) **Десктопные приложения**: JavaFX предоставляет средства для разработки графических десктопных приложений с интерактивным пользовательским интерфейсом.

4) **Работа с базами данных**: Java Database Connectivity (JDBC) позволяет взаимодействовать с реляционными базами данных, обеспечивая доступ к разным системам управления базами данных.

**5) Игровая разработка**: Java применяется для создания компьютерных игр и виртуальной реальности, используя библиотеки и фреймворки, такие как LibGDX и jMonkeyEngine.

**6) Научные и исследовательские приложения**: Java привлекателен для создания научных приложений, обработки данных и визуализации благодаря кроссплатформенности и богатому выбору научных библиотек.

**7) Системное и встроенное программирование**: Java применяется для создания системных приложений и программирования встроенных устройств, включая микроконтроллеры и встроенные системы.

**2.** **Синтаксис Java**

**2.1** **Переменные в Java**

Для хранения данных в программе предназначены переменные. Переменная представляет именованную область памяти, которая хранит значение определенного типа. Каждая переменная имеет тип, имя и значение. Тип определяет, какую информацию может хранить переменная (например, мы можем хранить число, слово, и даже целый список значений!).

Если обобщить, переменная - это некоторый контейнер, в котором может храниться значение для дальнейшего использования в программе.

**Объявление переменных в Java + присваивание**

Давайте рассмотрим такой пример:

int x = 9;

Разберем, что мы видим:

int — тип переменной, описывающий целые числа, находящиеся в промежутке -2147483648 до 2147483647 (есть также и другие обозначения, которые описывают большие или меньшие диапозоны чисел, о них мы будем говорить позже)

x — имя переменной (нам же нужно их отличать между собой, верно?)

В качестве имени переменной может выступать любое произвольное название, которое удовлетворяет следующим требованиям:

- имя может содержать любые алфавитно-цифровые символы, а также знак подчеркивания, при этом первый символ в имени не должен быть цифрой

- в имени не должно быть знаков пунктуации и пробелов

- имя не может быть ключевым словом языка Java

Кроме того, при объявлении и последующем использовании надо учитывать, что Java - регистрозависимый язык, поэтому следующие объявления int num; и int NUM; будут представлять две разных переменных.

= знак присваивания какой-то переменной, какого-то значения

9 — её непосредственной значение

; конец данной команды

А теперь соберём всё вместе: мы задаём, что переменная типа int, с именем x имеет значение 9.

Данный пример имеет сокращенную форму записи, полная выглядит так:

int x;

x = 9;

В первой строке мы видим объявление переменной с присвоением ей имени, то есть этим мы говорим JVM выделить место под переменную int (4 байта) и задать ей имя x.

Во второй мы задаём ей значение 9. До этого у нее было значение, установленное по умолчанию, а именно — 0.

Стоит сказать пару слов об именовании переменных. Как правило они пишутся в нижнем «верблюжьем стиле», от английского CamelCase. То есть, например, если у нас есть переменная, описывающая количество людей (count of people), подходящее для неё имя будет: countOfPeople;

В этом названии первое слово начинается со строчной (маленькой) буквы, а каждое следующее — с заглавной (большой). Это делается для удобства чтения таких имен, так как обычно имена переменных состоят из более чем одного слова.

**2.2** **Подробнее про целочисленные типы данных**

В Java существует несколько целочисленных (integer) типов данных, которые различаются по размеру и диапазону значений. Вот подробное описание типов данных byte, short, int и long:

**byte:**

Размер: 8 бит (1 байт).

Диапазон значений: от -128 до 127.

byte используется, когда нужно хранить маленькие целые числа, которые не выходят за пределы этого диапазона. Этот тип данных эффективен при работе с большими массивами байтов, например, при чтении и записи данных из файлов.

Пример объявления переменной типа byte:

byte age = 25;

**short:**

Размер: 16 бит (2 байта).

Диапазон значений: от -32,768 до 32,767.

short используется для хранения целых чисел в более широком диапазоне, чем byte, но все равно ограниченном. Он может быть полезен в приложениях, где требуется экономить память, но требуется больший диапазон значений, чем у byte.

Пример объявления переменной типа short:

short temperature = -100;

**int:**

Размер: 32 бита (4 байта).

Диапазон значений: от -2,147,483,648 до 2,147,483,647.

int - это стандартный тип для целых чисел в Java. Он позволяет работать с большими целыми числами и является наиболее распространенным целочисленным типом в языке. В большинстве случаев int будет достаточным для представления целых чисел.

Пример объявления переменной типа int:

int population = 1500000;

**long:**

Размер: 64 бита (8 байтов).

Диапазон значений: от -9,223,372,036,854,775,808 до 9,223,372,036,854,775,807.

long используется для хранения очень больших целых чисел, которые выходят за пределы диапазона int. Он может быть полезен, например, при работе с большими числами, временем или в некоторых вычислениях, где нужна высокая точность.

Пример объявления переменной типа long:

long bigNumber = 123456789012345L;

Обратите внимание, что при инициализации переменных типа long необходимо добавить суффикс L (или l) к числу, чтобы компилятор понял, что это long значение.

Выбор между этими типами данных зависит от требований конкретной задачи. Если вы не уверены, какой тип использовать, обычно безопаснее выбрать int, так как он предоставляет широкий диапазон и потребляет меньше памяти, чем long. Однако, в случае необходимости, всегда можно выбрать более подходящий тип для конкретной ситуации.

**2.3** **Тип данных boolean**

boolean - это простой логический тип данных в Java, который представляет два возможных значения: true (истина) и false (ложь). Тип boolean используется для хранения результатов логических операций, условных выражений и флагов, которые управляют выполнением программы. Вот основная информация о boolean:

**Значения true и false:**

true представляет истинное значение.

false представляет ложное значение.

Примеры:

boolean isJavaFun = true;

boolean isProgrammingHard = false;

**Инициализация boolean переменных**:

Переменные типа boolean могут быть инициализированы при объявлении или позднее в коде.

Если переменная не инициализирована явно, она будет иметь значение false по умолчанию.

Примеры:

boolean isSunny = true; // Инициализация при объявлении

boolean isWarm; // Неявная инициализация значением по умолчанию (false)

isWarm = true; // Поздняя инициализация

boolean играет важную роль в программировании, поскольку позволяет программистам принимать решения и управлять потоком выполнения программы на основе логических условий.

**2.4** **Операторы !, ||, &&**

Также в Java есть оператор «!». Оператор ! (логическое отрицание) в Java используется для инвертирования логического значения, то есть он изменяет true на false и наоборот. Оператор ! работает с переменными типа boolean и применяется для создания отрицательных условий в логических выражениях. Вот как это работает:

**Примеры использования оператора !:**

Инвертирование значения переменной:

boolean isSunny = true;

boolean isNotSunny = !isSunny; // isNotSunny будет равно false

&& (логическое И) и || (логическое ИЛИ) - это два основных логических оператора в Java, которые позволяют комбинировать и сравнивать логические значения (булевы значения true и false). Вот как они работают:

**&& (логическое И):**

Оператор && выполняет логическое И между двумя логическими выражениями.

Возвращает true, только если оба операнда равны true. В противном случае, возвращает false.

Примеры использования &&:

boolean isSunny = true;

boolean isWarm = true;

System.out.println(isSunny && isWarm);

В этом примере условие isSunny && isWarm истинно, так как оба операнда равны true.

**|| (логическое ИЛИ):**

Оператор || выполняет логическое ИЛИ между двумя логическими выражениями.

Возвращает true, если хотя бы один из операндов равен true. Возвращает false, только если оба операнда равны false.

Примеры использования ||:

boolean isSunny = true;

boolean isRainy = false;

System.out.println(isSunny || isRainy);

В этом примере условие isSunny || isRainy истинно, так как один из операндов равен true.

**Комбинированные логические выражения:**

Операторы && и || могут быть комбинированы в сложных логических выражениях.

Пример использования комбинированных операторов:

boolean isSunny = true;

boolean isWarm = false;

boolean isWeekend = true;

System.out.println(isSunny && (isWarm || isWeekend)));

В этом примере используются оба оператора && и ||, а также скобки для явного задания приоритета операций.

**Ленивые операторы:**

Операторы && и || - это ленивые операторы, что означает, что во время вычисления выражения они могут прервать выполнение, если результат уже известен. Например, если в выражении A && B значение A равно false, то B не будет вычисляться, так как результат всего выражения уже известен (false).

Это может быть полезно для оптимизации, особенно при выполнении вычислительно дорогих операций.

Логические операторы && и || играют важную роль в создании условных выражений, контроле потока выполнения программы и принятии решений на основе логических условий.

**2.5** **Несколько слов о выводе в консоль**

System.out.println - это стандартный метод в языке программирования Java, который используется для вывода текста или значений на консоль (стандартный вывод). Этот метод позволяет разработчикам отслеживать выполнение программы, выводить результаты и информацию для отладки. Вот как работает System.out.println:

**Синтаксис:**

Метод println является частью класса System в стандартной библиотеке Java. Синтаксис вызова System.out.println выглядит следующим образом:

System.out.println(выражение);

System - это класс, который предоставляет доступ к стандартным входным и выходным потокам, включая стандартный вывод.

println - это метод, который осуществляет вывод переданного ему выражения на стандартный вывод с добавлением перевода строки в конце вывода.

**Аргументы метода:**

выражение - это то, что вы хотите вывести на консоль. Это может быть текстовая строка, значение переменной, результат вычисления выражения и т. д.

**Примеры использования System.out.println:**

System.out.println("Привет, мир!"); // Вывод текстовой строки

int number = 42;

System.out.println("Число: " + number); // Вывод текста и значения переменной

**Перевод строки:**

Метод println автоматически добавляет символ перевода строки (\n) в конце вывода. Это делает следующий вывод на новой строке, что обеспечивает читабельность.

**Использование print:**

Если вы хотите вывести текст без автоматического перевода строки, вы можете использовать метод System.out.print. Этот метод не добавляет перевод строки и выводит текст в том же ряду, что и следующий вывод.

**Пример использования System.out.print:**

System.out.print("Это ");

System.out.print("вывод без перевода строки.");

Этот код выведет "Это вывод без перевода строки." в одной строке.

**Смешанный вывод:**

Вы можете смешивать вызовы System.out.print и System.out.println для создания вывода с переводом строк и без него по вашему усмотрению.

System.out.print("Первая строка.");

System.out.println(" Это вторая строка с переводом строки.");

System.out.print("Третья строка.");

**Этот код выведет:**

Первая строка. Это вторая строка с переводом строки.

Третья строка.

**2.6** **Чтение ввода с консоли**

Для чтения ввода с консоли в Java используется класс Scanner из пакета java.util. Давайте рассмотрим основные моменты работы с Scanner более подробно.

**Создание объекта Scanner:**

Scanner scanner = new Scanner(System.in);

Этот код создает новый объект Scanner, который будет считывать данные из стандартного потока ввода (System.in), который связан с консолью.

**Чтение строк:**

System.out.print("Введите ваше имя: ");

String name = scanner.nextLine();

System.out.println("Привет, " + name + "!");

Метод nextLine() считывает всю строку, введенную пользователем, включая пробелы.

**Чтение чисел:**

System.out.print("Введите ваш возраст: ");

int age = scanner.nextInt();

System.out.println("Ваш возраст: " + age);

Методы nextInt(), nextDouble(), и так далее, используются для чтения числовых значений. Они считывают только следующее число в строке.

**Проверка наличия данных:**

if (scanner.hasNextInt()) {

int number = scanner.nextInt();

System.out.println("Вы ввели число: " + number);

} else {

System.out.println("Это не число!");

}

Методы hasNextInt(), hasNextDouble(), и т.д. используются для проверки, является ли следующий ввод числом.

**Закрытие Scanner:**

scanner.close();

После использования Scanner важно закрыть его с помощью метода close(), чтобы избежать утечек ресурсов.

**2.7** **Пару слов о коментариях**

В примерах выше вы могли наблюдать необычные конструкции после написания кода. Это называется комментированием кода.

В Java комментарии - это специальные части кода, которые игнорируются компилятором и виртуальной машиной JVM при выполнении программы. Они служат для документирования кода, пояснения его структуры и функциональности, а также для временного отключения частей кода без их удаления. Вот как работают комментарии в Java:

**Однострочные комментарии:**

Однострочные комментарии начинаются с символа // и продолжаются до конца строки. Весь текст после // считается комментарием и игнорируется компилятором.

**Пример:**

int age = 25; // Это комментарий

Однострочные комментарии удобны для быстрых пояснений кода.

