

С. Г. Сурто

КРОСС- ПЛАТФОРМЕННОЕ ПРОГРАММИРОВАНИЕ

Электронный учебно-методический
комплекс для студентов
специальности 6-05-0611-05
«Компьютерная инженерия»

Новополоцк / 2026

Министерство образования Республики Беларусь
Учреждение образования
«Полоцкий государственный университет имени Евфросинии Полоцкой»

С. Г. Сурто

КРОССПЛАТФОРМЕННОЕ ПРОГРАММИРОВАНИЕ

Электронный учебно-методический комплекс
для студентов специальности 6-05-0611-05
«Компьютерная инженерия»

Текстовое электронное издание

Новополоцк
Полоцкий государственный университет
имени Евфросинии Полоцкой
2026

Об издании – [1](#), [2](#)

1 – дополнительный титульный экран – сведения об издании

УДК 004.438(075.8)(0.034)

Одобрено и рекомендовано к изданию
советом факультета информационных технологий (протокол № 6 от 30.06.2025)

Кафедра вычислительных систем и сетей

РЕЦЕНЗЕНТЫ:

кафедра «Информационные технологии»

Гомельского государственного технического университета им. П. О. Сухого
(зав. каф. канд. техн. наук, доц. К. С. КУРОЧКА);

канд. техн. наук, доц., гл. инженер-программист ООО «СофтКлуб»

Д. О. ГЛУХОВ

Сурто, С. Г.

Кроссплатформенное программирование : электрон. учеб.-метод.
комплекс / С. Г. Сурто. – Новополоцк: Полоц. гос. ун-т им. Евфросинии
Полоцкой, 2026. – URL: <https://elib.psu.by/handle/123456789/50449>. –
Текст электронный.

ISBN 978-985-531-949-9.

Содержит лекционный материал, задания для самостоятельной работы и практических занятий. ЭУМК позволяет обеспечить целостное понимание изучаемой области знаний, получить практические навыки объективной оценки результатов работы субъектов хозяйствования и использовать их для принятия обоснованных управленческих решений.

Предназначен для студентов специальности 6-05-0611-05 «Компьютерная инженерия».

№ госрегистрации
ISBN 978-985-531-949-9

© Сурто С. Г., 2026
© Полоцкий государственный университет
имени Евфросинии Полоцкой, 2026

2 – дополнительный титульный экран – производственно-технические сведения

Для создания текстового электронного издания «Кроссплатформенное программирование» С. Г. Сурто использованы текстовый процессор Microsoft Word и программа Adobe Acrobat XI Pro для создания и просмотра электронных публикаций в формате PDF.

Электронный учебно-методический комплекс включен в Государственный регистр информационного ресурса. Регистрационное свидетельство

Учебное издание

СУРТО Сергей Геннадьевич

КРОССАТФОРМЕННОЕ ПРОГРАММИРОВАНИЕ

Электронный учебно-методический комплекс
для студентов специальности 6-05-0611-05
«Компьютерная инженерия»

Редактор *Т. А. Дарьянова*
Компьютерный дизайн: *отдел по связям с общественностью*

Подписано к использованию 17.08.2026.
Объем издания 3,1 Мб. Заказ 267.

Издатель и полиграфическое исполнение:
учреждение образования «Полоцкий государственный университет
имени Евфросинии Полоцкой».

Свидетельство о государственной регистрации
издателя, изготовителя, распространителя печатных изданий
№ 1/305 от 22.04.2014., перерегистрация от 24.08.2022.

ЛП № 02330/278 от 27.05.2004.

211440, ул. Блохина, 29,
г. Новополоцк,
Тел. 8 (0214) 59-95-41, 59-95-44

Содержание

ВВЕДЕНИЕ	5
ЛЕКЦИОННЫЙ КУРС.....	7
РАЗДЕЛ 1	
ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ JAVA.....	7
ТЕМА 1.1 КРОССПЛАТФОРМЕННОЕ ПРОГРАММИРОВАНИЕ: ОБЩИЕ ПОНЯТИЯ	7
ТЕМА 1.2 ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ JAVA	14
РАЗДЕЛ 2	
ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ В ЯЗЫКЕ JAVA	26
ТЕМА 2.1 ОБЪЯВЛЕНИЕ КЛАССА И АТТРИБУТОВ КЛАССА	26
ТЕМА 2.2 МЕТОДЫ КЛАССА, КОНСТРУКТОРЫ	27
ТЕМА 2.3 АБСТРАКТНЫЕ КЛАССЫ, ИНТЕРФЕЙСЫ.....	29
РАЗДЕЛ 3	
СПЕЦИАЛЬНЫЕ ВОЗМОЖНОСТИ ЯЗЫКА JAVA	39
ТЕМА 3.1 ШАБЛОНЫ В ЯЗЫКЕ JAVA	39
ТЕМА 3.2 ОБРАБОТКА ИСКЛЮЧЕНИЙ В ЯЗЫКЕ JAVA	48
ТЕМА 3.3 БАЗОВЫЕ ОБЪЕКТЫ ЯЗЫКА JAVA. КЛАССЫ-ОБЕРТКИ. СТРОКИ.....	55
ТЕМА 3.4 СБОРКА МУСОРА. ОСНОВНЫЕ ПОНЯТИЯ. ОСОБЕННОСТИ НАСТРОЙКИ И РЕАЛИЗАЦИИ В ЯЗЫКЕ JAVA	73
ТЕМА 3.5 КОЛЛЕКЦИИ ЯЗЫКА JAVA, ОСНОВНЫЕ ИНТЕРФЕЙСЫ	80
ТЕМА 3.6 ОСНОВНЫЕ РЕАЛИЗАЦИИ КОЛЛЕКЦИЙ ЯЗЫКА JAVA.....	88
СПИСОК (LIST)	88
ТЕМА 3.7 ПОТОКИ ВВОДА-ВЫВОДА В ЯЗЫКЕ JAVA. РАБОТА С ФАЙЛАМИ	91
ТЕМА 3.8 МНОГОПОТОЧНОЕ ПРОГРАММИРОВАНИЕ В ЯЗЫКЕ JAVA, ОСНОВНЫЕ ПОНЯТИЯ. СОЗДАНИЕ ПРОЦЕССОВ И ПОТОКОВ.....	98
ТЕМА 3.9 УПРАВЛЕНИЕ ПОТОКАМИ ПРИ МНОГОПОТОЧНОМ ПРОГРАММИРОВАНИИ В ЯЗЫКЕ JAVA. СИНХРОНИЗАЦИЯ	104
ТЕМА 3.10 СЕТЕВОЕ ВЗАИМОДЕЙСТВИЕ.....	117
ТЕМА 3.11 ВВЕДЕНИЕ В LAMBDA -ВЫРАЖЕНИЯ.....	125
ЛАБОРАТОРНЫЙ ПРАКТИКУМ.....	133
ЛАБОРАТОРНАЯ РАБОТА 1 КЛАССЫ И ОБЪЕКТЫ В JAVA	133
ЛАБОРАТОРНАЯ РАБОТА 2 НАСЛЕДОВАНИЕ.....	143
ЛАБОРАТОРНАЯ РАБОТА 3 СЕТЕВОЕ ПРОГРАММИРОВАНИЕ С СОКЕТАМИ И КАНАЛАМИ.....	154
ЛАБОРАТОРНАЯ РАБОТА 4 МНОГОПОТОЧНОЕ ПРОГРАММИРОВАНИЕ, БАЗОВЫЕ ПОНЯТИЯ ..	161
ЛАБОРАТОРНАЯ РАБОТА 5 МНОГОПОТОЧНОЕ ПРОГРАММИРОВАНИЕ. СИНХРОНИЗАЦИЯ ПОТОКОВ	167
ЛАБОРАТОРНАЯ РАБОТА 6 МНОГОПОТОЧНОЕ ПРОГРАММИРОВАНИЕ.....	171
СИНХРОНИЗАЦИЯ РЕЗУЛЬТАТОВ РАБОТЫ НИТЕЙ	171
ЛАБОРАТОРНАЯ РАБОТА 7 РАБОТА С ФАЙЛАМИ	176
ЛАБОРАТОРНАЯ РАБОТА 8 ИСКЛЮЧЕНИЯ В JAVA.....	182
ПЕРЕЧЕНЬ ВОПРОСОВ К ЗАЧЕТУ.....	189
КОНТРОЛЬ КАЧЕСТВА УСВОЕНИЯ ЗНАНИЙ	190

Введение

Учебно-методический комплекс «Кроссплатформенное программирование» разработан на основе базового учебного плана (регистрационный номер 04-21 / уч. ФИТ от 27.04.2021) для студентов специальности 6-05-0611-05 «Компьютерная инженерия».

Целью учебной дисциплины является изучение современных технологий программирования для различных архитектур и платформ.

Достижение поставленной цели предполагает решение следующих **задач**:

- сформировать системное базовое представление, первичные знания, умения и навыки по основам кроссплатформенного программирования для платформы Java;
- изучить этапы создания приложений в интегрированных средах разработки;
- показать основные характеристики исполняемого кода на различных платформах.

В результате изучения дисциплины формируются следующие **компетенции**:
специализированные:

СК-13. Разрабатывать системное и прикладное программное обеспечение, используя кроссплатформенный подход.

В результате изучения дисциплины студент должен

знать:

- архитектуру и компоненты кроссплатформенных решений;
- особенности программирования кроссплатформенных решений;
- многопоточное программирование на основе кроссплатформенного языка;
- концепцию и особенности применения функционального программирования;
- особенности программирования на функциональном языке;

уметь:

- разрабатывать системное и прикладное программное обеспечение, используя кроссплатформенный подход;
- применять функциональный подход при разработке программного обеспечения;

владеть:

- инструментарием разработки кроссплатформенных приложений.

Учебная дисциплина включена в цикл дисциплин, входящих в компонент учреждения высшего образования. Данная учебная дисциплина основывается на знаниях, полученных в результате изучения учебной дисциплины «Основы алгоритмизации и программирования». В свою очередь она необходима для усвоения таких учебных дисциплин, как «Программирование на языке высокого уровня», «Базы данных».

В соответствии с учебным планом изучение учебной дисциплины рассчитывается следующим образом:

Форма получения высшего образования	дневная
Курс	2
Семестр	3
Всего часов по учебной дисциплине	108
Аудиторных часов по учебной дисциплине	64
Лекции, часов	32
Лабораторные занятия, ч	32
Самостоятельная работа студентов	44
Трудоемкость дисциплины, зачет. ед.	3
Форма текущей аттестации	зачет

ЛЕКЦИОННЫЙ КУРС

Раздел 1

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ JAVA

Тема 1.1

КРОССПЛАТФОРМЕННОЕ ПРОГРАММИРОВАНИЕ: ОБЩИЕ ПОНЯТИЯ

Кроссплатформенное программирование – это подход к разработке программного обеспечения, при котором приложение может работать на различных платформах без значительных изменений в коде. Это позволяет охватить более широкую аудиторию пользователей, т.к. программа не привязана к конкретной операционной системе или аппаратному обеспечению.

Что такое платформа? Платформа – это среда, в которой выполняется программное обеспечение. Она может определяться:

- типом процессора (ЦП): например, x86, ARM;
- операционной системой (ОС): например, Windows, Linux, macOS, Android, iOS;
- комбинацией аппаратного и программного обеспечения: например, Windows на компьютерах с процессорами Intel.

Примеры платформ:

- настольные компьютеры: Windows, Linux/Unix, macOS;
- мобильные устройства: Android, iOS.

Зависимость от платформы. Обычное программное обеспечение часто разрабатывается с учетом особенностей конкретной платформы, будь то аппаратное обеспечение, операционная система или виртуальная машина.

Java как пример кроссплатформенной среды. Java использует виртуальную машину (JVM), которая работает на разных ОС и аппаратных платформах. Это позволяет Java-приложениям быть кроссплатформенными.

Типы платформ:

- аппаратные платформы: Определяются архитектурой набора команд процессора (например, x86, ARM). Разные операционные системы могут работать на одной и той же аппаратной платформе;
- программные платформы: Обычно представляют собой комбинацию операционной системы и среды программирования. Исключением является Java, которая использует виртуальную машину, независимую от операционной системы.

Примеры программных платформ:

- BlackBerry;
- Android для смартфонов и планшетных компьютеров (x86, ARM);
- iOS (ARM);
- Microsoft Windows (x86, ARM);
- Интерфейс командной строки, также известный под именами реализации .NET Framework (от Microsoft) и кроссплатформенным вариантом Mono (ранее Novell, а теперь Xamarin);
- Java;
- Веб-браузеры – более или менее совместимы друг с другом, работают с веб-приложениями на JavaScript.

Кроссплатформенное программирование – это разработка программ, которые могут функционировать на различных операционных системах и аппаратных платформах.

Кроссплатформенная разработка: стратегии и инструменты

Создание приложений, работающих на различных операционных системах (Windows, macOS, Linux, Android, iOS и др.), требует применения специальных подходов. Существуют различные стратегии кроссплатформенной разработки:

- **раздельная разработка (нативная разработка):** Этот подход предполагает создание отдельных версий приложения для каждой целевой платформы. Каждая версия разрабатывается с использованием нативных инструментов и языков программирования, специфичных для данной платформы. Хотя это обеспечивает максимальную производительность и доступ к полному спектру возможностей платформы, разработка и поддержка становятся более трудоемкими и дорогими, особенно при большом количестве поддерживаемых платформ. Возникают сложности с синхронизацией функциональности и исправлением ошибок, так как каждая версия может иметь свои уникальные проблемы;
- **абстракция платформы:** Этот метод использует промежуточное программное обеспечение (например, виртуальную машину), которое скрывает различия между операционными системами. Приложение разрабатывается для этой абстрактной среды и, следовательно, может работать на любой платформе, поддерживающей эту среду. Примером является Java Virtual Machine (JVM), позволяющая запускать Java-приложения на различных платформах;
- **гибридный подход:** Этот подход комбинирует элементы раздельной разработки и абстракции платформы. Например, приложение может использовать абстракцию для основных компонентов, но при этом иметь отдельные модули,

специфичные для каждой платформы (например, для работы с графическим интерфейсом). Также могут использоваться различные языки сценариев для упрощения переноса кода. Браузер Firefox является примером гибридного подхода, используя XUL, CSS и JavaScript для большей части функциональности, а также нативные компоненты для специфичных для платформы задач.

Инструменты для кроссплатформенной разработки

Существует множество инструментов, облегчающих процесс создания кроссплатформенных приложений. Вот некоторые примеры:

- 8th: Кроссплатформенный язык программирования, использующий Juce для графического интерфейса. Поддерживает Android, iOS, Windows, macOS, Linux и Raspberry Pi;

- Anant Computing: Платформа для мобильных приложений, поддерживающая все индийские языки, включая клавиатуры, а также AppWallet и нативные функции на различных операционных системах;

- ApperIQ: Платформа для разработки и развертывания корпоративных приложений. Использует собственные контейнеры для доступа к аппаратным функциям мобильных устройств через API для HTML5, упрощая создание кроссплатформенных мобильных приложений;

- Boden: Кроссплатформенная среда для создания пользовательских интерфейсов, написанная на C++;

- Cairo: Бесплатная библиотека для векторной графики, предоставляющая независимый от устройства API. Поддерживает двухмерное рисование в различных приложениях. Написана на C и имеет привязки для многих языков программирования;

- Cocos2d: Набор инструментов и игровой движок с открытым исходным кодом для разработки 2D и простых 3D кроссплатформенных игр и приложений;

- Codename One: Кроссплатформенная платформа с открытым исходным кодом, реализующая принцип “Write Once, Run Anywhere” (WORA) для разработчиков Java и Kotlin;

- Delphi: Это интегрированная среда разработки (IDE), позволяющая создавать приложения для разных операционных систем, таких как Android, iOS, Windows и macOS. В основе Delphi лежит язык программирования Pascal;

- Ecere SDK: Это кроссплатформенный набор инструментов для разработки графических интерфейсов, 2D/3D графики и IDE. Он написан на языке eC, но также поддерживает C и Python. Ecere SDK позволяет создавать приложения для Linux, FreeBSD, Windows, Android, macOS и веб-приложений (через Emscripten или Binaryen/WebAssembly);

- Eclipse: Это кроссплатформенная IDE с открытым исходным кодом, написанная на Java. Благодаря своей модульной архитектуре, Eclipse поддерживает множество инструментов для разработки программного обеспечения. Существуют плагины для работы с различными языками программирования, включая Java и C++;
- FLTK: Это кроссплатформенный и легковесный набор инструментов для создания графических интерфейсов;
- Flutter: Это UI-фреймворк, разработанный Google, который позволяет создавать кроссплатформенные приложения для Android и iOS;
- fpGUI: Это набор виджетов с открытым исходным кодом, полностью написанный на Object Pascal. Он позволяет разрабатывать приложения для Linux, Windows и Windows CE;
- GeneXus: Это платформа для быстрой разработки приложений, позволяющая создавать и развертывать кроссплатформенные приложения для различных платформ, включая Android, BlackBerry, iOS (Objective-C), C#, COBOL, Java, RPG, Ruby, Visual Basic и Visual FoxPro. GeneXus использует подход, основанный на представлении знаний;
- GLBasic: Это диалект языка BASIC и компилятор, который генерирует код на C++. Он включает в себя кросс-компиляторы для различных платформ, таких как Windows, Mac, Linux, Android, iOS и некоторые портативные устройства;
- GTK+: Это набор виджетов с открытым исходным кодом, предназначенный для Unix-подобных систем с X11 и Microsoft Windows;
- Нахе: это кроссплатформенный язык программирования с открытым исходным кодом;
- Juice: Это фреймворк для разработки приложений, написанный на C++. Он позволяет создавать нативные приложения для различных операционных систем (Microsoft Windows, POSIX, macOS) без необходимости изменения кода;
- Kivy: Это кроссплатформенный UI-фреймворк с открытым исходным кодом, написанный на Python. Он поддерживает разработку приложений для Android, iOS, GNU/Linux, OS X, Windows и Raspberry Pi;
- Lazarus: Это среда разработки, использующая компилятор FreePascal. Она позволяет создавать как графические, так и консольные приложения, работающие на Linux, MacOSX, iOS, Android, WinCE, Windows и в веб-браузере;
- Max/MSP: Визуальная среда программирования для создания мультимедийных приложений. Она позволяет разрабатывать код, который работает на разных операционных системах (macOS, Windows, Android), при этом часть кода адаптируется под конкретную платформу. Интересно, что приложения для Android могут запускаться на iOS и macOS без изменений;