**Многострочные комментарии:**

Многострочные комментарии начинаются с /\* и заканчиваются \*/. Все, что находится между этими символами, считается комментарием и игнорируется компилятором. Многострочные комментарии могут занимать несколько строк.

**2.8** **Тип данных float и double**

float и double являются типами данных с плавающей запятой в Java и предназначены для хранения чисел с плавающей точкой, то есть чисел, которые могут иметь десятичную часть и/или экспоненту. Однако они отличаются по точности и размеру.

**Вот подробное описание float и double:**

**float:** Размер: 32 бита (4 байта).

Диапазон значений: от приблизительно -3.4 x 10^38 до 3.4 x 10^38.

float используется для представления чисел с плавающей запятой меньшей точности. Этот тип данных обычно используется, когда требуется экономия памяти, и точность вычислений не так важна, как, например, в вычислениях с плавающей точкой, связанных с графикой или сенсорами.

float temperature = 27.5f; // Обязательно добавьте суффикс "f" (или "F") к числу

**double:**

Размер: 64 бита (8 байтов).

Диапазон значений: от приблизительно -1.7 x 10^308 до 1.7 x 10^308.

double представляет числа с плавающей запятой более высокой точности по сравнению с float. Этот тип данных часто используется для научных вычислений, финансовых расчетов, обработки данных и других приложений, где точность играет важную роль.

double price = 49.99; // Для литералов с плавающей точкой суффикс не требуется

**Суффиксы и литералы:**

При использовании литералов (значений напрямую в коде) для float требуется добавить суффикс "f" или "F", например, 27.5f.

Для double суффикс не обязателен, например, 49.99.

**Опасность потери точности:**

Использование float может привести к потере точности в сложных математических операциях из-за его меньшей точности по сравнению с double. Поэтому, при необходимости высокой точности, рекомендуется использовать double.

**По умолчанию:**

Если вы используете десятичное значение без суффиксов, Java будет считать его double. Например, 3.14 будет воспринято как double, а не float.

**2.9** **Тип данных String + конкатенация**

String - это класс в Java, который представляет строковый тип данных. Он используется для хранения и обработки текстовых данных, таких как имена, сообщения, адреса электронной почты, текстовые файлы и многое другое. String является одним из самых часто используемых типов данных в Java. Вот некоторые важные особенности типа данных String:

**Создание строк:**

В Java строки можно создавать, просто используя двойные кавычки. Например:

String name = "John";

String message = "Hello, World!";

**Сцепление строк (concatenation):**

Для объединения строк можно использовать оператор +. Этот процесс называется сцеплением строк. Конкатенация (сцепление) - это процесс объединения строк или других данных в одну строку или последовательность. В Java, конкатенация строк выполняется с помощью оператора +. Когда вы используете + для объединения строк, вы создаете новую строку, содержащую объединенное содержимое.

Пример:

String firstName = "John";

String lastName = "Smith";

String fullName = firstName + " " + lastName; // "John Smith"

В этом примере + используется для объединения firstName, пробела и lastName, создавая новую строку fullName. Конкатенация полезна при создании динамических строк, сообщений и вывода текста.

**Индексирование строк:**

Символы в строке нумеруются, начиная с нуля. Вы можете получить доступ к символу в строке по его индексу.

Пример:

String text = "Hello";

char firstChar = text.charAt(0); // 'H'

char thirdChar = text.charAt(2); // 'l'

**Длина строки:**

Метод length() позволяет определить длину строки (количество символов в строке).

Пример:

String text = "Java";

int length = text.length(); // 4

**Сравнение строк:**

Для сравнения строк по их содержимому следует использовать метод equals(). Сравнение строк с помощью оператора == может работать не так, как ожидается, потому что он сравнивает ссылки на объекты, а не их содержимое.

String str1 = "Hello";

String str2 = "World";

boolean areEqual = str1.equals(str2); // false

**Неизменяемость строк:**

Строки в Java неизменяемы, что означает, что после создания строки ее содержимое нельзя изменить. Любые операции, такие как сцепление строк или замена символов, создают новую строку.

String original = "Hello";

String modified = original + ", World!"; // Создается новая строка

**Методы строк:**

Класс String предоставляет множество полезных методов для работы с текстовыми данными, такие как toUpperCase(), toLowerCase(), substring(), trim() и другие.

String text = "Java Programming";

String upperCase = text.toUpperCase(); // "JAVA PROGRAMMING"

String subString = text.substring(0, 4); // "Java"

String trimmed = " Hello, World! ".trim(); // "Hello, World!"

**Неявное создание строк:**

Java позволяет неявно создавать строки при конкатенации строк с другими типами данных. В этом случае вызывается метод toString() для другого типа данных.

int number = 42;

String result = "The answer is: " + number; // "The answer is: 42"

String является неотъемлемой частью разработки приложений на Java и широко используется для работы с текстовой информацией в программировании.

**2.10** **Арифметические операторы + инкремент/декремент**

В Java арифметические операторы используются для выполнения математических операций над переменными. Вот основные арифметические операторы:

**Сложение (+)**

int a = 5;

int b = 3;

int sum = a + b; // sum равно 8

**Вычитание (-)**

int x = 10;

int y = 4;

int difference = x - y; // difference равно 6

**Умножение (\*)**

int m = 3;

int n = 7;

int product = m \* n; // product равно 21

**Деление (/)**

int p = 20;

int q = 5;

int result = p / q; // result равно 4

**Остаток от деления (%)**

int num1 = 15;

int num2 = 4;

int remainder = num1 % num2; // remainder равно 3

**Инкремент (++) и Декремент (--)**

Операторы инкремента и декремента используются для увеличения или уменьшения значения переменной на 1 соответственно.

**Инкремент (++)**

int counter = 10;

counter++; // после этой операции counter равно 11

**Инкремент можно использовать и в выражениях**

int x = 5;

int result = x++ + 3; // result равно 8, но x стало равно 6

**Декремент (--)**

int value = 8;

value--; // после этой операции value равно 7

**Декремент также можно использовать в выражениях**

int y = 12;

int result = y-- \* 2; // result равно 24, но y стало равно 11

Инкремент и декремент можно применять как к целочисленным переменным, так и к переменным типа float или double.

**2.2.2** **Операторы сравнения**

Операторы сравнения в Java используются для сравнения значений и возвращают логическое значение true или false в зависимости от результата сравнения. Давайте рассмотрим основные операторы сравнения:

**Оператор равенства (==)**

int x = 5;

int y = 7;

boolean isEqual = (x == y); // isEqual равно false

**Оператор неравенства (!=)**

int a = 10;

int b = 10;

boolean notEqual = (a != b); // notEqual равно false

**Оператор больше (>)**

double p = 8.5;

double q = 5.2;

boolean isGreater = (p > q); // isGreater равно true

**Оператор меньше (<)**

int m = 15;

int n = 20;

boolean isLess = (m < n); // isLess равно true

**Оператор больше или равно (>=)**

int num1 = 12;

int num2 = 12;

boolean isGreaterOrEqual = (num1 >= num2); // isGreaterOrEqual равно true

**Оператор меньше или равно (<=)**

int value1 = 30;

int value2 = 40;

boolean isLessOrEqual = (value1 <= value2); // isLessOrEqual равно true

Эти операторы могут использоваться с различными типами данных, включая целочисленные, числа с плавающей точкой, символы и т.д.

**Замечания:**

Операторы сравнения возвращают булево значение (true или false).

Важно не путать операторы сравнения (==, !=, <, >, <=, >=) с операторами присваивания (=). Оператор сравнения использует два символа (==), в то время как оператор присваивания использует один символ (=).

Операторы сравнения могут быть использованы в условных операторах, циклах и других конструкциях для принятия решений в программе на основе результатов сравнений.

**2.11** **Конструкция if, else if, else (условные операторы)**

Условные операторы в Java позволяют выполнить различные блоки кода в зависимости от выполнения определенных условий. Конструкция if, else if, else используется для создания ветвлений в программе.

**Оператор if**

int x = 10;

if (x > 5) {

System.out.println("x больше 5");

}

Если условие внутри скобок ( и ) истинно, то выполняется блок кода внутри фигурных скобок { и }.

**Оператор else if**

int y = 3;

if (y > 5) {

System.out.println("y больше 5");

} else if (y > 0) {

System.out.println("y больше 0, но не больше 5");

}

Если первое условие в if ложно, то проверяется следующее условие в else if. Выполняется только первый блок кода, у которого условие истинно.

**Оператор else**

int z = -2;

if (z > 0) {

System.out.println("z больше 0");

} else if (z == 0) {

System.out.println("z равно 0");

} else {

System.out.println("z меньше 0");

}

Оператор else выполняет блок кода, если все предыдущие условия ложны. Этот блок кода в else не имеет условия, так как он выполняется, если ни одно из предыдущих условий не было истинным.

**Вложенные условные операторы**

int a = 15;

if (a > 5) {

System.out.println("a больше 5");

if (a < 20) {

System.out.println("a меньше 20");

}

}

Условные операторы могут быть вложенными, то есть один if может находиться внутри другого. В этом случае вложенный if будет проверяться только в том случае, если условие внешнего if истинно.

**2.12 Условные операторы (тернарный оператор)**

Тернарный оператор представляет собой сокращенную форму условного оператора if-else. Его синтаксис выглядит следующим образом:

variable = (condition) ? expressionIfTrue : expressionIfFalse;

**Пример использования тернарного оператора:**

int age = 20;

String result = (age >= 18) ? "Совершеннолетний" : "Несовершеннолетний";

System.out.println(result);

В этом примере, если age больше или равно 18, переменной result будет присвоено значение "Совершеннолетний", в противном случае - "Несовершеннолетний".

**Тернарный оператор возвращения значения:**

Тернарный оператор часто используется для присвоения значения переменной в зависимости от условия. Например:

int x = 10;

int y = 5;

int max = (x > y) ? x : y;

System.out.println("Максимальное значение: " + max);

Здесь переменной max присваивается значение x, если x > y, иначе ей присваивается значение y.

**Тернарный оператор в аргументах метода:**

Тернарный оператор может быть использован в аргументах метода:

int a = 15;

int b = 20;

int result = addNumbers((a > 10) ? a : b, b);

int addNumbers(int x, int y) {

return x + y;

}

Здесь метод addNumbers принимает в качестве аргументов результат тернарного оператора.

Тернарный оператор полезен, когда вы хотите присвоить переменной значение в зависимости от условия, и это значение зависит от результатов выражения, которое легко уместить в одной строке.

**2.13** **Оператор switch**

switch в Java является оператором выбора, который позволяет выбирать один из множества вариантов выполнения кода в зависимости от значения выражения. Оператор switch может быть использован для более чистого и лаконичного кода, чем последовательность операторов if-else if-else. Вот его общая структура:

switch (expression) {

case value1:

// блок кода, выполняемый при expression равном value1

break;

case value2:

// блок кода, выполняемый при expression равном value2

break;

// ...

default:

// блок кода, выполняемый, если ни одно из предыдущих условий не выполнено

}

**Пример использования switch:**

int dayOfWeek = 3;

String day;

switch (dayOfWeek) {

case 1:

day = "Понедельник";

break;

case 2:

day = "Вторник";

break;

case 3:

day = "Среда";

break;

case 4:

day = "Четверг";

break;

case 5:

day = "Пятница";

break;

case 6:

day = "Суббота";

break;

case 7:

day = "Воскресенье";

break;

default:

day = "Неверный день";

}

System.out.println("День недели: " + day);

В этом примере, в зависимости от значения переменной dayOfWeek, будет выбран соответствующий блок кода. Если значение переменной не соответствует ни одному из вариантов (case), будет выполнен блок кода в секции default.

**Важные моменты использования switch:**

Каждый case должен завершаться оператором break, иначе выполнение кода будет продолжено до следующего case.

В Java 7 и более поздних версиях, вы можете использовать строки в выражении switch.

Оператор default необязателен, но он предоставляет блок кода, который выполняется, когда не совпадает ни одно из значений case.

String fruit = "apple";

switch (fruit) {

case "apple":

System.out.println("Яблоко");

break;

case "orange":

System.out.println("Апельсин");

break;

default:

System.out.println("Неизвестный фрукт");

}

Оператор switch удобен, когда необходимо выбрать значение из набора возможных вариантов. В противном случае, оператор if-else if-else может быть более гибким в некоторых ситуациях.