- Mendix: Платформа для быстрой разработки приложений в облаке, требующая минимального написания кода;
- MonoCross: Архитектурный шаблон (MVC) для создания кроссплатформенных приложений. Основная логика (модель и контроллер) является общей для всех платформ, а пользовательский интерфейс (представление) разрабатывается отдельно для каждой платформы;
- Mono: Кроссплатформенная реализация .NET Framework, позволяющая запускать .NET-приложения на разных операционных системах;
- MoSync: Комплект разработки (SDK) с открытым исходным кодом для создания мобильных приложений на C++;
- Mozilla Application Framework: Платформа с открытым исходным кодом для разработки приложений, работающих на macOS, Windows и Linux;
- Кроссплатформенный JavaScript/TypeScript фреймворк: Инструмент для разработки приложений под Android и iOS с использованием JavaScript или TypeScript;
- OpenGL: Стандартизированная библиотека для работы с трехмерной графикой, поддерживаемая на разных операционных системах;
- PureBasic: Коммерческий язык программирования и среда разработки для создания приложений под macOS, Windows и Linux;
- ReNative: Универсальный SDK для создания мультиплатформенных приложений на базе React Native. Поддерживает широкий спектр платформ, включая мобильные, телевизионные, веб-платформы и операционные системы для носимых устройств;
- Qt: Фреймворк для разработки приложений с графическим интерфейсом и набором виджетов. Поддерживает Unix-подобные системы, Windows и macOS. Доступен как по открытой, так и по коммерческой лицензии;
- Simple and Fast Multimedia Library (SFML): C++ библиотека для работы с мультимедиа, предоставляющая доступ к графике, вводу, звуку и другим функциям на низком и высоком уровнях;
- Simple DirectMedia Layer (SDL): Кроссплатформенная библиотека на C, которая упрощает работу с графикой, звуком и устройствами ввода, абстрагируясь от особенностей разных операционных систем. Используется в основном для разработки игр и мультимедийных приложений;
- Smartface: Инструмент для разработки нативных мобильных приложений под Android и iOS с использованием визуального редактора (WYSIWYG) и JavaScript;

- Ultimate++ (U++): Среда быстрой разработки приложений на C++, ориентированная на повышение продуктивности программистов. Включает в себя набор библиотек и интегрированную среду разработки (TheIDE) с технологией BLITZ-build для ускорения компиляции. Поддерживает Windows и Unix-подобные операционные системы;
- Unity: Это кроссплатформенный инструмент разработки, основанный на игровом движке Unity, позволяющий создавать приложения для различных платформ;
- Платформа Uno: Используя C#, эта платформа позволяет разрабатывать приложения, работающие на Windows, macOS, iOS, Android, WebAssembly и Linux;
- Unreal Engine: Это кроссплатформенный SDK, использующий игровой движок Unreal Engine для создания приложений;
- V-Play Engine: Кроссплатформенный SDK для разработки, построенный на базе Qt. Разработка ведется в Qt Creator;
- WaveMaker: Инструмент разработки с низким кодом, предназначенный для быстрого создания адаптивных веб-приложений и гибридных мобильных приложений для Android и iOS;
- WinDev: Это интегрированная среда разработки (IDE) для создания бизнес-приложений для Windows, Linux, .Net и Java, а также веб-приложений. Отличается высокой скоростью разработки благодаря использованию готовых модулей, что сокращает объем необходимого кода. Использует процедурный язык программирования WLanguage с поддержкой императивного и объектно-ориентированного программирования;
- wxWidgets: Это кроссплатформенная библиотека виджетов с открытым исходным кодом и фреймворк для создания приложений. Приложения, разработанные с использованием wxWidgets, могут работать на Unix-подобных системах, Windows и macOS, если не используют специфические для операционной системы функции;
- Хоjo: Это среда быстрой разработки (RAD IDE) от Хоjo, Inc., использующая объектно-ориентированный язык программирования для создания приложений для настольных компьютеров, веб-приложений и iOS. Хоjo компилирует приложения для macOS, Windows, Linux и Raspberry Pi. Веб-приложения компилируются и могут запускаться как отдельные серверы или через CGI. Также поддерживается создание собственных приложений для iOS.

Проблемы кроссплатформенной разработки

Кроссплатформенная разработка, несмотря на свои преимущества, сопряжена с рядом сложностей:

- сложность тестирования: Обеспечение стабильной работы приложения на различных платформах требует тщательного тестирования, поскольку даже незначительные различия в операционных системах и аппаратном обеспечении могут приводить к непредсказуемым ошибкам. Это часто иронично описывают как «напиши один раз, отлаживай везде»;

- ограничение функциональности: Разработчики вынуждены ориентироваться на наименьший общий набор функций, доступных на всех целевых платформах. Это может негативно сказаться на производительности приложения и лишить его возможности использовать уникальные возможности каждой платформы;

- проблемы с пользовательским интерфейсом: Разные платформы имеют свои собственные стандарты и соглашения в отношении пользовательского интерфейса. Несоблюдение этих стандартов в кроссплатформенном приложении может привести к тому, что оно будет выглядеть чужеродным и неудобным для пользователей. В критических ситуациях, например, при работе с диалоговыми окнами, это может даже привести к потере данных;

- снижение производительности: Использование языков сценариев и виртуальных машин для обеспечения кроссплатформенности может приводить к снижению производительности, поскольку код должен транслироваться в машинный код каждой платформы во время выполнения. Хотя существуют методы оптимизации, такие как JIT-компиляция, избежать некоторой потери производительности полностью не удастся;

- разные форматы пакетов: Каждая платформа требует использования своего собственного формата пакетов для установки приложений (например, RPM, MSI). Для решения этой проблемы используются специализированные мультиплатформенные установщики;

- уязвимости безопасности: Кроссплатформенные среды выполнения могут содержать уязвимости, которые делают их привлекательной целью для создания межплатформенного вредоносного программного обеспечения.

Тема 1.2 ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ JAVA

Синтаксис языка Java

Приложение Hello, World!

```
public class HelloWorld {  
    // Print string  
    public static void main(String args[]) { System.out.println("Hello, World!");  
    }  
}
```

Листинг 1

```
/**  
 * Программа пишет "Hello, World"  
 * в стандартный поток вывода  
 * @version 2  
 */  
public class HelloWorld { private String name;  
  
    public static void main(String args[]) { HelloWorld h;  
        if (args.length > 0)  
            h = new HelloWorld(args[0]); else  
            h = new HelloWorld("World");  
    }  
  
    /**  
     * Конструктор создает объект с заданным именем  
     */  
    public HelloWorld(String s) {  
        name = s;  
    }  
  
    /**  
     * Метод выводит "Hello" и имя  
     * в стандартный поток вывода  
     */  
    public void sayHello() {  
        System.out.println("Hello, " + name + "!");  
    }  
}
```

1. Создайте текстовый файл с именем HelloWorld.java, содержащий приведенную программу (Листинг 1).
2. Скомпилируйте программу:
`javac HelloWorld.java`
3. Получившийся в результате файл HelloWorld.class запустите на выполнение:
`java HelloWorld`
4. Создайте документацию (для второго варианта программы)
`javadoc -version HelloWorld.java`

Лексический разбор исходного кода

1. Кодировка исходного кода Java.

Исходный код Java представляет собой последовательность символов Unicode, закодированных в UTF-16. Хотя Java изначально использует UTF-16, код может быть написан и в других кодировках, например, ASCII. В этом случае символы Unicode, отсутствующие в ASCII, представляются escape-последовательностями вида `\uXXXX`, где XXXX – шестнадцатеричное представление кода символа в UTF-16.

2. Этапы лексического анализа (токенизации).

Лексический анализатор (сканер) преобразует исходный код в последовательность лексем, выполняя следующие шаги:

- обработка Unicode escape-последовательностей: Все последовательности `\uXXXX` заменяются соответствующими символами Unicode;
- разбиение на строки: Исходный текст разделяется на строки. Разделителями строк служат символы перевода строки (LF), возврата каретки (CR) или их комбинация (CRLF);
- удаление комментариев: Удаляются как однострочные комментарии (от `//` до конца строки), так и многострочные комментарии (между `/*` и `*/`);
- выделение лексем: Текст разбивается на лексемы. Разделителями лексем служат пробельные символы (пробел, табуляция, перевод страницы) и концы строк.