**2.14** **Цикл for**

Цикл for в Java предоставляет компактный способ для повторения определенного блока кода определенное количество раз. Вот его общая структура:

for (инициализация; условие; выражение\_итерации) {

// тело цикла

}

**Инициализация (инициализация):**

Инициализация выполняется один раз в начале цикла. Здесь обычно устанавливаются начальные значения счетчиков или переменных, используемых в цикле.

**Условие (условие):**

Условие проверяется перед каждой итерацией цикла. Если условие истинно, тело цикла выполняется. Если условие ложно, цикл завершается.

**Выражение итерации (выражение\_итерации):**

Выражение итерации выполняется после каждой итерации цикла. Обычно здесь изменяются значения счетчиков или переменных.

**Тело цикла:**

Это блок кода, который будет выполняться, если условие цикла истинно.

**Пример использования цикла for:**

for (int i = 1; i <= 5; i++) {

System.out.println("Итерация номер " + i);

}

В этом примере:

int i = 1 - инициализация, устанавливаем начальное значение счетчика i равным 1.

i <= 5 - условие, цикл будет выполняться, пока i меньше или равно 5.

i++ - выражение итерации, после каждой итерации увеличиваем i на 1.

**Важные моменты:**

Инициализация, условие и выражение итерации могут содержать несколько выражений, разделенных запятыми.

Любая из трех частей цикла for может быть опущена, но точки с запятой ; должны присутствовать. Например:

int i = 1;

for (; i <= 5;) {

System.out.println("Итерация номер " + i);

i++;

}

В блоке итерации можно использовать операторы break и continue для изменения нормального потока выполнения цикла.

Цикл for обычно используется, когда известно точное количество итераций, которые необходимо выполнить.

**2.15** **Цикл while**

Цикл while в Java выполняет блок кода, пока указанное условие истинно. Вот его общая структура:

while (условие) {

// тело цикла

}

**Условие (условие):**

Это булево выражение, которое проверяется перед каждой итерацией цикла. Если условие истинно, тело цикла выполняется. Если условие ложно, выполнение цикла прекращается, и управление передается следующему блоку кода после цикла.

**Тело цикла:**

Это блок кода, который будет выполняться, если условие цикла истинно.

Пример использования цикла while:

int i = 1;

while (i <= 5) {

System.out.println("Итерация номер " + i);

i++;

}

В этом примере:

int i = 1 - инициализация, устанавливаем начальное значение переменной i равным 1.

i <= 5 - условие, цикл будет выполняться, пока i меньше или равно 5.

i++ - инкремент, после каждой итерации увеличиваем значение i на 1.

**Важные моменты использования цикла while:**

Условие должно быть булевым выражением. Если условие изначально ложно, тело цикла ни разу не выполнится.

Тело цикла должно содержать какие-то изменения, которые в конечном итоге сделают условие ложным. В противном случае цикл может выполняться вечно (бесконечный цикл).

В блоке тела цикла можно использовать операторы break и continue.

Важно обеспечить корректное изменение переменных в условии итерации, чтобы избежать бесконечных циклов.

Цикл while полезен, когда количество итераций заранее неизвестно, и выполнение блока кода зависит от условия.

**2.16** **Оператор continue, break**

**Оператор break:**

Оператор break используется для прерывания выполнения цикла или переключения из блока кода внутри условных операторов. Вот пример использования в цикле for:

for (int i = 1; i <= 10; i++) {

if (i == 5) {

break; // Прерывание цикла, если i равно 5

}

System.out.println(i);

}

В этом примере, при достижении i значения 5, оператор break прерывает выполнение цикла, и программа переходит к следующему блоку кода после цикла.

**Оператор continue:**

Оператор continue используется для пропуска текущей итерации цикла и перехода к следующей итерации. Вот пример использования в цикле for:

for (int i = 1; i <= 10; i++) {

if (i == 5) {

continue; // Пропуск текущей итерации, если i равно 5

}

System.out.println(i);

}

В этом примере, при достижении i значения 5, оператор continue пропускает оставшуюся часть кода внутри цикла для текущей итерации и переходит к следующей итерации.

**Пример использования в цикле while:**

int i = 1;

while (i <= 10) {

if (i % 2 == 0) {

i++;

continue; // Пропуск четных значений

}

System.out.println(i);

i++;

}

В этом примере, если i является четным числом, оператор continue пропускает оставшуюся часть кода внутри цикла для текущей итерации.

**Важные моменты:**

break и continue могут использоваться как в циклах for, так и в циклах while и do-while.

Оба оператора влияют только на ближайший цикл, в котором они находятся.

Использование операторов break и continue требует осторожности, поскольку они могут сделать код сложным для понимания и отладки.

Оба оператора могут использоваться с метками (label), чтобы прервать или продолжить выполнение цикла с конкретной меткой, если в программе есть вложенные циклы.

**3.** **Структуры данных**

**3.1** **Массивы и двумерные массивы**

**1) Одномерные массивы:**

Одномерные массивы в Java представляют собой упорядоченные коллекции элементов одного типа. Объявление и инициализация одномерного массива выглядит следующим образом:

// Объявление массива

тип\_данных[] имя\_массива;

// Инициализация массива

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

// Пример

int[] numbers = new int[5];

numbers[0] = 1;

numbers[1] = 2;

// ...

Одномерные массивы индексируются с нуля. Так, в массиве numbers, numbers[0] содержит 1, numbers[1] содержит 2 и так далее.

**2) Двумерные массивы:**

Двумерные массивы в Java представляют собой массивы массивов. Такой массив можно представить как таблицу, состоящую из строк и столбцов. **Объявление и инициализация двумерного массива:**

// Объявление массива

тип\_данных[][] имя\_массива;

// Инициализация массива

тип\_данных[][] имя\_массива = new тип\_данных[количество\_строк][количество\_столбцов];

// Пример

int[][] matrix = new int[3][4];

matrix[0][0] = 1;

matrix[0][1] = 2;

// ...

В этом примере matrix - это двумерный массив размером 3x4.

**Основные операции с массивами:**

Доступ по индексу: array[index] - получение элемента массива по указанному индексу.

Проход по массиву: с использованием цикла for или foreach.

Длина массива: array.length - возвращает количество элементов в массиве.

// Проход по одномерному массиву

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

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

}

// Проход по двумерному массиву

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

for (int j = 0; j < matrix[i].length; j++) {

System.out.println(matrix[i][j]);

}

}

**Особенности:**

Размер массива фиксирован при его создании.

В Java массивы являются объектами, поэтому можно использовать методы, предоставляемые классом Arrays для сортировки, копирования и других операций.

// Пример сортировки массива

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

Arrays.sort(numbers);

Многомерные массивы (трехмерные, четырехмерные и т.д.):

Аналогично, можно создавать многомерные массивы с более чем двумя измерениями, например, трехмерные массивы:

int[][][] threeDArray = new int[3][4][5];

Замечание: В Java массивы являются объектами, и их размер не может быть изменен после создания. Для динамического хранения данных можно использовать коллекции, такие как ArrayList.

**3.2** **ArrayList + списки**

ArrayList в Java является частью библиотеки коллекций и предоставляет динамический массив, который может изменять свой размер по мере необходимости. В отличие от обычных массивов, ArrayList позволяет легко добавлять, удалять и получать элементы. Вот основные характеристики ArrayList:

**1) Объявление и создание ArrayList:**

import java.util.ArrayList;

// Объявление

ArrayList<тип\_данных> имя\_списка;

// Создание

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

**2) Основные операции с ArrayList:**

**Добавление элемента:**

names.add("Alice");

names.add("Bob");

**Получение элемента по индексу:**

String firstPerson = names.get(0); // "Alice"

**Изменение элемента по индексу:**

names.set(1, "Charlie");

**Удаление элемента:**

names.remove("Alice"); // По значению

names.remove(0); // По индексу

**Проверка наличия элемента:**

boolean containsBob = names.contains("Bob");

**Размер ArrayList:**

int size = names.size();

**3) Проход по ArrayList:**

Существует несколько способов пройти по элементам ArrayList:

**Цикл for:**

for (int i = 0; i < names.size(); i++) {

System.out.println(names.get(i));

}

**Цикл for each:**

for (String name : names) {

System.out.println(name);

}

**4) Преобразование ArrayList в массив:**

String[] namesArray = names.toArray(new String[0]);

**5) Использование различных типов данных:**

ArrayList<Integer> numbers = new ArrayList<Integer>();

numbers.add(1);

numbers.add(2);

**Списки в Java:**

В Java интерфейс List представляет общий интерфейс для списков, таких как ArrayList, LinkedList и других. List предоставляет методы для работы с упорядоченными коллекциями, поддерживающими дублирование элементов. **Примеры операций List:**

import java.util.List;

import java.util.ArrayList;

List<String> names = new ArrayList<String>();

names.add("Alice");

names.add("Bob");

names.add("Charlie");

String firstPerson = names.get(0); // "Alice"

names.remove("Bob");

int size = names.size(); // 2

ArrayList - это конкретная реализация интерфейса List, предоставляющая динамический массив. Интерфейс List обеспечивает удобные методы для манипуляции данными в упорядоченной коллекции, что делает его одним из наиболее часто используемых интерфейсов коллекций в Java.

**3.3** **Множества**

В Java интерфейс Set представляет собой коллекцию, которая не допускает дублирования элементов. Он моделирует математическое понятие множества и предоставляет множество методов для манипуляции данными. Некоторые реализации Set включают HashSet, TreeSet, LinkedHashSet, и другие.

**1) Создание множества:**

import java.util.Set;

import java.util.HashSet;

Set<String> set = new HashSet<>();

**2) Основные операции с множеством:**

Добавление элемента:

set.add("apple");

set.add("banana");

Проверка наличия элемента:

boolean containsApple = set.contains("apple"); // true

Удаление элемента:

set.remove("banana");

Размер множества:

int size = set.size();

Проверка наличия элементов:

boolean isEmpty = set.isEmpty(); // false

**3) Итерация по множеству:**

for (String element : set) {

System.out.println(element);

}

**4) Использование различных реализаций Set:**

**HashSet:**

Реализация основана на хэш-таблице.

Порядок элементов может быть неопределенным.

Set<String> hashSet = new HashSet<>();

**TreeSet:**

Элементы упорядочены по возрастанию (естественный порядок или заданный компаратор).

Set<String> treeSet = new TreeSet<>();

LinkedHashSet:

Сохраняет порядок вставки элементов.

Set<String> linkedHashSet = new LinkedHashSet<>();

**5) Операции над множествами:**

Объединение двух множеств:

Set<String> set1 = new HashSet<>();

Set<String> set2 = new HashSet<>();

Set<String> union = new HashSet<>(set1);

union.addAll(set2);

Пересечение двух множеств:

Set<String> intersection = new HashSet<>(set1);

intersection.retainAll(set2);

Разность двух множеств:

Set<String> difference = new HashSet<>(set1);

difference.removeAll(set2);

**6) Иммутабельные множества:**

Java предоставляет неизменяемые (immutable) множества через методы Collections.unmodifiableSet:

Set<String> immutableSet = Collections.unmodifiableSet(originalSet);

Иммутабельные множества не могут быть изменены после создания.

**7) Примечания:**

Set является полезным инструментом для удаления дубликатов из коллекции и выполнения операций над уникальными элементами.

При выборе между различными реализациями следует учитывать требования к порядку элементов и производительности операций.

Реализации Set также могут быть использованы для проверки уникальности элементов в коллекциях, например, в списке.

**3.4** **Карты (Map)**

В Java интерфейс Map представляет ассоциативные коллекции, где каждый элемент представляет собой пару "ключ-значение". Ключи в Map уникальны, и каждый ключ связан с одним значением. Некоторые распространенные реализации интерфейса Map включают HashMap, TreeMap, LinkedHashMap, Hashtable, и другие.

**1) Создание карты:**

import java.util.Map;

import java.util.HashMap;

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

**2) Основные операции с картой:**

Добавление элемента:

map.put("apple", 5);

map.put("banana", 3);

Получение значения по ключу:

int apples = map.get("apple"); // 5

Проверка наличия ключа:

boolean containsKey = map.containsKey("apple"); // true

Проверка наличия значения:

boolean containsValue = map.containsValue(3); // true

Удаление элемента по ключу:

map.remove("banana");

Размер карты:

int size = map.size();

Проверка наличия элементов:

boolean isEmpty = map.isEmpty(); // false

**3) Итерация по картам:**

Итерация по ключам:

for (String key : map.keySet()) {

System.out.println("Key: " + key + ", Value: " + map.get(key));

}

Итерация по значениям:

for (int value : map.values()) {

System.out.println("Value: " + value);

}

Итерация по парам "ключ-значение":

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

System.out.println("Key: " + entry.getKey() + ", Value: " + entry.getValue());

}

**4) Реализации интерфейса Map:**

HashMap:

Основан на хэш-таблице.