3. Типы лексем.

Выделенные лексемы классифицируются как:

- идентификаторы;
- ключевые слова;
- литералы;
- разделители;
- операторы.

4. Идентификаторы и ключевые слова:

- идентификатор: Это последовательность букв и цифр, начинающаяся с буквы и не совпадающая ни с одним из ключевых слов языка. Идентификаторы используются для именования различных элементов программы.
- ключевые слова: Это зарезервированные слова языка Java, имеющие специальное значение. Литералы (например, true, false, null) также рассматриваются как ключевые слова в контексте идентификаторов, то есть не могут быть использованы в качестве имен. Список ключевых слов и литералов представлен в таблице 1.

Таблица 1

abstract	default	if	package	this
assert	do	goto	private	throw
boolean	double	implements	protected	throws
break	else	import	public	transient
byte	enum	instanceof	return	<u>true</u>
case	extends	int	short	try
catch	<u>false</u>	interface	static	void
char	final	long	strictfp	volatile
class	finally	native	super	while
const	float	new	switch	
continue	for	<u>null</u>	synchronized	

Типы данных и литералы

Java – это язык, где типы данных определяются строго и заранее, еще до запуска программы. Это значит, что каждая переменная и каждое выражение имеют чётко определенный тип, который известен компилятору. Java также строго следит за тем, чтобы переменной присваивалось значение только совместимого типа. Тип определяет, какие операции можно выполнять с переменной или выражением.

В Java существуют два основных вида типов данных: примитивные и ссылочные. Переменная примитивного типа хранит непосредственно само значение, а переменная ссылочного типа хранит указатель (ссылку) на место в памяти, где это значение находится.

В языке Java существует 8 примитивных типов данных (таблица 2).

Ссылочные типы данных включают в себя классы, интерфейсы, переменные типа (type variables) и массивы. Значения, присваиваемые переменным,

представляются в виде литералов. Числовые литералы подразделяются на два основных типа: целые числа и числа с плавающей точкой. Целые литералы могут иметь префикс и суффикс, а также само значение. Литералы с плавающей точкой состоят из целой части, дробной части, разделителя (точка), основания, порядка и, возможно, суффикса. Детали о допустимых форматах и обозначениях для каждого элемента числового литерала обычно приводятся в таблице 3.

Таблица 2

Тип	Бит	Диапазон значений	default
byte	8	[-128 ... 127]	0
short	16	[-32768 ... 32767]	0
int	32	[-2147483648 ... 2147483647]	0
long	64	[-9223372036854775808 ... 9223372036854775807]	0L
char	16	['\u0000' ... '\uffff'] или [0 ... 65535]	'\u0000'
float	32	$\pm[0; 1.4 \cdot 10^{-45} \dots 3.4028235 \cdot 10^{38}; \infty]$, NaN	0.0F
double	64	$\pm[0; 4.9 \cdot 10^{-324} \dots 1.7976931348623157 \cdot 10^{308}; \infty]$, NaN	0.0
boolean		false; true	false

Таблица 3

Элемент	Символ		Значение
префикс	0		восьмеричный литерал (целый)
	0x	0X	шестнадцатеричный литерал
	0b	0B	двоичный литерал (целый)
суффикс	l	L	литерал типа long
	f	F	литерал типа float
	d	D	литерал типа double
основание	e	E	десятичное основание степени
	p	P	двоичное основание степени

Числовые литералы (целые и с плавающей точкой) определяются по следующим правилам:

- Система счисления: Определяется префиксом. Отсутствие префикса означает десятичную систему. Другие системы: восьмеричная, шестнадцатеричная и двоичная, каждая со своим префиксом.

- Тип данных: Определяется суффиксом. Если суффикс отсутствует, литерал по умолчанию имеет тип `int` (для целых) или `double` (для чисел с плавающей точкой).
- Регистр: Буквы в литералах (префиксы, суффиксы, цифры в шестнадцатеричной системе) могут быть как строчными, так и заглавными.
- Разделители: Для улучшения читаемости длинные числовые литералы можно разделять символом подчеркивания (`_`).
- Тип `long`: Не рекомендуется использовать строчную букву `l` в качестве суффикса для типа `long`, т.к. ее легко спутать с цифрой 1.

Числа с плавающей точкой:

- десятичный формат: `aEb` представляет число $a * 10^b$;
- шестнадцатеричный формат: `0xPb` представляет число $a_{16} * 2^b$;
- обязательные элементы:
 - минимум одна цифра в целой или дробной части;
 - десятичная точка или показатель степени (с основанием) или суффикс типа для десятичного формата;
 - показатель степени (с основанием) для шестнадцатеричного формата.

Логические литералы:

- Имеют тип `boolean` и могут принимать значения `true` или `false`.

Символьные литералы:

- представляют собой один символ Unicode;
- имеют тип `char`;
- заключаются в одинарные кавычки;
- могут быть заданы:
 - явно (непосредственно символ);
 - с помощью Unicode-последовательности `\uXXXX`, где `XXXX` – шестнадцатеричный код символа;
 - с помощью восьмеричной последовательности `\XXX`, где `XXX` – восьмеричный код символа (от 000 до 377);
 - с помощью экранированных символов (например, `\n` для новой строки).

Строковые литералы:

- представляют собой последовательность символов Unicode;
- имеют тип `String` и являются ссылкой на объект класса `String`;
- заключаются в двойные кавычки;

– символы Unicode внутри строки могут быть заданы теми же способами, что и для символьных литералов (явно, \uXXXX, \XXX, экранированные символы).

Экранированные символы используются для представления специальных символов, которые иначе нарушили бы синтаксис программы (таблица 4).

Таблица 4

Экранированный символ	Значение
\b	символ возврата
\t	табуляция
\f	перевод страницы
\n	перевод строки
\r	возврат каретки
\"	двойная кавычка
\'	одинарная кавычка
\\	обратная косая черта

Неопределенный литерал null представляет неопределенное значение ссылочного типа — пустая ссылка. Примеры литералов приведены в таблице 5.

Таблица 5. – Примеры литералов

987_654_321	//	десятичный типа int
0765_4321_0123_4567L	//	восьмеричный типа long
0xFFFF_CAFE	//	шестнадцатеричный типа int
0b0001_0010_0100_1000L	//	двоичный типа long
1.	//	десятичный типа double = 1.0
1e0	//	десятичный типа double = 1.0
1d	//	десятичный типа double = 1.0
3.14159F	//	десятичный типа float = π
6.67408e-11f	//	десятичный типа float = 6.67408·10 ⁻¹¹
0xFFp4	//	шестнадцатеричный типа double
	//	FF16·24 = FF016
true	//	логический
'π'	//	символьный
'\u03c0'	//	символьный = 'π'
"Привет"	//	строковый

Переменные. Имя переменной должно быть уникальным и состоять из символов Unicode. Нельзя использовать ключевые слова Java, а также слова **true** и **false** в качестве имен переменных. В пределах одной части программы (области видимости) не может быть двух переменных с одинаковым именем.

Ключевые слова Java

Список ключевых слов представлен в таблице 6.

Таблица 6

abstract	continue	for	new	switch
assert	default	goto	package	synchronized
boolean	do	if	private	this
break	double	implements	protected	throw
byte	else	import	public	throws
case	enum	instanceof	return	transient
catch	extends	int	short	try
char	final	interface	static	void
class	finally	long	strictfp	volatile
const	float	native	super	while

Зарезервированные литералы — null, true и false.

Области видимости

Область видимости определяет, в какой части кода эта переменная может быть использована и к ней можно обратиться по ее имени. Другими словами, это зона, где переменная «видна» и доступна для работы (рисунок 1).

Как правило, **область видимости переменной ограничена блоком кода, в котором она объявлена**. Блок кода обычно заключен в фигурные скобки {}.

Более подробно, правила определения области видимости в Java:

- **import:** Импортированные классы и интерфейсы доступны во всех классах и интерфейсах, объявленных в том же файле.
- **Классы и интерфейсы верхнего уровня:** Область видимости класса или интерфейса, не вложенного в другой класс, распространяется на весь пакет, к которому он принадлежит.
- **Члены класса/интерфейса (поля, методы):** Члены класса (поля, методы) доступны во всем теле класса или интерфейса, включая вложенные типы (внутренние классы, интерфейсы).
- **Константы перечисления (enum):** Константы перечисления доступны во всем теле перечисления, а также в любом switch блоке, использующем это перечисление.
- **Параметры методов, конструкторов, циклов, лямбда-выражений:** Параметры, объявленные в методах, конструкторах, циклах (например, for), или лямбда-выражениях, доступны только внутри тела этого метода, конструктора, цикла или лямбда-выражения.