Не гарантирует порядок элементов.

Map<String, Integer> hashMap = new HashMap<>();

TreeMap:

Элементы упорядочены по ключам в естественном порядке или по заданному компаратору.

Map<String, Integer> treeMap = new TreeMap<>();

LinkedHashMap:

Сохраняет порядок вставки элементов.

Map<String, Integer> linkedHashMap = new LinkedHashMap<>();

Hashtable:

Аналогичен HashMap, но синхронизирован (thread-safe).

Map<String, Integer> hashtable = new Hashtable<>();

**5) Иммутабельные карты:**

Map<String, Integer> immutableMap = Collections.unmodifiableMap(originalMap);

Иммутабельные карты не могут быть изменены после создания.

**6) Примечания:**

Map предоставляет эффективный способ хранения данных в виде ассоциаций "ключ-значение".

При выборе реализации Map следует учитывать требования к порядку элементов и производительности операций.

**3.5** **Стеки**

Стек в Java представляет собой коллекцию, в которой элементы управляются по принципу "последний вошел, первый вышел" (Last In, First Out, LIFO). Операции добавления элемента называются "push", а удаления — "pop".

**1) Создание стека:**

import java.util.Stack;

Stack<Integer> stack = new Stack<>();

**2) Основные операции со стеком:**

Добавление элемента (push):

stack.push(1);

stack.push(2);

Удаление элемента (pop):

int topElement = stack.pop(); // Возвращает и удаляет верхний элемент

Получение верхнего элемента без удаления (peek):

int topElement = stack.peek(); // Только возвращает верхний элемент, не удаляя его

Проверка наличия элементов:

boolean isEmpty = stack.isEmpty(); // true, если стек пуст

Размер стека:

int size = stack.size();

**3) Использование стека в алгоритмах:**

Стек широко используется в алгоритмах, особенно при работе с рекурсией, обходе деревьев, анализе выражений и других задачах.

**4) Пример использования стека:**

// Пример: Проверка сбалансированности скобок в выражении

public boolean isBalanced(String expression) {

Stack<Character> stack = new Stack<>();

for (char ch : expression.toCharArray()) {

if (ch == '(' || ch == '[' || ch == '{') {

stack.push(ch);

} else if (ch == ')' && !stack.isEmpty() && stack.peek() == '(') {

stack.pop();

} else if (ch == ']' && !stack.isEmpty() && stack.peek() == '[') {

stack.pop();

} else if (ch == '}' && !stack.isEmpty() && stack.peek() == '{') {

stack.pop();

} else {

return false; // Несбалансированные скобки

}

}

return stack.isEmpty(); // Сбалансированные скобки, если стек пуст

}

**5) Примечания:**

Stack в Java расширяет класс Vector и предоставляет удобные методы для работы со стеком.

Вместо Stack также можно использовать Deque (Double-ended queue), который предоставляет те же функции и является более современным вариантом.

Помимо push, pop и peek, Deque также предоставляет методы offerFirst и pollFirst, что позволяет использовать его как стек.

**4.** **Методы, вызовы методов**

**4.1 Методы, вызовы методов**

Методы в Java представляют собой блоки кода, которые выполняют определенные действия. Они используются для организации кода, уменьшения дублирования, и сделать программы более читаемыми и структурированными.

**1) Объявление метода:**

// Метод без возвращаемого значения и параметров

public void printHello() {

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

}

// Метод с возвращаемым значением и параметром

public int add(int a, int b) {

return a + b;

}

public: Модификатор доступа, который означает, что метод доступен из любого места в программе.

void: Тип возвращаемого значения. void означает, что метод ничего не возвращает.

printHello(): Имя метода.

( ): Список параметров (в данном случае, метод без параметров).

{ }: Тело метода, где размещаются операторы.

**2) Вызов метода:**

// Вызов метода без параметров и возвращаемого значения

printHello();

// Вызов метода с параметрами и возвращаемым значением

int sum = add(5, 3);

System.out.println("Sum: " + sum);

printHello(): Вызов метода без параметров и возвращаемого значения.

add(5, 3): Вызов метода с передачей параметров и получением возвращаемого значения.

**3) Статические методы:**

Методы могут быть статическими, что означает, что они принадлежат классу, а не экземпляру объекта.

public class MathOperations {

public static int square(int x) {

return x \* x;

}

}

Вызов статического метода:

int result = MathOperations.square(4);

**4) Примечания:**

Методы являются фундаментальными элементами структуры программы на Java.

Именование методов следует стандартам языка (camelCase).

Методы улучшают читаемость кода, обеспечивают модульность и облегчают сопровождение программы.

**4.2** **Параметры методов**

Параметры методов в Java представляют собой значения, которые передаются методу при его вызове. Они предоставляют методам информацию, необходимую для выполнения конкретных задач. Вот основные аспекты параметров методов:

**1) Объявление параметров:**

public void printMessage(String message, int count) {

for (int i = 0; i < count; i++) {

System.out.println(message);

}

}

В данном примере message и count - это параметры метода printMessage. Они объявляются в круглых скобках после имени метода.

String message - это параметр типа String с именем message.

int count - это параметр типа int с именем count.

**2) Передача аргументов:**

При вызове метода значения, которые передаются методу, называются аргументами.

printMessage("Hello, Java!", 3);

Здесь "Hello, Java!" - это аргумент для параметра message, а 3 - аргумент для параметра count.

**3) Порядок параметров:**

Порядок аргументов при вызове метода должен соответствовать порядку объявления параметров в методе.

printMessage(3, "Hello, Java!"); // Неверно! Типы и порядок не соответствуют.

**4) Параметры методов и возвращаемые значения:**

Метод может принимать ноль или более параметров и может возвращать значение. Возвращаемый тип указывается перед именем метода, а параметры - в круглых скобках.

public int add(int a, int b) {

return a + b;

}

В данном примере int a и int b - это параметры, а int перед именем метода - это возвращаемый тип.

**5) Параметры по умолчанию (с Java 8):**

С Java 8 появилась возможность использовать параметры по умолчанию. Это позволяет объявлять методы с параметрами, которые имеют значения по умолчанию, и их можно не передавать при вызове метода.

public void greet(String name, String greeting = "Hello") {

System.out.println(greeting + ", " + name + "!");

}

Теперь можно вызывать метод двумя способами:

greet("Alice"); // Выведет: Hello, Alice!

greet("Bob", "Hi there"); // Выведет: Hi there, Bob!

**6) Передача параметров по значению:**

В Java параметры метода передаются по значению, что означает, что при вызове метода передается копия значения параметра. Изменения, внесенные внутри метода, не влияют на оригинальные аргументы.

public void modifyValue(int x) {

x = x \* 2;

System.out.println("Inside method: " + x); // Выведет: Inside method: 20

}

int value = 10;

modifyValue(value);

System.out.println("Outside method: " + value); // Выведет: Outside method: 10

В данном примере value не изменяется после вызова метода.

**7) Примечания:**

Именование параметров следует стандартам языка (camelCase).

Параметры методов позволяют создавать более универсальные и гибкие методы, которые могут принимать различные входные данные.

Параметры по умолчанию облегчают использование методов с различными наборами аргументов.

Важно помнить, что изменения в параметрах метода не влияют на оригинальные аргументы, так как параметры передаются по значению.

**4.3** **Оператор return, возвращаемый тип**

Оператор return в Java используется для завершения выполнения метода и возврата значения вызывающему коду. Вместе с return указывается значение, которое должно быть возвращено. Здесь рассмотрим основные аспекты return и возвращаемых типов:

**1) Возвращаемый тип метода:**

Возвращаемый тип указывает, какой тип данных может вернуть метод. Он определяется перед именем метода.

public int add(int a, int b) {

return a + b;

}

В данном примере int перед именем метода add указывает на то, что этот метод вернет целочисленное значение.

**2) Возвращаемое значение:**

Оператор return используется для передачи значения из метода обратно в вызывающий код. Тип значения должен соответствовать объявленному возвращаемому типу метода.

public String greetPerson(String name) {

return "Hello, " + name + "!";

}

**3) Возвращаемый тип void:**

Если метод не возвращает значения, то возвращаемый тип указывается как void.

public void printMessage(String message) {

System.out.println(message);

}

В этом случае оператор return используется без значения, просто для завершения метода.

return;

**4) Использование return для завершения метода:**

Оператор return также используется для преждевременного завершения выполнения метода.

public int findPositiveNumber(int[] numbers) {

for (int num : numbers) {

if (num > 0) {

return num; // Возвращаем первое положительное число

}

}

return -1; // Если нет положительных чисел

}

В этом примере метод завершается, как только найдено положительное число в массиве.

**5) Примечания:**

Возвращаемый тип и тип, указанный в операторе return, должны совпадать или быть совместимыми.

Если возвращаемый тип метода не void, то каждый путь выполнения метода должен завершаться оператором return, который возвращает значение.

В случае, если метод объявлен с возвращаемым типом, но внутри отсутствует return, компилятор может выдать ошибку.

Оператор return также служит для передачи управления вызывающему коду.

**4.4** **Перегрузка методов**

Перегрузка методов - это возможность объявления нескольких методов с одним и тем же именем в одном классе, но с различными параметрами. Вот основные аспекты перегрузки методов:

**1) Перегрузка по параметрам:**

public class MathOperations {

public int add(int a, int b) {

return a + b;

}

public double add(double a, double b) {

return a + b;

}

public String add(String a, String b) {

return a + b;

}

}

Здесь add перегружен трижды: с параметрами int, double и String. Компилятор определяет, какой метод вызывать, на основе типов переданных аргументов.

**2) Перегрузка по количеству параметров:**

public class Display {

public void showMessage(String message) {

System.out.println(message);

}

public void showMessage(String message, int times) {

for (int i = 0; i < times; i++) {

System.out.println(message);

}

}

}

Здесь showMessage перегружен дважды: с одним параметром String и двумя параметрами String и int.

**3) Перегрузка по типам возвращаемого значения:**

Перегрузка методов также может происходить по типам возвращаемого значения, но в этом случае необходимо также изменять типы параметров, чтобы компилятор мог различить методы.

public class Result {

public int calculateResult(int a, int b) {

return a + b;

}

public double calculateResult(double a, double b) {

return a + b;

}

}

**4) Особенности перегрузки:**

Имя метода должно быть одинаковым для всех перегруженных методов.

Параметры или типы возвращаемого значения должны различаться.

Перегрузка может происходить в пределах одного класса.

Компилятор определяет, какой метод вызывать, исходя из типов и количества переданных аргументов.

**5) Примечания:**

Перегрузка методов повышает гибкость кода, позволяя использовать одно имя метода для различных сценариев.

Перегрузка методов не зависит от типа возвращаемого значения, только от параметров.

Важно, чтобы перегруженные методы не вызывали путаницы и были легко узнаваемыми в контексте их использования.

**4.5** **Рекурсивные методы/вызовы**

Рекурсия в программировании - это процесс, при котором метод вызывает сам себя. Рекурсивные методы используются для решения задач, которые могут быть разбиты на более простые подзадачи. Вот основные аспекты рекурсивных методов в Java:

**1) Основные компоненты рекурсивного метода:**

Базовый случай (base case): Это условие, при котором рекурсия завершается. Без базового случая рекурсивный метод будет вызывать себя бесконечно.

Шаг рекурсии (recursive step): Это шаг, в котором рекурсивный метод вызывает сам себя с новыми параметрами, приближая задачу к базовому случаю.

**2) Пример факториала:**

public class Factorial {

public static int calculateFactorial(int n) {

// Базовый случай

if (n == 0 || n == 1) {

return 1;

} else {

// Шаг рекурсии

return n \* calculateFactorial(n - 1);

}

}

public static void main(String[] args) {

int result = calculateFactorial(5);

System.out.println("Factorial of 5: " + result);

}

}

В этом примере метод calculateFactorial рекурсивно вызывает сам себя до достижения базового случая.

**3) Пример суммы массива:**

public class ArraySum {

public static int sumArray(int[] arr, int index) {

// Базовый случай

if (index == arr.length) {

return 0;

} else {

// Шаг рекурсии

return arr[index] + sumArray(arr, index + 1);

}

}

public static void main(String[] args) {

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

int sum = sumArray(array, 0);

System.out.println("Sum of array elements: " + sum);

}

}

В этом примере метод sumArray вычисляет сумму элементов массива с использованием рекурсии.

**4) Преимущества рекурсии:**

Рекурсия может сделать код более читаемым и понятным для решения определенных задач.

Она может предоставить более компактное и красивое решение для некоторых задач.

**5) Недостатки рекурсии:**

Рекурсивные вызовы требуют больше памяти, так как для каждого вызова создается новый фрейм стека.

Неправильная реализация может привести к бесконечной рекурсии.

**6) Примечания:**

Важно правильно выбирать базовый случай и шаг рекурсии, чтобы избежать бесконечной рекурсии.

Рекурсивные методы могут быть менее эффективными по сравнению с итеративными, поэтому их следует использовать там, где это имеет смысл и необходимо.

**5.** **Классы, ООП**

**5.1** **Понятие ООП**

Объектно-ориентированное программирование (ООП) - это парадигма программирования, основанная на концепции "объектов". В ООП программы рассматриваются как набор взаимодействующих объектов, каждый из которых является экземпляром класса. Вот ключевые понятия ООП:

**1) Пример:**

// Пример класса в Java

public class Animal {

// Поля (свойства)

String name;

int age;

// Конструктор

public Animal(String name, int age) {

this.name = name;

this.age = age;

}

// Метод

public void makeSound() {

System.out.println("Some generic sound");

}

}

// Пример подкласса, наследующего от Animal

public class Dog extends Animal {

// Конструктор подкласса

public Dog(String name, int age) {

super(name, age);

}

// Переопределенный метод

@Override

public void makeSound() {

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

}

// Дополнительный метод только для Dog

public void fetch() {

System.out.println("Fetching the ball");

}

}

**2) Преимущества ООП:**

Модульность: Код разбивается на небольшие, независимые модули (классы).

Повторное использование: Классы и объекты можно повторно использовать в различных частях программы.

Управляемость: Инкапсуляция и абстракция помогают управлять сложностью программы.

**3) Примечания:**

Java и ООП: Java является языком программирования, полностью ориентированным на объекты. Весь код в Java находится внутри классов и объектов.

Проектирование ООП: Проектирование с использованием ООП требует грамотного планирования и разбиения задач на классы и объекты.

**5.2** **Классы и объекты, экземпляры класса; поля, методы класса**

**1) Классы и объекты:**

Класс (Class): Класс - это шаблон или чертеж, описывающий состояние (поля) и поведение (методы) объекта. Например, класс "Собака" может иметь поля "Имя" и "Возраст", а также методы "Лаять" и "Бегать".

public class Dog {

// Поля класса

String name;

int age;

// Методы класса

void bark() {

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

}

void run() {

System.out.println(name + " is running.");

}

}

Объект (Object): Объект - это конкретный экземпляр класса. Например, если у нас есть класс "Собака", то объектами этого класса могут быть конкретные собаки, такие как "Барон" или "Лайка".

public class Main {

public static void main(String[] args) {

// Создание объекта класса Dog

Dog baron = new Dog();

// Установка значений полей объекта

baron.name = "Baron";

baron.age = 3;

// Вызов методов объекта

baron.bark();

baron.run();

}

}

**2) Экземпляры класса:**

Экземпляр (Instance): Экземпляр класса - это конкретный объект, созданный на основе определенного класса. В приведенном выше примере, baron - это экземпляр класса Dog.

**3) Поля и методы класса:**

Поля (Fields): Поля представляют собой переменные, которые хранят данные объекта. В примере с классом Dog, name и age - это поля.

Методы (Methods): Методы представляют собой функции или процедуры, которые могут выполняться объектом. В примере с классом Dog, bark() и run() - это методы.

public class Dog {

// Поля класса

String name;

int age;

// Методы класса

void bark() {

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

}

void run() {

System.out.println(name + " is running.");

}

}

**4) Создание объектов и доступ к их членам:**

Создание объекта: Для создания объекта используется ключевое слово new.

Dog baron = new Dog();

Доступ к полям и методам объекта: Доступ к полям и методам объекта осуществляется с использованием оператора точки ..

baron.name = "Baron";

baron.bark();

**5) Примечания:**

Классы и объекты являются основными строительными блоками в объектно-ориентированном программировании (ООП).

Класс определяет структуру и поведение объекта, а объект представляет конкретный экземпляр этого класса.

Поля определяют состояние объекта, а методы определяют его поведение.

**5.3** **Пакеты**

**1) Что такое пакет:**

Пакет (Package): Пакет в Java - это механизм организации классов и интерфейсов в иерархическую структуру. Он используется для группировки связанных классов, чтобы предотвратить конфликты имен и улучшить управление кодом.

**2) Преимущества использования пакетов:**

Изоляция: Пакеты предоставляют пространство имен, что позволяет изолировать классы одного пакета от классов другого пакета. Это уменьшает вероятность конфликтов имен.

Управление кодом: Пакеты упрощают организацию кода, делая его более легким для понимания и поддержки.

Безопасность: Модификаторы доступа (public, private, protected) могут быть использованы для ограничения доступа к членам пакета, что улучшает безопасность.

**3) Структура пакета:**

Объявление пакета: Объявление пакета должно быть указано в начале каждого файла класса.

package com.example.myproject;

Объявление пакета предшествует всем другим инструкциям в файле.

Иерархия пакетов: Пакеты могут быть вложенными, создавая иерархию. Например, пакет com.example может содержать в себе подпакет utils.

**4) Импорт классов из других пакетов:**

Импорт (Import): Для использования класса из другого пакета, его нужно импортировать. Инструкция импорта обычно следует за объявлением пакета и предшествует остальному коду класса.

package com.example.myproject;

import java.util.ArrayList;

import java.util.List;

**5) Пример:**

// Файл: com/example/myproject/Main.java

package com.example.myproject;

import com.example.myproject.utils.Utility;

public class Main {

public static void main(String[] args) {

Utility.printMessage("Hello from Main");

}

}

// Файл: com/example/myproject/utils/Utility.java

package com.example.myproject.utils;

public class Utility {

public static void printMessage(String message) {

System.out.println(message);

}

}

**6) Соглашения по именованию пакетов:**

Имена пакетов, как правило, записываются в нижнем регистре и следуют стандартам доменных имен в обратном порядке (например, com.example.myproject).

**7) Примечание:**

Java API предоставляет множество библиотек, организованных в пакеты. Например, стандартные классы коллекций находятся в пакете java.util.

**5.4** **Модификаторы доступа и инкапсуляция, абстракция**

**1) Модификаторы доступа:**

В Java существует четыре модификатора доступа:

public: Класс, метод или поле с модификатором public доступен из любого места, даже из других пакетов.

protected: Элементы с модификатором protected доступны внутри своего пакета и подклассам в любых пакетах.

default (package-private): Если не указан модификатор доступа (нет явного public, protected, или private), то элемент считается "пакетно-приватным" и доступным только внутри своего пакета.

private: Элемент с модификатором private доступен только внутри своего класса.

Пример:

public class Example {

public int publicField;

protected int protectedField;

int defaultField; // package-private

private int privateField;

}

**2) Инкапсуляция:**

Инкапсуляция (Encapsulation): Это концепция, согласно которой внутреннее представление объекта скрыто от внешнего мира, и доступ к данным осуществляется только через методы класса. Она обеспечивает контроль доступа к состоянию объекта и защиту от неправильного использования.

Пример:

public class BankAccount {

private double balance;

// Методы для доступа к полю balance

public double getBalance() {

return balance;

}

public void deposit(double amount) {

if (amount > 0) {

balance += amount;

}

}

public void withdraw(double amount) {

if (amount > 0 && amount <= balance) {

balance -= amount;

}

}

}

В этом примере поле balance является private, и к нему можно получить доступ только через методы getBalance(), deposit(), и withdraw(). Это предотвращает прямое изменение balance извне класса и позволяет контролировать операции с балансом.

**3) Абстракция:**

Абстракция (Abstraction): Это процесс выделения важных характеристик объекта и игнорирование неважных. В Java абстракция достигается с использованием абстрактных классов и интерфейсов.

Пример:

// Абстрактный класс

abstract class Shape {

abstract double getArea(); // Абстрактный метод

}

// Конкретный подкласс

class Circle extends Shape {

private double radius;

Circle(double radius) {

this.radius = radius;

}

@Override

double getArea() {

return Math.PI \* radius \* radius;

}

}

В этом примере Shape - абстрактный класс с абстрактным методом getArea(). Circle - конкретный подкласс, реализующий этот метод. Абстракция позволяет обобщить понятие формы (в данном случае) и использовать его для разных конкретных форм без дублирования кода.

**5.5** **Иерархия наследование и полиморфизм**

**1) Иерархия наследования:**

Наследование (Inheritance): Это механизм, позволяющий создавать новый класс на основе существующего, наследуя его свойства и методы. Класс, от которого наследуются, называется родительским или базовым классом, а новый класс - дочерним или производным классом.

Пример:

// Родительский класс

class Vehicle {

void start() {

System.out.println("Vehicle started");

}

}

// Дочерний класс, наследующий от Vehicle

class Car extends Vehicle {

void accelerate() {

System.out.println("Car is accelerating");

}

}

В этом примере Car наследует от Vehicle. Теперь Car имеет доступ ко всем методам Vehicle, а также может добавить свои собственные методы, например, accelerate().

**2) Полиморфизм:**

Полиморфизм (Polymorphism): Это концепция, которая позволяет объектам одного типа использоваться как объекты другого типа. В Java полиморфизм реализуется через механизмы перегрузки методов и динамического связывания.

Пример:

// Родительский класс

class Animal {

void makeSound() {

System.out.println("Some generic sound");

}

}

// Дочерние классы, переопределяющие метод makeSound

class Dog extends Animal {

@Override

void makeSound() {

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

}

}

class Cat extends Animal {

@Override

void makeSound() {

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

}

}

В этом примере Dog и Cat являются разными типами, но оба они являются подтипами Animal. С использованием полиморфизма мы можем создать массив или список объектов типа Animal и вызывать их методы, не заботясь о конкретном типе каждого объекта.

Пример использования полиморфизма:

public class Main {

public static void main(String[] args) {

Animal[] animals = new Animal[2];

animals[0] = new Dog();

animals[1] = new Cat();

for (Animal animal : animals) {

animal.makeSound(); // Вызов метода makeSound для каждого объекта, не зная их конкретного типа

}

}

}

В этом примере, вне зависимости от того, что находится в массиве animals, мы можем вызвать метод makeSound, и он будет вызван с соответствующей реализацией для каждого объекта.

**3) Перегрузка методов:**

Перегрузка методов (Method Overloading): Это механизм, позволяющий создавать несколько методов с одним и тем же именем, но разными параметрами.

Пример:

class Calculator {

int add(int a, int b) {

return a + b;

}

double add(double a, double b) {

return a + b;

}

}

Здесь add перегружен с двумя разными сигнатурами (типами параметров), что позволяет использовать его для работы с целыми числами и числами с плавающей запятой.

**4) Примечание:**

Наследование и полиморфизм являются ключевыми концепциями объектно-ориентированного программирования (ООП) и позволяют создавать гибкий и поддерживаемый код.

**5.6** **Интерфейсы и абстрактные классы**

**1) Абстрактные классы:**

Абстрактный класс (Abstract Class): Это класс, который не может быть инстанциирован (т.е., создан объект этого класса), и может содержать абстрактные методы (методы без реализации), а также обычные методы с реализацией. Абстрактные классы могут содержать поля и конструкторы.

Пример:

// Абстрактный класс

abstract class Shape {

// Абстрактный метод (без реализации)

abstract double getArea();

// Обычный метод с реализацией

void printMessage() {

System.out.println("This is a shape.");

}

}

// Конкретный подкласс

class Circle extends Shape {

private double radius;

Circle(double radius) {

this.radius = radius;

}

@Override

double getArea() {

return Math.PI \* radius \* radius;

}

}

В этом примере Shape - абстрактный класс с абстрактным методом getArea() и обычным методом printMessage(). Circle - конкретный подкласс, который расширяет Shape и предоставляет реализацию для getArea().

**2) Интерфейсы:**

Интерфейс (Interface): Это коллекция абстрактных методов. В интерфейсах все методы являются абстрактными, и они не содержат состояния (полей) и реализации методов. Класс может реализовать несколько интерфейсов.

Пример:

// Интерфейс

interface Printable {

void print();

}

// Класс, реализующий интерфейс

class Printer implements Printable {

@Override

public void print() {

System.out.println("Printing...");

}

}

Здесь Printable - интерфейс с единственным методом print(). Printer - класс, который реализует этот интерфейс и предоставляет конкретную реализацию для метода print().