- **Параметры типа:** область видимости параметра типа элемента — все тело элемента, исключая модификаторы.
- **Локальные классы и переменные:** локальные классы и переменные, объявленные внутри блока кода, доступны только внутри этого блока, от точки объявления до конца блока.
- **Параметр catch блока:** параметр, объявленный в блоке catch для обработки исключений, доступен только внутри этого catch блока.
- **Переменные в try-with-resources:** переменные, объявленные в блоке try с ресурсами (try-with-resources), доступны во всем блоке try.

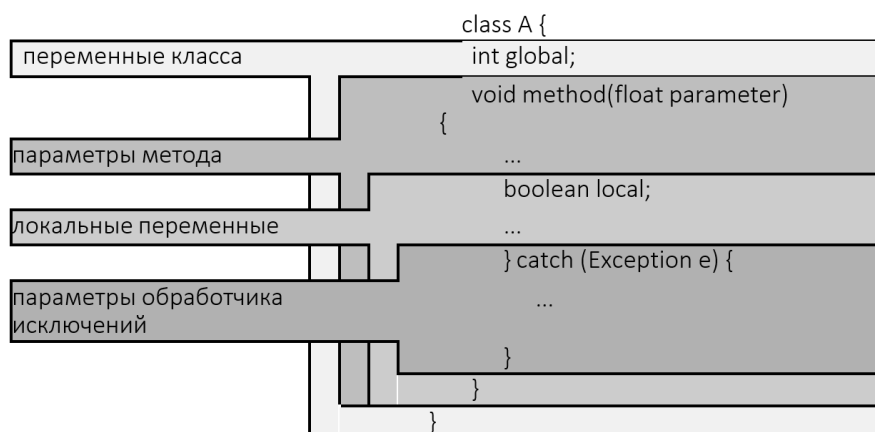


Рисунок 1. – Области видимости переменных

Операторы

Приоритеты операторов

Таблица 7

постфиксные операторы	<i>expr++ expr-- . ::</i>
унарные операторы	<i>++expr --expr +expr -expr ~ !</i>
операции с типами	<i>new (cast)expr</i>
умножение/деление	<i>* / %</i>
сложение/вычитание	<i>+ -</i>
операторы сдвига	<i><< >> >>></i>
операторы отношения	<i>< > <= >= instanceof</i>
операторы равенства	<i>== !=</i>
поразрядное И	<i>&</i>
поразрядное искл. ИЛИ	<i>^</i>
поразрядное ИЛИ	<i> </i>
логическое И	<i>&&</i>
логическое ИЛИ	<i> </i>
условный оператор	<i>? :</i>
операторы присваивания	<i>= += -= *= /= %= >>= <<= >>>= &= ^=</i> <i> =</i>
стрелка лямбда	<i>-></i>

Управляющие конструкции

Листинг 2

<pre>// ветвление if (boolean <i>expr</i>) { <i>op1</i>; } else { <i>op2</i>; } }</pre>	<pre>// цикл с параметром /* init – инициализация <i>expr</i> – условие продолжения <i>incr</i> – при- ращение параметра */ for (<i>init</i>; <i>expr</i>; <i>incr</i>) { <i>op</i>; }</pre>
<pre>// множественный выбор switch (int String enum <i>expr</i>) { case const1: <i>op1</i>; case const2: <i>op2</i>; break; default: <i>op3</i>; }</pre>	<pre>// цикл по элементам /* coll – любой тип, реализующий интерфейс Itera- ble<T> (например, коллекция) или массив, со- держащий элементы типа T */ for (T x : <i>coll</i>) { <i>op</i>; }</pre>
<pre>// цикл с предусловием while (boolean <i>expr</i>) { <i>op</i>; }</pre>	<pre>// цикл с постусловием do { <i>op</i>; } while (boolean <i>expr</i>);</pre>
<pre>// обработка исключения try { <i>op</i>; } catch (Exception e) { <i>block1</i>; } finally { <i>block2</i>; }</pre>	<pre>// метка для выхода из цикла label: for (...) { for (...) { <i>op</i>; continue label; } }</pre>
<pre>// генерация исключения throw new Exception();</pre>	<pre>// возврат из функции return;</pre>

Стоит обратить внимание на команду **switch**. Она претерпевала множество изменений (точнее дополнений, поскольку базовая версия работает, как и прежде) с выходом новых версий java. Наиболее значимые, пожалуй, связаны с введением возможности напрямую возвращать значение. Законченный, на текущий момент, вид этот оператор принял в версии 14. Пример:

```
String dayOfWeek = " Tuesday";
String greeting = switch (dayOfWeek) {
    case "Monday", "Tuesday", "Wednesday", "Thursday" -> {
        yield "Working day";
    }
    case "Friday" -> {
        yield "Almost weekend";
    }
    default -> {
        yield "Weekend";
    }
};
System.out.println(greeting);
```

В этом примере переменная **greeting** принимает значение, возвращаемое **switch**-ом. Из других отличий от базовой версии стоит отметить замену **break** на **yield**, который и возвращает значение и замену «:» после **case** на **->**.

Модификаторы

Таблица 8

abstract	метод только описан, но не реализован; класс содержит абстрактные методы
final	переменная является константой; метод не может быть переопределен; класс не может быть расширен
static	переменная является переменной класса; метод является методом класса; инициализатор запускается при загрузке класса; класс является классом верхнего уровня
public	метод или переменная доступны везде
protected	метод или переменная доступны в подклассах и в пределах пакета
private	метод или переменная доступны только в пределах класса
(package)	метод или переменная доступны только в пределах пакета
synchronized	метод, для которого осуществляется блокировка доступа к ресурсу
transient	переменная не должна быть сериализована
native	метод реализован на С или другим платформозависимым способом
volatile	переменная не должна оптимизироваться; переменная может быть изменена разными потоками, и компилятор должен гарантировать, что все потоки видят самое последнее значение этой переменной

Аннотации

Аннотации – это специальные пометки в коде, которые не меняют его поведение напрямую, но предоставляют дополнительную информацию. Эта информация может быть использована инструментами разработки, компилятором или самой программой во время работы.

Стандартные аннотации языка Java – таблица 9.

Таблица 9

Аннотация	Описание
@Deprecated	<pre>/* Этот метод устарел, заменён * на aBetterAlternative() и */ не рекомендуется к использованию. @Deprecated public void theDeprecatedMethod() { } public void aBetterAlternative() { }</pre>
@SuppressWarnings	<pre>public static void main(String[] args) { // Блокирует предупреждение // компилятора @SuppressWarnings("unchecked") Collection<Integer> c = new LinkedList(); }</pre>
@Override	<pre>class BaseClass { public void toBeOverriddenMethod() { } } public class ClassWithOverrideMethod { // Переопределяет метод // родительского класса @Override public void toBeOverriddenMethod() { } }</pre>

Перечисляемые типы

Перечисляемый тип (enum) – это тип, значения которого ограничены набором констант.

```

public enum Season {
    WINTER, SPRING, SUMMER, AUTUMN
}

public class TestSeason {
    public static void main(String[] args) {
        Season s = Season.SUMMER;
        System.out.println("Current season is " + s);
    }
}

```

Вывод в консоль в результате запуска программы:

```
Current season is SUMMER
```

Использование объектов

Таблица 10

Задача	Выполнение
объявление объекта (a = null)	Point a;
создание объекта (выделяется память)	a = new Point(10,10);
доступ к переменным	a.x = 20;
вызов методов	a.show();
уничтожение неиспользуемого объекта	"сборщик мусора"

Метод finalize()

```

protected void finalize() throws Throwable { super.finalize();
    if (file != null) { file.close();
        file = null;
    }
}

```

Раздел 2

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ В ЯЗЫКЕ JAVA

Тема 2.1

ОБЪЯВЛЕНИЕ КЛАССА И АТТРИБУТОВ КЛАССА

Класс – это прототип, описывающий переменные и методы, определяющие характеристики объектов данного класса.

Объект – это программная модель объектов реального мира или абстрактных понятий, представляющая собой совокупность переменных, задающих состояние объекта, и связанных с ними методов, определяющих поведение объекта.

Характеристики объектов (таблица 11)

Таблица 11

Класс	Состояние	Поведение
<i>собака</i>	кличка, возраст, порода,	лает, кусает, виляет хвостом,
	цвет	грызет новые тапочки
<i>автомобиль</i>	цвет, марка, скорость, передача	ускоряется, тормозит, меняет передачу
<i>Point</i>	x, y	show(), hide()
	Переменные	Методы

Создание объектов

Процесс создания новых объектов по описанию, объявленному в классе, называется созданием или реализацией (instantiation).

Объекты создаются с помощью инструкции **new**. При этом происходит выделение памяти для хранения переменных объекта.

Классы и объекты (таблицы 12, 13)

Таблица 12

Класс	Объявление переменных	Объект	Переменные
Car	color	myCar	color = red
	speed		speed = 90
	brand		brand = Volvo
Point	x	a	x = 40
	y		y = 25
	isVisible		isVisible = true

Таблица 13

Объявление класса	Реализация объекта
<pre>class Point { int x, y; boolean isVisible; }</pre>	<pre>Point a; // объявление объекта a = new Point(); // создание объекта a.x = 40; a.y = 25; a.isVisible = true;</pre>

Тема 2.2

МЕТОДЫ КЛАССА, КОНСТРУКТОРЫ

Конструкторы (таблица 14)

Для инициализации объекта при его создании используются конструкторы.