**3) Различия между абстрактными классами и интерфейсами:**

Абстрактные классы:

Могут содержать как абстрактные, так и обычные методы.

Могут иметь поля и конструкторы.

Один класс может наследоваться только от одного абстрактного класса.

Интерфейсы:

Содержат только абстрактные методы (до Java 8, после Java 8 добавлены default и static методы).

Не содержат полей и конструкторов.

Класс может реализовывать несколько интерфейсов.

**4) Примечание:**

В Java 8 и последующих версиях добавлены лямбда-выражения и функциональные интерфейсы, что делает интерфейсы более мощными и удобными в использовании. Функциональный интерфейс - это интерфейс с одним абстрактным методом.

**5.7** **Перечисления enum**

В Java enum - это специальный тип данных, предназначенный для представления фиксированного набора констант. Перечисления в Java предоставляют удобный способ определения и использования констант.

1) Создание перечислений:

// Простое перечисление

enum Day {

SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY

}

// Перечисление с данными

enum Month {

JANUARY(31), FEBRUARY(28), MARCH(31), APRIL(30), MAY(31), JUNE(30),

JULY(31), AUGUST(31), SEPTEMBER(30), OCTOBER(31), NOVEMBER(30), DECEMBER(31);

private final int days;

Month(int days) {

this.days = days;

}

public int getDays() {

return days;

}

}

В приведенном примере Day - это простое перечисление, представляющее дни недели. Month - перечисление с данными, представляющее месяцы года с дополнительной информацией (количество дней в месяце).

2) Использование перечислений:

public class EnumExample {

public static void main(String[] args) {

// Простое перечисление

Day today = Day.MONDAY;

System.out.println("Today is " + today);

// Перечисление с данными

Month currentMonth = Month.JANUARY;

System.out.println("Days in " + currentMonth + ": " + currentMonth.getDays());

}

}

3) Методы в перечислениях:

Перечисления могут содержать методы:

enum TrafficLight {

RED("Stop"), YELLOW("Prepare to stop"), GREEN("Go");

private final String description;

TrafficLight(String description) {

this.description = description;

}

public String getDescription() {

return description;

}

}

4) Сравнение и перебор:

// Сравнение

if (today == Day.MONDAY) {

System.out.println("It's Monday!");

}

// Перебор значений

for (Day day : Day.values()) {

System.out.println(day);

}

5) Switch с перечислениями:

switch (today) {

case MONDAY:

System.out.println("It's Monday!");

break;

case TUESDAY:

System.out.println("It's Tuesday!");

break;

// и так далее

}

**6. Исключения**

**6.1** **Исключения. Обработка исключений**

**1) Исключения в Java:**

Исключение (Exception): Это событие, которое происходит во время выполнения программы и приводит к изменению нормального потока выполнения программы. Исключения могут быть вызваны различными факторами, такими как ошибка в коде, неверные входные данные или проблемы внешних ресурсов.

**Два типа исключений:**

Проверяемые (Checked): Наследуются от класса Exception, и компилятор обязывает их обработку.

Непроверяемые (Unchecked): Наследуются от класса RuntimeException, и их обработка не является обязательной.

**2) Обработка исключений:**

Try-Catch:

try {

// Код, который может вызвать исключение

} catch (ExceptionType1 e1) {

// Обработка исключения типа ExceptionType1

} catch (ExceptionType2 e2) {

// Обработка исключения типа ExceptionType2

} finally {

// Код, который выполняется всегда, независимо от наличия исключения

}

Блок finally: Код в блоке finally выполняется всегда, независимо от того, было ли исключение или нет. Блок finally часто используется для освобождения ресурсов (например, закрытия файлов).

**3) Пример:**

import java.io.FileReader;

import java.io.IOException;

public class ExceptionExample {

public static void main(String[] args) {

FileReader fileReader = null;

try {

// Открытие файла

fileReader = new FileReader("example.txt");

// Код для чтения файла

} catch (IOException e) {

// Обработка исключения ввода/вывода

e.printStackTrace();

} finally {

// Закрытие файла в блоке finally

try {

if (fileReader != null) {

fileReader.close();

}

} catch (IOException e) {

// Обработка исключения ввода/вывода при закрытии файла

e.printStackTrace();

}

}

}

}

В этом примере открывается файл (FileReader), и в блоке try выполняется код для чтения файла. Если возникает ошибка ввода/вывода (IOException), она обрабатывается в блоке catch. Блок finally используется для закрытия файла, даже если возникло исключение.

**6.2** **Оператор throws**

Оператор throws используется в сигнатуре метода для указания, что данный метод может бросать определенные исключения. Это предупреждение для вызывающего кода о возможности генерации исключений внутри метода.

**Синтаксис:**

type method\_name(parameters) throws exception\_type1, exception\_type2, ... {

// код метода

}

type: Тип возвращаемого значения метода.

method\_name: Имя метода.

parameters: Параметры метода.

throws exception\_type1, exception\_type2, ...: Список исключений, которые метод может бросать.

Пример:

import java.io.IOException;

public class Example {

// Метод, который может бросить IOException

void readFile(String fileName) throws IOException {

// код для чтения файла

}

public static void main(String[] args) {

Example example = new Example();

try {

example.readFile("example.txt");

} catch (IOException e) {

// обработка IOException

e.printStackTrace();

}

}

}

В этом примере метод readFile объявляет, что он может бросить исключение IOException. В методе main вызывающий код должен либо обработать это исключение с использованием блока catch, либо также указать throws IOException в своей сигнатуре метода.

**Когда использовать throws:**

Передача ответственности: Когда метод не способен адекватно обработать исключение, он может передать ответственность за обработку вызывающему коду, добавив throws в сигнатуру метода.

Соблюдение соглашения об исключениях: Если метод из интерфейса или абстрактного класса объявляет, что он может бросать исключение, то все его реализации или подклассы также должны уведомлять о том, что они могут бросать это исключение. Это соглашение об исключениях помогает вызывающему коду адекватно обрабатывать возможные ошибки.

**Примечания:**

throws в сигнатуре метода не обязательно означает, что метод всегда будет бросать исключение. Это просто указание, что метод может бросить исключение, и вызывающий код должен быть к этому готов.

Если метод бросает проверяемое исключение (checked exception), но не содержит кода, который может вызвать это исключение, компилятор также потребует обработки этого исключения в коде метода или добавления throws в сигнатуру метода.

**6.3** **Классы исключений**

В Java исключения представлены классами, которые являются частью иерархии исключений. Все классы исключений наследуются от базового класса Throwable. Иерархия исключений разделена на две основные ветви: Checked Exceptions (проверяемые исключения) и Unchecked Exceptions (непроверяемые исключения).

**Проверяемые исключения (Checked Exceptions):**

Exception (Исключение): Базовый класс для всех проверяемых исключений. От него наследуются многие другие классы, представляющие различные сценарии ошибок.

Примеры:

IOException: Исключение ввода/вывода.

SQLException: Исключение, связанное с работой с базой данных.

FileNotFoundException: Исключение, возникающее при попытке открыть несуществующий файл.

RuntimeException (Исключение времени выполнения): Основной класс для всех непроверяемых исключений.

Примеры:

NullPointerException: Попытка обращения к объекту через null.

ArrayIndexOutOfBoundsException: Выход за пределы массива.

ArithmeticException: Арифметическая ошибка, например, деление на ноль.

**Непроверяемые исключения (Unchecked Exceptions):**

RuntimeException (Исключение времени выполнения): Уже упомянутый базовый класс для всех непроверяемых исключений.

Примеры (повторение):

NullPointerException

ArrayIndexOutOfBoundsException

ArithmeticException

Error (Ошибка): Представляет серьезные проблемы, которые обычно не должны быть обработаны в программе. Наследуются от Throwable.

Примеры:

OutOfMemoryError: Нехватка памяти.

StackOverflowError: Переполнение стека.

**Ключевые моменты:**

Throwable: Базовый класс для всех исключений и ошибок.

Проверяемые исключения:

Наследуются от Exception.

Обычно отражают ошибки, которые можно предвидеть и обработать.

Непроверяемые исключения:

Наследуются от RuntimeException.

Обычно связаны с ошибками программирования и обычно не обрабатываются явно.

**6.4** **Создание своих классов исключений**

В Java вы можете создавать собственные классы исключений, чтобы более точно отразить сценарии ошибок, связанных с вашим приложением. Для этого нужно создать класс, который является подклассом существующего класса исключения. Вот пример создания своего класса исключения:

// Создание собственного класса исключения

class MyCustomException extends Exception {

MyCustomException(String message) {

// Конструктор, который вызывает конструктор родительского класса

super(message);

}

}

// Пример использования

public class ExceptionExample {

public static void main(String[] args) {

try {

// Генерация исключения

throw new MyCustomException("This is a custom exception");

} catch (MyCustomException e) {

// Обработка исключения

System.out.println(e.getMessage());

}

}

}

**Объяснение кода:**

MyCustomException: Это пользовательский класс исключения, который расширяет встроенный класс Exception.

MyCustomException(String message): Это конструктор класса, который принимает сообщение и передает его в конструктор родительского класса (Exception). Таким образом, при создании объекта MyCustomException с сообщением, это сообщение будет передано родительскому классу.

**Пример использования:**

В методе main создается исключение типа MyCustomException с сообщением "This is a custom exception" и затем обрабатывается в блоке catch. В блоке catch выводится сообщение об исключении.

**7. Потоки**

**7.1 Понятие потоков + закрытие. Реализация потоков**

Потоки (или нити) в Java представляют собой легковесные подпрограммы, которые выполняются независимо друг от друга и обладают собственным путем выполнения. Программа может выполнять несколько потоков параллельно, что позволяет эффективнее использовать ресурсы компьютера и повысить производительность.

В Java существует два основных способа создания потоков:

**Наследование от класса Thread:**

class MyThread extends Thread {

public void run() {

// Код, который будет выполнен в потоке

}

}

// Создание и запуск потока

MyThread myThread = new MyThread();

myThread.start();

**Реализация интерфейса Runnable:**

class MyRunnable implements Runnable {

public void run() {

// Код, который будет выполнен в потоке

}

}

// Создание и запуск потока

Thread myThread = new Thread(new MyRunnable());

myThread.start();

**Основные понятия:**

run() метод: Все потоки должны реализовать метод run(), в котором содержится код, выполняемый потоком.

start() метод: Метод start() запускает выполнение потока. Этот метод вызывает метод run(), но в отдельном потоке.

Многозадачность (Multithreading): Это концепция, при которой несколько потоков выполняются параллельно, что позволяет улучшить производительность программы.

**Закрытие потоков в Java:**

Закрытие потоков в Java обычно не требуется в явном виде, поскольку они уничтожаются автоматически после завершения выполнения метода run(). Однако есть несколько важных моментов:

Метод join(): Вы можете использовать метод join() для ожидания завершения выполнения потока. Например:

myThread.join(); // Ждем, пока myThread завершит выполнение

**Использование флага для завершения потока:**

Если у вас есть длительная операция в методе run(), вы можете использовать флаг для управления завершением потока:

class MyRunnable implements Runnable {

private volatile boolean isRunning = true;

public void run() {

while (isRunning) {

// Код выполнения в потоке

}

}

public void stop() {

isRunning = false;

}

}

В этом примере volatile используется для обеспечения видимости флага для других потоков.

**Использование метода interrupt():**

Метод interrupt() посылает потоку сигнал прерывания. Поток может периодически проверять свой статус прерывания с помощью метода isInterrupted() и завершить выполнение при необходимости.

class MyRunnable implements Runnable {

public void run() {

while (!Thread.currentThread().isInterrupted()) {

// Код выполнения в потоке

}

}

}

**7.2** **Многозадачность**

Многозадачность в программировании означает способность системы выполнять несколько задач (или процессов) одновременно. В контексте Java многозадачность может быть реализована с использованием потоков (threads).

**Виды многозадачности в Java:**

**Многозадачность на уровне процессов (Multiprocessing):**

Запуск нескольких независимых процессов, каждый из которых выполняется в отдельной области памяти.

Например, выполнение нескольких экземпляров Java приложения.

**Многозадачность на уровне потоков (Multithreading):**

Запуск нескольких потоков в рамках одного процесса, которые разделяют общую область памяти.

Каждый поток выполняет свою собственную задачу, что позволяет более эффективно использовать ресурсы компьютера.

**Потоки в Java:**

Класс Thread: Позволяет создавать и управлять потоками.

Интерфейс Runnable: Предоставляет более гибкий способ создания потоков, чем наследование от Thread.