Таблица 14

Объявление класса	Создание объекта
<pre>class Point { int x, y; boolean isVisible = false; Point() { x = y = 0; } Point(x0, y0) { x = x0; y = y0; } }</pre>	<pre>Point a = new Point(); a.isVisible = true; Point b = new Point(20,50);</pre>

Значения переменных объектов (таблица 15)

Таблица 15

Point	<i>a</i>	<i>b</i>
int x	0	20
int y	0	50
boolean isVisible	True	false

Сообщения

Объекты взаимодействуют между собой путем отправки друг другу сообщений, которые могут содержать параметры. Отправка сообщения осуществляется с помощью вызова соответствующего метода объекта (таблица 16).

Таблица 16

Объявление класса	Вызов методов
<pre> class Point { int x, y; boolean isVisible; void hide() { isVisible = false; } void show() { isVisible = true; } void move(x1, y1) { x = x1; y = y1; } } </pre>	<pre> Point a = new Point(20,30); a.isVisible = true; a.hide(); a.move(60,80); a.show(); Point center = new Point(); center.show(); </pre>

Компоненты сообщения (таблица 17)

Таблица 17

Объект	Имя метода	Параметры
a.	move	(60,80)
center.	show	()

Инкапсуляция

Инкапсуляция – сокрытие данных внутри объекта, и обеспечение доступа к ним с помощью общедоступных методов (таблица 18).

Таблица 18

Объявление класса	Доступ к переменным класса
<pre> public class Point { private int x, y; public void move(x1,y1) { x = x1; y = y1; } public int getX() { return x; } } </pre>	<pre> Point a = new Point(20,30); int z; a.x = 25; // запрещено z = a.y; // запрещено z = a.getY(); a.move(25,a.getY()); a.move(a.getX()-5,a.getY()+5); </pre>

Окончание таблицы 18

Объявление класса	Доступ к переменным класса
<pre> } public int getY() { return y; } }</pre>	

Тема 2.3 АБСТРАКТНЫЕ КЛАССЫ. ИНТЕРФЕЙСЫ

Наследование

Наследование или расширение – приобретение одним классом (подклассом) свойств другого класса (суперкласса) (таблица 19).

Таблица 19

Объявление суперкласса	Наследование
<pre> public class Point { protected int x, y; public Point() { x = y = 0; } public Point(int x, int y){ this.x = x; this.y = y; } public int getX() { return x; } public int getY() { return y; } }</pre>	<pre> public class Pixel extends Point { private int color; public Pixel() { color = 0; } public Pixel(int x, int y){ super(x,y); color = 0; } public setColor(int c) { color = c; } public int getColor() { return color; } }</pre>

Пример использования приведен в таблице 20.

Таблица 20

<pre>Point a = new Point(30,40); int ax, ay; ax = a.getX(); ay = a.getY();</pre>	<pre>Pixel b = new Pixel(20,50); b.setColor(2); int bx, by, bc; bx = b.getX(); by = b.getY(); bc = b.getColor();</pre>
--	--

Полиморфизм

Полиморфизм – возможность единообразно обрабатывать объекты разных типов. Различают полиморфизм включения (или полиморфизм подтипов), который позволяет обращаться с помощью единого интерфейса к классу и к любым его потомкам, и параметрический полиморфизм, который определяет обобщенный интерфейс с одинаковой реализацией (таблица 21).

Таблица 21

Объявления классов	Использование полиморфизма
<pre>public class Shape { ... abstract void draw(); } public class Rectangle extends Shape { ... void draw() { ... } }</pre>	<pre>public class FiguresSet { ... public static void main (String args[]) { ... Shape rect; Shape circ; ... rect = new Rectangle(); circ = new Circle(); ... rect.draw(); circ.draw(); } }</pre>
<pre>public class Circle extends Shape { ... void draw() { ... } }</pre>	

Интерфейсы

Интерфейс – абстрактное описание набора методов и констант, необходимых для реализации определенной функции (таблица 22).

Таблица 22

Объявление интерфейса	Воплощение интерфейса
<pre>interface Displayable { void hide(); void show(); }</pre>	<pre>public class Pixel extends Point implements Displayable { ... void hide() { ... } void show() { ... } }</pre>

Класс может воплощать любое количество интерфейсов.

Листинг 3

```
/*  
 * До модификации  
 */  
public interface Orderable {  
    void orderByID();  
}  
// Класс скомпилируется  
public class MyCollection implements Orderable {  
    public void orderByID() { ...  
    }  
    ...  
}  
/*  
 * После модификации  
 */  
public interface Orderable {  
    void orderByID();  
    void orderByName(); // Новый метод  
}
```

```
// Класс не скомпилируется –  
// не реализован метод orderByName  
public class MyCollection implements Orderable {  
    ...  
}
```

В Java 7 и более ранних версиях интерфейсы могли содержать **только** заголовки методов и константы. Это приводило к проблемам в случае модификации интерфейса – при добавлении к нему нового метода все его реализации «ломались» (см. листинг 3) до тех пор, пока в них не будет добавлена реализация добавленного метода:

```
public void orderByName() {...  
}  
...
```

Эта проблема была решена в Java 8. Теперь, помимо заголовков методов и констант, интерфейсы могут содержать и конкретные реализации методов. Такие методы называются **методами по умолчанию**; при их объявлении должно быть использовано ключевое слово `default` (листинг 4).

Листинг 4

```
public interface Orderable {  
    void orderByID();  
    default void orderByName() {  
        ... // Какая-то реализация  
    }  
}  
  
// Класс скомпилируется –  
// метод orderByName уже реализован в интерфейсе  
public class MyCollection implements Orderable {  
    ...  
    public void orderByName() {  
        ...  
    }  
    ...  
}
```

Класс может реализовывать множество интерфейсов, поэтому может возникнуть ситуация, когда один и тот же метод объявлен или реализован сразу в нескольких из них. Если метод объявлен более, чем в одном из интерфейсов, то программист **обязан** переопределить его в классе независимо от того, реализован этот метод в интерфейсах или нет.

Таблица 23

Код	Комментарий
<pre>interface Human { default void getName(); } public class Businessman implements Human, Company { // Метод getName не реализован ... }</pre>	Код не скомпилируется – требуется реализовать метод getName() в классе
<pre>interface Human { default void getName() { ... // Реализация } } public class Businessman implements Human, Company { // Метод getName не реализован ... }</pre>	Код скомпилируется – метод уже реализован в интерфейсе; реализация в классе не нужна
<pre>interface Human { default void getName() { ... // Реализация } } interface Company { default void getName() { ... // Реализация } } public class Businessman implements Human, Company { // Метод getName не реализован ... }</pre>	Код не скомпилируется – «конфликт» между интерфейсами. Нужно переопределить метод в классе
<pre>interface Human { default void getName() { ... // Реализация } }</pre>	Код не скомпилируется – нужно переопределить метод getName() в классе, несмотря на то, что он уже реализован в интерфейсе Human

Окончание таблицы 23

<pre> } interface Company { void getName(); } public class Businessman implements Human, Company { // Метод getName не реализован ... } </pre>	
--	--

Если класс «получает по наследству» реализацию метода одновременно из родительского класса и из интерфейса, то всегда применяется реализация метода из класса (листинг 5).

Листинг 5

<pre> public interface Orderable { default void orderByName(){ ... // Реализация } } // Не реализует интерфейс Orderable public class OrderedCollection { // Сигнатура совпадает с методом в интерфейсе public void orderByName() { ... // Какой-то код } } // Класс скомпилируется – метод orderByName // будет унаследован от класса OrderedCollection public class MyCollection extends OrderedCollection implements Orderable { // Метод orderByName не реализован ... } </pre>
<pre> ... // Какой-то код } } </pre>

```
// Класс скомпилируется – метод orderByName
// будет унаследован от класса OrderedCollection
public class MyCollection extends OrderedCollection
                                implements Orderable {
    // Метод orderByName не реализован
    ...
}
```

Вложенные, локальные и анонимные классы

Вложенный класс – это класс, объявленный внутри объявления другого класса.

```
public class EnclosingClass {
    ...
    public class NestedClass {
        ...
    }
}
```

Локальный класс – это класс, объявленный внутри блока кода. Область видимости локального класса ограничена тем блоком, внутри которого он объявлен.

Анонимный класс – это локальный класс без имени (листинг 6).

Листинг 6

```
public class EnclosingClass {
    ...
    public void enclosingMethod(){
        // Этот класс доступен только внутри
        // enclosingMethod()
        public class LocalClass {
            ...
        }
    }
}
```


Листинг 7

```
// Параметр конструктора – экземпляр анонимного класса,  
// реализующего интерфейс Runnable  
new Thread(new Runnable() {  
    public void run() {  
        ...  
    }  
}).start();
```

Записи (Records)

Записи (Records) в Java — в окончательной ревизии появились в релизе Java 17. Записи помогают сокращать огромное количество шаблонного кода. Более того, благодаря записям не нужно писать ни одной строки кода, чтобы сделать класс неизменяемым.