**Пример многозадачности в Java:**

public class MultithreadingExample {

public static void main(String[] args) {

// Создание объекта Runnable

MyRunnable myRunnable = new MyRunnable();

// Создание и запуск потока

Thread thread1 = new Thread(myRunnable);

thread1.start();

// Запуск второго потока

Thread thread2 = new Thread(new MyRunnable());

thread2.start();

// Выполнение основного потока

for (int i = 0; i < 5; i++) {

System.out.println("Main Thread: " + i);

}

}

}

// Реализация интерфейса Runnable

class MyRunnable implements Runnable {

public void run() {

// Код, который будет выполнен в потоке

for (int i = 0; i < 5; i++) {

System.out.println("Child Thread: " + i);

}

}

}

В этом примере создаются два потока, один из которых запускается через объект Thread, а второй — напрямую. Оба потока выполняют код из метода run() объекта MyRunnable. В то время как они выполняются параллельно, основной поток также продолжает свое выполнение. При выполнении данного кода могут возникнуть различные комбинации вывода, так как потоки выполняются независимо друг от друга.

**7.3** **Синхронизация**

Синхронизация в Java - это механизм, который обеспечивает правильное выполнение многопоточных программ, предотвращая возможные проблемы согласованности данных, такие как состояние гонки и доступ к общим ресурсам. Она обеспечивает контроль над тем, какие потоки могут одновременно выполняться, а какие должны ждать.

**Основные моменты синхронизации в Java:**

**Синхронизированные методы:**

Синхронизированный метод блокируется для других потоков до тех пор, пока поток не выполнит метод.

Может быть применен к целому методу:

public synchronized void synchronizedMethod() {

// Код, требующий синхронизации

}

**Синхронизированные блоки:**

Синхронизированные блоки предоставляют более гибкий уровень синхронизации, чем синхронизированные методы.

Блокировка происходит только для кода, находящегося внутри блока.

public void someMethod() {

// Несинхронизированный код

synchronized (lockObject) {

// Код, требующий синхронизации

}

// Несинхронизированный код

}

**Мониторы и блокировки:**

В Java каждый объект имеет ассоциированный монитор (или блокировку), который используется для управления доступом к объекту.

Синхронизация происходит с использованием оператора synchronized или методов wait() и notify().

synchronized (lockObject) {

// Критическая секция, требующая синхронизации

}

**Volatile переменные:**

Ключевое слово volatile используется для переменных, которые могут быть изменены несколькими потоками.

Обеспечивает гарантию видимости изменений в переменной между потоками.

private volatile int sharedVariable;

Методы wait(), notify(), notifyAll():

Эти методы используются для согласования потоков внутри критических секций.

wait(): Переводит поток в режим ожидания, пока другой поток не вызовет notify() или notifyAll().

notify(): Пробуждает один поток, находящийся в режиме ожидания.

notifyAll(): Пробуждает все потоки, находящиеся в режиме ожидания.

**Пример синхронизации:**

class Counter {

private int count = 0;

public synchronized void increment() {

count++;

}

public synchronized int getCount() {

return count;

}

}

public class SynchronizationExample {

public static void main(String[] args) {

Counter counter = new Counter();

Runnable task = () -> {

for (int i = 0; i < 1000; i++) {

counter.increment();

}

};

// Создание двух потоков, которые изменяют счетчик

Thread thread1 = new Thread(task);

Thread thread2 = new Thread(task);

// Запуск потоков

thread1.start();

thread2.start();

try {

// Ожидание завершения потоков

thread1.join();

thread2.join();

} catch (InterruptedException e) {

e.printStackTrace();

}

// Вывод результата

System.out.println("Count: " + counter.getCount());

}

}

В этом примере объект Counter имеет два синхронизированных метода (increment() и getCount()), что предотвращает состояние гонки при изменении и чтении переменной count.

**8.** **Работы с файлами**

**8.1** **Создание файла**

В Java создание файла можно выполнить с использованием классов из пакета java.nio.file или java.io. В этом примере я расскажу о создании файла с использованием класса Files из пакета java.nio.file.

**Создание файла с использованием java.nio.file:**

import java.io.IOException;

import java.nio.file.Files;

import java.nio.file.Path;

import java.nio.file.Paths;

public class CreateFileExample {

public static void main(String[] args) {

// Путь к файлу

String filePath = "C:\\example.txt";

// Создание объекта Path

Path path = Paths.get(filePath);

try {

// Создание файла

Files.createFile(path);

System.out.println("File created successfully.");

} catch (IOException e) {

System.err.println("Unable to create file: " + e.getMessage());

}

}

}

В этом примере:

Указывается путь к файлу в переменной filePath.

Создается объект Path с использованием статического метода Paths.get().

С использованием Files.createFile(path) создается файл по указанному пути.

Пожалуйста, помните, что при создании файла может возникнуть IOException, и его необходимо обработать.

**Создание файла с использованием java.io:**

import java.io.File;

import java.io.IOException;

public class CreateFileExample {

public static void main(String[] args) {

// Путь к файлу

String filePath = "C:\\example.txt";

// Создание объекта File

File file = new File(filePath);

try {

// Создание файла

if (file.createNewFile()) {

System.out.println("File created successfully.");

} else {

System.out.println("File already exists.");

}

} catch (IOException e) {

System.err.println("Unable to create file: " + e.getMessage());

}

}

}

В этом примере:

Указывается путь к файлу в переменной filePath.

Создается объект File с использованием этого пути.

С использованием file.createNewFile() создается файл. Если файл уже существует, метод вернет false.

Оба этих примера создадут файл с указанным путем. Выбор зависит от ваших предпочтений.

**8.2** **Чтение и запись в файл**

Чтение и запись в файл в Java выполняется с использованием различных классов из пакетов java.nio.file и java.io. Ниже представлены примеры использования обоих подходов.

**Чтение файла с использованием java.nio.file:**

import java.io.IOException;

import java.nio.file.Files;

import java.nio.file.Path;

import java.nio.file.Paths;

import java.util.List;

public class ReadFileExample {

public static void main(String[] args) {

// Путь к файлу

String filePath = "C:\\example.txt";

// Создание объекта Path

Path path = Paths.get(filePath);

try {

// Чтение всех строк из файла

List<String> lines = Files.readAllLines(path);

// Вывод содержимого файла

for (String line : lines) {

System.out.println(line);

}

} catch (IOException e) {

System.err.println("Unable to read file: " + e.getMessage());

}

}

}

**Запись в файл с использованием java.nio.file:**

import java.io.IOException;

import java.nio.file.Files;

import java.nio.file.Path;

import java.nio.file.Paths;

import java.util.Arrays;

import java.util.List;

public class WriteFileExample {

public static void main(String[] args) {

// Путь к файлу

String filePath = "C:\\example.txt";

// Создание объекта Path

Path path = Paths.get(filePath);

// Содержимое для записи в файл

List<String> linesToWrite = Arrays.asList("Hello, World!", "This is a sample file.");

try {

// Запись строк в файл

Files.write(path, linesToWrite);

System.out.println("File written successfully.");

} catch (IOException e) {

System.err.println("Unable to write to file: " + e.getMessage());

}

}

}

**Запись в файл с использованием java.io:**

import java.io.FileWriter;

import java.io.IOException;

import java.io.PrintWriter;

public class WriteFileExample {

public static void main(String[] args) {

// Путь к файлу

String filePath = "C:\\example.txt";

try {

// Открытие файла для записи

PrintWriter writer = new PrintWriter(new FileWriter(filePath));

// Запись строк в файл

writer.println("Hello, World!");

writer.println("This is a sample file.");

// Закрытие файла

writer.close();

System.out.println("File written successfully.");

} catch (IOException e) {

System.err.println("Unable to write to file: " + e.getMessage());

}

}

}

Оба эти примера демонстрируют простой способ чтения и записи файлов в Java. Обратите внимание, что при работе с файлами может возникнуть IOException, его следует обрабатывать.

**8.3** **Удаление в файле**

Удаление файла в Java можно выполнить с использованием классов из пакета java.nio.file или java.io. Вот примеры использования обоих подходов.

**Удаление файла с использованием java.nio.file:**

import java.io.IOException;

import java.nio.file.Files;

import java.nio.file.Path;

import java.nio.file.Paths;

public class DeleteFileExample {

public static void main(String[] args) {

// Путь к файлу

String filePath = "C:\\example.txt";

// Создание объекта Path

Path path = Paths.get(filePath);

try {

// Удаление файла

Files.delete(path);

System.out.println("File deleted successfully.");

} catch (IOException e) {

System.err.println("Unable to delete file: " + e.getMessage());

}

}

}

**Удаление файла с использованием java.io:**

import java.io.File;

public class DeleteFileExample {

public static void main(String[] args) {

// Путь к файлу

String filePath = "C:\\example.txt";

// Создание объекта File

File file = new File(filePath);

// Удаление файла

if (file.delete()) {

System.out.println("File deleted successfully.");

} else {

System.err.println("Unable to delete file.");

}

}

}

**9.** **Графический интерфейс с JavaFX**

**9.1 Создание графических приложений. Установка и демонстрация работы.**

JavaFX предоставляет богатые средства для создания графических приложений с использованием Java. Вот основные шаги по созданию графического приложения с помощью JavaFX:

**Шаг 1: Установка JavaFX**

Убедитесь, что у вас установлен JavaFX или JDK с поддержкой JavaFX. Начиная с Java 11, JavaFX больше не включен в JDK по умолчанию. Вы можете использовать отдельный пакет JavaFX или JDK с модулем JavaFX.

**Шаг 2: Создание проекта**

Создайте новый проект в вашей среде разработки (например, IntelliJ IDEA, Eclipse) и добавьте необходимые библиотеки JavaFX. Если вы используете Maven или Gradle, вы можете добавить зависимости для JavaFX.

**Шаг 3: Создание основного класса**

Создайте основной класс вашего приложения. Этот класс должен наследоваться от класса Application из пакета javafx.application.

import javafx.application.Application;

import javafx.scene.Scene;

import javafx.scene.control.Button;

import javafx.scene.layout.StackPane;

import javafx.stage.Stage;

public class MyJavaFXApp extends Application {

public static void main(String[] args) {

launch(args);

}

@Override

public void start(Stage primaryStage) {

primaryStage.setTitle("My JavaFX App");

Button btn = new Button("Click me!");

btn.setOnAction(e -> System.out.println("Hello, JavaFX!"));

StackPane root = new StackPane();

root.getChildren().add(btn);

primaryStage.setScene(new Scene(root, 300, 250));

primaryStage.show();

}

}

**Шаг 4: Запуск приложения**

В методе start создайте пользовательский интерфейс, который будет отображаться в вашем приложении. В приведенном выше примере создается простое окно с кнопкой. Метод launch(args) запускает приложение.

**Шаг 5: Запуск приложения**

В классе с методом main создайте объект вашего основного класса и вызовите метод launch(args).

public class Main {

public static void main(String[] args) {

MyJavaFXApp.launch(args);

}

}

**Шаг 6: Сборка и запуск**

Соберите и запустите ваш проект. Вы должны увидеть окно приложения с кнопкой, которая выводит "Hello, JavaFX!" в консоль при нажатии.

Это основы создания графического приложения с использованием JavaFX.

**9.2** **Обработка событий и пользовательского ввода**

В JavaFX события происходят, когда пользователь взаимодействует с элементами интерфейса, такими как кнопки, текстовые поля и другие элементы. Обработка событий позволяет приложению реагировать на действия пользователя. Вот пример обработки событий и пользовательского ввода в JavaFX:

**Обработка событий кнопки:**

import javafx.application.Application;

import javafx.scene.Scene;

import javafx.scene.control.Button;

import javafx.scene.layout.StackPane;

import javafx.stage.Stage;

public class EventHandlingExample extends Application {

public static void main(String[] args) {

launch(args);

}

@Override

public void start(Stage primaryStage) {

primaryStage.setTitle("Event Handling Example");

Button btn = new Button("Click me!");

// Обработчик события при нажатии кнопки

btn.setOnAction(e -> handleButtonClick());

StackPane root = new StackPane();

root.getChildren().add(btn);

primaryStage.setScene(new Scene(root, 300, 250));

primaryStage.show();

}

// Метод, который вызывается при нажатии кнопки

private void handleButtonClick() {

System.out.println("Button Clicked!");

}

}

**Обработка событий мыши:**

import javafx.application.Application;

import javafx.scene.Scene;

import javafx.scene.layout.StackPane;

import javafx.stage.Stage;

public class MouseEventExample extends Application {

public static void main(String[] args) {

launch(args);

}

@Override

public void start(Stage primaryStage) {

primaryStage.setTitle("Mouse Event Example");

StackPane root = new StackPane();

// Обработчик событий мыши при наведении курсора

root.setOnMouseEntered(e -> handleMouseEntered());

// Обработчик событий мыши при уходе курсора

root.setOnMouseExited(e -> handleMouseExited());

primaryStage.setScene(new Scene(root, 300, 250));

primaryStage.show();

}

private void handleMouseEntered() {

System.out.println("Mouse Entered!");

}

private void handleMouseExited() {

System.out.println("Mouse Exited!");

}

}

В этих примерах создаются простые графические приложения с одной кнопкой или панелью, и к ним добавляются обработчики событий.

**9.3** **Разработка пользовательских интерфейсов**

Разработка пользовательских интерфейсов в JavaFX включает в себя создание разнообразных элементов управления, их стилизацию и управление их расположением. JavaFX предоставляет богатый набор встроенных элементов управления и возможности для их настройки.

**Создание разнообразных элементов управления:**

JavaFX предоставляет широкий набор элементов управления, таких как кнопки, текстовые поля, метки, таблицы, списки и многое другое. Пример создания некоторых элементов:

import javafx.application.Application;

import javafx.scene.Scene;

import javafx.scene.control.Button;

import javafx.scene.control.Label;

import javafx.scene.control.TextField;

import javafx.scene.layout.HBox;

import javafx.stage.Stage;

public class UserInterfaceExample extends Application {

public static void main(String[] args) {

launch(args);

}

@Override

public void start(Stage primaryStage) {

primaryStage.setTitle("User Interface Example");

// Кнопка

Button btn = new Button("Click me!");

// Метка

Label label = new Label("Enter your name:");

// Текстовое поле

TextField textField = new TextField();

// Горизонтальный контейнер для элементов

HBox hbox = new HBox(label, textField, btn);

primaryStage.setScene(new Scene(hbox, 300, 100));

primaryStage.show();

}

}

**Стилизация элементов:**

Вы можете стилизовать элементы управления с использованием CSS. JavaFX поддерживает стилизацию, которая позволяет определять внешний вид вашего интерфейса. Пример применения стилей:

import javafx.application.Application;

import javafx.scene.Scene;

import javafx.scene.control.Button;

import javafx.scene.layout.StackPane;

import javafx.stage.Stage;

public class StylingExample extends Application {

public static void main(String[] args) {

launch(args);

}

@Override

public void start(Stage primaryStage) {

primaryStage.setTitle("Styling Example");

Button btn = new Button("Click me!");

btn.getStyleClass().add("my-button");

StackPane root = new StackPane(btn);

Scene scene = new Scene(root, 300, 250);

// Подключение файла стилей

scene.getStylesheets().add("styles.css");

primaryStage.setScene(scene);

primaryStage.show();

}

}

**9.4** **Макеты и расположение элементов**

В JavaFX для расположения элементов интерфейса используются макеты (layout). Макеты определяют, как элементы будут размещены внутри родительского контейнера. Ниже приведены некоторые из основных макетов и их использование.

**1) VBox (Vertical Box):**

VBox располагает дочерние элементы вертикально, сверху вниз.

import javafx.application.Application;

import javafx.scene.Scene;

import javafx.scene.control.Button;

import javafx.scene.layout.VBox;

import javafx.stage.Stage;

public class VBoxExample extends Application {

public static void main(String[] args) {

launch(args);

}

@Override

public void start(Stage primaryStage) {

primaryStage.setTitle("VBox Example");

Button button1 = new Button("Button 1");

Button button2 = new Button("Button 2");

Button button3 = new Button("Button 3");

VBox vbox = new VBox(button1, button2, button3);

primaryStage.setScene(new Scene(vbox, 300, 200));

primaryStage.show();

}

}

**2) HBox (Horizontal Box):**

HBox располагает дочерние элементы горизонтально, слева направо.

import javafx.application.Application;

import javafx.scene.Scene;

import javafx.scene.control.Button;

import javafx.scene.layout.HBox;

import javafx.stage.Stage;

public class HBoxExample extends Application {

public static void main(String[] args) {

launch(args);

}

@Override

public void start(Stage primaryStage) {

primaryStage.setTitle("HBox Example");

Button button1 = new Button("Button 1");

Button button2 = new Button("Button 2");

Button button3 = new Button("Button 3");

HBox hbox = new HBox(button1, button2, button3);

primaryStage.setScene(new Scene(hbox, 300, 200));

primaryStage.show();

}

}

**3) BorderPane:**

BorderPane размещает дочерние элементы в пределах пяти областей: верхней, нижней, левой, правой и центральной.

import javafx.application.Application;

import javafx.scene.Scene;

import javafx.scene.control.Button;

import javafx.scene.layout.BorderPane;

import javafx.stage.Stage;

public class BorderPaneExample extends Application {

public static void main(String[] args) {

launch(args);

}

@Override

public void start(Stage primaryStage) {

primaryStage.setTitle("BorderPane Example");

Button topButton = new Button("Top");

Button bottomButton = new Button("Bottom");

Button leftButton = new Button("Left");

Button rightButton = new Button("Right");

Button centerButton = new Button("Center");

BorderPane borderPane = new BorderPane();

borderPane.setTop(topButton);

borderPane.setBottom(bottomButton);

borderPane.setLeft(leftButton);

borderPane.setRight(rightButton);

borderPane.setCenter(centerButton);

primaryStage.setScene(new Scene(borderPane, 300, 200));

primaryStage.show();

}

}

**4) GridPane:**

GridPane располагает дочерние элементы в сетке, используя строки и столбцы.

import javafx.application.Application;

import javafx.scene.Scene;

import javafx.scene.control.Button;

import javafx.scene.layout.GridPane;

import javafx.stage.Stage;

public class GridPaneExample extends Application {

public static void main(String[] args) {

launch(args);

}

@Override

public void start(Stage primaryStage) {

primaryStage.setTitle("GridPane Example");

Button button1 = new Button("Button 1");

Button button2 = new Button("Button 2");

Button button3 = new Button("Button 3");

GridPane gridPane = new GridPane();

gridPane.add(button1, 0, 0); // (row, column)

gridPane.add(button2, 1, 0);

gridPane.add(button3, 0, 1);

primaryStage.setScene(new Scene(gridPane, 300, 200));

primaryStage.show();

}

}

**9.5** **Графика и анимация**

В JavaFX графика и анимации предоставляют мощные инструменты для создания интересных и визуально привлекательных пользовательских интерфейсов. Вот обзор основных компонентов для работы с графикой и анимациями в JavaFX:

**1) Графика:**

**Canvas:**

Canvas представляет собой область, на которой можно рисовать различные фигуры и изображения.

import javafx.application.Application;

import javafx.scene.Scene;

import javafx.scene.canvas.Canvas;

import javafx.scene.canvas.GraphicsContext;

import javafx.scene.paint.Color;

import javafx.stage.Stage;

public class CanvasExample extends Application {

public static void main(String[] args) {

launch(args);

}

@Override

public void start(Stage primaryStage) {

primaryStage.setTitle("Canvas Example");

Canvas canvas = new Canvas(300, 200);

GraphicsContext gc = canvas.getGraphicsContext2D();

// Рисование красного круга

gc.setFill(Color.RED);

gc.fillOval(50, 50, 100, 100);

primaryStage.setScene(new Scene(canvas, 300, 200));

primaryStage.show();

}

}

**2) Фигуры и цвета:**

JavaFX предоставляет различные классы для рисования фигур, таких как Rectangle, Circle, Line и др. Вы можете также настраивать цвета, ширины линий и другие параметры.

import javafx.application.Application;

import javafx.scene.Scene;

import javafx.scene.paint.Color;

import javafx.scene.shape.Circle;

import javafx.stage.Stage;

public class ShapesExample extends Application {

public static void main(String[] args) {

launch(args);

}

@Override

public void start(Stage primaryStage) {

primaryStage.setTitle("Shapes Example");

Circle circle = new Circle(100, 100, 50);

circle.setFill(Color.BLUE);

circle.setStroke(Color.BLACK);

circle.setStrokeWidth(2);

primaryStage.setScene(new Scene(circle, 200, 200));

primaryStage.show();

}

}

**3) Анимации:**

**TranslateTransition:**

TranslateTransition позволяет создавать анимации перемещения элементов.

import javafx.animation.TranslateTransition;

import javafx.application.Application;

import javafx.scene.Scene;

import javafx.scene.layout.StackPane;

import javafx.scene.paint.Color;

import javafx.scene.shape.Rectangle;

import javafx.stage.Stage;

import javafx.util.Duration;

public class TranslateTransitionExample extends Application {

public static void main(String[] args) {

launch(args);

}

@Override

public void start(Stage primaryStage) {

primaryStage.setTitle("Translate Transition Example");

Rectangle rectangle = new Rectangle(50, 50, Color.BLUE);

TranslateTransition transition = new TranslateTransition(Duration.seconds(2), rectangle);

transition.setToX(200);

StackPane stackPane = new StackPane(rectangle);

stackPane.setPrefSize(300, 200);

primaryStage.setScene(new Scene(stackPane));

primaryStage.show();

// Запуск анимации

transition.play();

}

}

**RotateTransition:**

RotateTransition создает анимации вращения.

import javafx.animation.RotateTransition;

import javafx.application.Application;

import javafx.scene.Scene;

import javafx.scene.layout.StackPane;

import javafx.scene.paint.Color;

import javafx.scene.shape.Rectangle;

import javafx.stage.Stage;

import javafx.util.Duration;

public class RotateTransitionExample extends Application {

public static void main(String[] args) {

launch(args);

}

@Override

public void start(Stage primaryStage) {

primaryStage.setTitle("Rotate Transition Example");

Rectangle rectangle = new Rectangle(50, 50, Color.BLUE);

RotateTransition transition = new RotateTransition(Duration.seconds(2), rectangle);

transition.setByAngle(360);

StackPane stackPane = new StackPane(rectangle);

stackPane.setPrefSize(300, 200);

primaryStage.setScene(new Scene(stackPane));

primaryStage.show();

// Запуск анимации

transition.play();

}

}

**9.6** **Progressbar, menubar**

ProgressBar в JavaFX представляет собой графический элемент, который отображает ход выполнения какой-либо операции. Вот пример использования **ProgressBar:**

import javafx.application.Application;

import javafx.scene.Scene;

import javafx.scene.control.ProgressBar;

import javafx.scene.layout.StackPane;

import javafx.stage.Stage;

public class ProgressBarExample extends Application {

public static void main(String[] args) {

launch(args);

}

@Override

public void start(Stage primaryStage) {

primaryStage.setTitle("ProgressBar Example");

ProgressBar progressBar = new ProgressBar();

progressBar.setProgress(0.5); // Установка текущего значения

StackPane stackPane = new StackPane(progressBar);

stackPane.setPrefSize(300, 200);

primaryStage.setScene(new Scene(stackPane));

primaryStage.show();

}

}

**MenuBar в JavaFX:**

MenuBar представляет собой полосу меню в верхней части окна приложения. Каждый элемент MenuBar может содержать подменю с дополнительными пунктами меню. Вот пример использования MenuBar:

import javafx.application.Application;

import javafx.scene.Scene;

import javafx.scene.control.Menu;

import javafx.scene.control.MenuBar;

import javafx.scene.control.MenuItem;

import javafx.scene.layout.VBox;

import javafx.stage.Stage;

public class MenuBarExample extends Application {

public static void main(String[] args) {

launch(args);

}

@Override

public void start(Stage primaryStage) {

primaryStage.setTitle("MenuBar Example");

Menu fileMenu = new Menu("File");

MenuItem newItem = new MenuItem("New");

MenuItem openItem = new MenuItem("Open");

MenuItem exitItem = new MenuItem("Exit");

fileMenu.getItems().addAll(newItem, openItem, exitItem);

Menu editMenu = new Menu("Edit");

MenuItem cutItem = new MenuItem("Cut");

MenuItem copyItem = new MenuItem("Copy");

MenuItem pasteItem = new MenuItem("Paste");

editMenu.getItems().addAll(cutItem, copyItem, pasteItem);

MenuBar menuBar = new MenuBar(fileMenu, editMenu);

VBox vbox = new VBox(menuBar);

vbox.setPrefSize(300, 200);

primaryStage.setScene(new Scene(vbox));

primaryStage.show();

}

}

**10. Работа с базами данных (to be continue...)**