Если вы хотите передавать неизменяемые данные между разными уровнями вашего приложения, то использование Record может стать хорошим выбором. По умолчанию записи (Records) в Java являются неизменяемыми, а это означает, что мы не можем изменить их свойства после их создания. В результате это помогает нам избежать ошибок и повышает надежность кода. Проще говоря, используя Record в Java, можно автоматически генерировать классы.

Запись (Record) в Java — это класс, предназначенный для хранения данных. Он похож на традиционный класс Java, но более легкий и обладает некоторыми уникальными функциями. Записи неизменяемы по умолчанию, что означает, что их состояние не может быть изменено после их создания. Это делает их идеальными для хранения неизменяемых данных, таких как параметры конфигурации или значения, возвращенные из запроса к базе данных.

Также это способ создания пользовательского типа данных, состоящего из набора полей или переменных, а также методов доступа к этим полям и их изменения. Record в Java упрощает работу с данными за счет уменьшения объема шаблонного кода, который разработчики используют для написания снова и снова.

Синтаксис создания записи в Java:

```
record Record_Name(Fields....)
```

Пример:

```
public record Book(String name, double price) { }
```

В этом примере мы создали запись под названием Book с двумя полями: name и price. Ключевое слово public указывает, что к этой записи можно получить доступ вне пределов пакета, в котором она определена.

В Java записи создаются так же, как и обычные классы, с использованием ключевого слова new. Например: `Book book = new Book("Book to sale", 22);`.

После создания экземпляра записи, доступ к ее полям осуществляется напрямую, используя имя поля как имя метода. В отличие от обычных классов, в записях нет методов get и set.

Например: `String name = book.name();` и `double price = book.price();`.

Записи автоматически предоставляют реализации методов `toString()`, `equals()` и `hashCode()`, что упрощает работу с ними.

Важно помнить, что записи в Java **неизменяемы**. Это значит, что после создания экземпляра записи, его значения полей нельзя изменить.

Что еще умеют записи?

- **Пользовательские конструкторы:** Записи поддерживают как обычные параметризованные конструкторы, которые могут вызывать конструктор по умолчанию, так и «компактные» конструкторы, которые позволяют добавлять дополнительную логику, например, проверки, непосредственно в конструктор.

- **Методы экземпляра:** Как и обычные классы, записи могут содержать методы экземпляра, позволяющие добавлять специфическую функциональность.

- **Статические поля и методы:** Хотя записи не позволяют определять переменные экземпляра вне списка параметров конструктора, они поддерживают статические поля и методы. Это позволяет хранить и использовать данные, общие для всех экземпляров записи.

- Запись в Java может реализовывать интерфейсы.

Sealed классы

Sealed (закрытые или запечатанные) классы были представлены в Java 15 в качестве способа ограничить иерархию наследования класса или интерфейса. Sealed класс или интерфейс ограничивает набор классов, которые могут наследовать или реализовывать его, что может помочь предотвратить потенциальные ошибки и сделать код более удобным в сопровождении.

Создание иерархии закрытого типа

Sealed классы могут создавать иерархию закрытого типа. Речь идет об ограниченном наборе классов, которые не могут быть расширены или реализованы вне определенного пакета. Это гарантирует, что могут быть использованы только классы из определенного набора, и предотвращает нежелательные расширения или реализации.

```
public sealed class Animal permits Cat, Dog {  
    // Определение класса  
}  
public final class Cat extends Animal {  
    // Определение класса  
}  
public final class Dog extends Animal {  
    // Определение класса  
}
```

В этом примере `Animal` является `sealed` классом, который позволяет расширять его только классам `Cat` и `Dog`. Любая другая попытка расширения `Animal` приведет к ошибке компиляции.

Создание ограниченного набора реализаций

`Sealed` классы также могут создавать ограниченный набор реализаций для определенного интерфейса или абстрактного класса. Это нужно, чтобы владельцы интерфейса или абстрактного класса могли контролировать и изменять набор реализаций.

```
public sealed interface Shape permits Circle, Square {  
    double getArea();  
}  
public final class Circle implements Shape {  
    // Определение класса  
}  
public final class Square implements Shape {  
    // Определение класса  
}
```

В этом примере `Shape` – это `sealed` интерфейс, который позволяет реализовать его только классам `Circle` и `Square`. Это гарантирует, что создать любую другую реализацию `Shape` будет невозможно.

Раздел 3 СПЕЦИАЛЬНЫЕ ВОЗМОЖНОСТИ ЯЗЫКА JAVA

Тема 3.1 ШАБЛОНЫ В ЯЗЫКЕ JAVA

Обобщения позволяют создавать структуры данных и алгоритмы, которые можно было бы использовать с различными типами, не создавая каждый раз специфичные версии тех же структур для каждого из типов в отдельности. Особенно важно это для абстрактных типов данных вроде стеков и очередей. Конечно, можно создать десяток версий стека для работы с целыми числами (IntStack), символами (CharStack) или числами с плавающей точкой (FloatStack). Но гораздо удобнее было бы реализовать некий общий интерфейс и пользоваться им, не завися от типа данных, с которым мы работаем.

Теоретически мы могли бы создать одну структуру данных, хранящую объекты, приведенные к типу Object, но в этом случае никто не может гарантировать что в этой структуре данных действительно будут храниться данные только нужно нам типа.

Шаблон класса

Параметры шаблона класса указываются в угловых скобках. Класс не может быть потомком класса Throwable. Если в C++ по шаблону класса создается именно новый класс, то в Java типы, созданные по одному шаблону, являются классом этого шаблона. Поэтому тип параметра нельзя использовать для определения статических членов или использовать в статических методах.

Листинг 8

```
public class TstTpl<T> {
    private T t;
    public TstTpl(T tt) {
        t = tt;
    }
    public void out() {
        System.out.println("tplclass="+getClass()+
            ", param class=" + t.getClass() + ", val="
            + t.toString());
    }
    public static void main(String[] args) {
        new TstTpl<Boolean>(true).out();
    }
}
```

```
// сгенерированные типы принадлежат одному классу
TstTpl<Integer> a = new TstTpl<Integer>(1);
TstTpl<Double> b = new TstTpl<Double>(3.14);
System.out.println(a.getClass() == b.getClass());
}
}
```

В угловых скобках указывается имя параметра типа. Затем вместо него будет подставлено имя реального типа. Т.е. повсюду параметр T будет заменен на тип, указанный в угловых скобках.

Метод out() вспомогательный, и в нем мы выводим на экран реальный тип хранимого объекта, используя такое свойство, как рефлексия. Рефлексия позволяет во время выполнения программы получать сведения о классе. В данном случае это имя класса. Классы, необходимые для использования рефлексии, хранятся в пакете java.lang.reflect.

Шаблон метода

Аналогичным образом в классе можно определить шаблонный метод (листинг 9).

Листинг 9

```
public class TstTplm {
    static <T1, T2> void outmerge(T1 t1, T2 t2) {
        System.out.println(t1.toString() + t2);
    }
    public static void main(String[] args) {
        outmerge(new Double(3.14), new Boolean(true));
    }
}
```

Обобщенные интерфейсы

Так же, как и классы интерфейсы могут использовать обобщенные параметры.

```
interface Playable <T>{
    public void addSong(T t);
    public T playSong();
}
```

Существует несколько вариантов того, как класс может реализовать этот интерфейс. Первый вариант – это конкретизировать обобщенный тип в классе реализующим интерфейс (листинг 10):

Листинг 10

```
interface Playable <T>{  
    public void addSong(T t);  
    public T playSong();  
}  
class Player implements Playable<Music>{  
    private Music song;  
    @Override  
    public void addSong(Music song){  
        this.song = song;  
    }  
    @Override  
    public Music playSong(){  
        return song;  
    }  
}
```

Второй вариант – это продолжить использовать обобщения в классе реализующим интерфейс (листинг 11):

Листинг 11

```
class Player<T> implements Playable<T>{  
    private T song;  
    @Override  
    public void addSong(T song){  
        this.song = song;  
    }  
    @Override  
    public T playSong(){  
        return song;  
    }  
}
```

И последний способ, он же не самый лучший – это написать код в «старом стиле», т.е. не использовать обобщения в принципе (листинг 12):

Листинг 12

```
class Player implements Playable{  
    private Object song;  
    @Override  
    public void addSong(Object song){  
        this.song = song;  
    }  
}
```

```
}  
@Override  
public Object playSong(){  
    return song;  
}}
```

Стирание типов (type erasure) и ограничения

В Java для реализации параметризованных классов с целью обеспечения совместимости со старыми версиями языка используется механизм стирания типов во время компиляции. То есть во время выполнения программы фактический тип неизвестен. Стирание выполняется до типа `Object` либо до типа, указанного при помощи указания ограничения. Следующие два примера демонстрируют поведение кода с ограничениями и без них (листинг 13):

Листинг 13

```
public class TstTpl<T> {  
    private T t;  
    public foo() {  
        // t.bar(); Данная запись недопустима!  
    }  
    //...  
}
```

В закомментированной строке производится вызов некоторого метода объекта типа `T`, но в этом случае компилятор выведет сообщение об ошибке. Это произойдет из-за принципа работы механизма стирания. Во время выполнения программа ничего не знает о том, какой конкретный тип имеет переменная `t` и, соответственно, ничего не знает о его методах. Все, что мы можем, – это вызывать методы, определенные в классе **Object**, т.к. стирание произойдет до типа **Object**. А что если мы знаем о том, какие типы будет принимать данный класс и хотим вызывать их методы? Для этого существуют ограничения. Пример с использованием синтаксиса ограничений приведен ниже (листинг 14).

Листинг 14

```
Class MyClass {  
    public void doSomething(){}  
}  
//используем ограничения  
class Generic<T extends MyClass>{  
    private T obj;  
    public void method()
```

```
{
    //Компилятор не выдает никаких сообщений об ошибке!
    obj.doSomething();
}
}
```

Данный тип ограничения называется ограничением сверху. В итоге мы говорим компилятору: в качестве параметра типа могут использоваться только класс `MyClass` или его подклассы. При попытке создать объект параметризованного типа с объектом, не относящимся к данной иерархии класса, компилятор выдаст ошибку. Теперь компилятору известно, что класс `MyClass` и его подклассы имеют метод `doSomething()`, поэтому никакой ошибки не произойдет.

Для задания ограничений типов шаблона метода используется синтаксис со знаком вопроса:

- `?` – указывает на любой класс. классы, но сама специфика класса для них не важна;
- `? extends T` – определяет множество классов потомков от `T`;
- `? super T` – определяет множество родительских классов класса `T`.

Например, если нам нужен метод вывода списка фигур, потомков абстрактного класса `Shape`, то определение метода будет выглядеть следующим образом.

```
public void draw(List<? extends Shape> shape) {
    //...
}
```

Другой пример:

```
interface A{}
class Test{
    public static <T extends A> void check(T t){}; }
```

В метод `check()` мы можем передать только объекты классов реализующих интерфейс `A`. Обратите внимание, что в объявлении используется ключевое слово **`extends`** вместо **`implements`**! Как вы знаете в Java отсутствует множественное наследование в явном виде, но мы вольны классом реализовывать столько интерфейсов, сколько посчитаем нужным, так же и в обобщениях ограничивающим параметром можно указать реализацию двух интерфейсов разделив их знаком **`&`**:

```
interface A{}
interface B{}
class Test{
    public static <T extends A & B> void check(T t){};
}
```


Теперь метод **check()** принимает в качестве аргументов объект класса реализующего интерфейсы **A** и **B**. Ограничения реализуются не только интерфейсами:

```
class A{}
interface B{}
class Test{
    public static <T extends A & B> void check(T t){};
}
```

На этот раз класс должен реализовывать интерфейс **B** и расширять класс **A**. В последнем примере есть один нюанс – если в качестве ограничителя вы вводите какой-либо класс (в нашем примере класс **A**), то он должен стоять на первом месте:

```
class A{}
interface B{}
class Test{
    public static <T extends B & A> void check(T t){}; // ошибка компиляции
}
```

Универсальные обобщения без ограничений

Всякий раз, когда вы используете такие шаблоны, вы как бы говорите, что все в порядке, здесь можно использовать переменную любого типа и код будет работать.

```
List<?> ls = new ArrayList<String>();
List<?> ls1 = new ArrayList<Object>();
```

Как видите, мы можем создать список, содержащий любой обобщенный тип, ограничений никаких не накладывается. Практичный пример (листинг 15):

Листинг 15

```
public class Example{
    public static void main(String... args){
        List<String> list = new ArrayList<>();
        list.add("Обобщения в Java");
        printList(list); // Ошибка компиляции
    }
    public static void printList(List<Object> list){
        for(Object obj : list)
            System.out.println(obj);
    }
}
```

Несмотря на то, что класс **String** является подклассом **Object**, компилятор все равно не дает запустить данный код. Это может показаться запутанным, так оно и есть. Еще один пример:

```
List<Integer> numbers = new ArrayList<>();
numbers.add(new Integer(1));
List<Object> obj = numbers; //Ошибка компиляции
obj.add("one");
System.out.println(numbers.get(1));
```

Ситуация такая же, как и в первом примере, первой строчкой мы явно указали, что будем хранить в списке объекты типа **Integer**, если бы компилятор дал нам это скомпилировать, то мы бы нарушили ограничение, которое сами же и наложили, добавив в наш список строку. Вернемся к первому примеру, ситуация вроде бы безвыходная, но на помощь придут универсальные ограничители (листинг 16).

Листинг 16

```
public class Example{
    public static void main(String... args){
        List<String> list = new ArrayList<>();
        list.add("Обобщения в Java");
        printList(list);
    }
    public static void printList(List<?> list){
        for(Object obj : list) System.out.println(obj);
    }
}
```

На этот раз в методе **printList()** вместо списка с типом **Object** мы воспользовались списком с универсальным ограничителем **?**, который подразумевает под собой любой тип и тип **String** полностью соответствует этому условию.

Универсальные обобщения с восходящим ограничением (ограничение на подтип)

Предположим, надо создать список, который бы позволял хранить в себе переменные подклассов **Number** (иначе говоря, любые числа). Из предыдущей главы мы уже разобрались, почему так сделать нельзя:

```
List<Number> list = new ArrayList<Integer>(); //Ошибка компиляции
```

Как и в прошлый раз, на помощь нам придут универсальные ограничители:

```
List<? extends Number> list = new ArrayList<Integer>();
```

В данном случае универсальный ограничитель сообщает, что мы можем использовать в качестве формального параметра любой подкласс **Number**. Давайте напишем метод, который суммирует все числа в списке:

```
public static long sumList(List<? extends Number> list){
    long sum = 0;
    for(Number num : list) sum += num.longValue();
    return sum;
}
```

Есть одна интересная особенность первых двух рассмотренных ограничителей. При их использовании список становится логически неизменяемым (листинг 17).

Листинг 17

```
class A{}
class B extends A{}
public class Example{
    public static void main(String... args){
        List<? extends A> a = new ArrayList<A>();
        a.add(new B()); //Ошибка компиляции
        a.add(new A()); //Ошибка компиляции
    }
}
```

Проблема кроется в том, что Java не знает какой тип кроется за **<? extends A>**, это может быть **A** или **B** или класс, который еще не написан. Иначе говоря, переменные класса **A** не могут быть добавлены в **List**, а переменные класса **B** не могут быть добавлены в **List<A>**, с точки зрения Java оба сценария возможны, поэтому подобные операции пресекаются компилятором на стадии компиляции. Давайте посмотрим, как универсальные ограничители работают с интерфейсами (листинг 18).

Листинг 18

```
interface ABC{public void letter();}
class A implements ABC{public void letter(){System.out.print("A");}}
class B implements ABC{public void letter(){System.out.print("B");}}
public class Example{
    public static void main(String... args){
        List<B> b = new ArrayList<B>();
        b.add(new B());
    }
}
```

```

    showLetters(b);
}
public static void showLetters(List <? extends ABC> list){
    for(ABC abc: list) abc.letter();
}
}

```

Самое первостепенное, на что хотелось бы обратить внимание, это метод **showLetters()**. При объявлении универсальных ограничителей всегда используется ключевое слово **extends**, не важно с чем вы работаете – с классом или интерфейсом.

Универсальные обобщения с нисходящим ограничением (ограничение на супертип)

Начнем с примера (листинг 19).

Листинг 19

```

import java.util.ArrayList;
import java.util.List;
public class Example{
    public static void main(String... args){
        List<String> listStr = new ArrayList<String>();
        List<Object> listObj = new ArrayList<Object>();
        addString(listStr);
        addString(listObj);
    }
    public static void addString(List<? super String> list){
        list.add("Обобщения в Java");
    }
}

```

В методе **addString()** используется ограничитель с нисходящим ограничением, означающий, что мы можем использовать любые классы, которые являются родительскими классу **String**, включая класс **String**. Почему в методе **addString()** нельзя воспользоваться другими ограничителями?

Рассмотрим этот вопрос подробнее:

- при использовании универсального ограничителя метод попросту не скомпилируется, потому что наш список автоматически становится неизменяемым (**immutable**);

- метод так же не скомпилируется, по той же самой причине, что и выше, при использовании ограничителей с восходящим ограничением – список становится неизменяемым;
- с ограничением только на класс Object в наш метод мы не сможем передать список из строк.

Тема 3.2

ОБРАБОТКА ИСКЛЮЧЕНИЙ В ЯЗЫКЕ JAVA

Класс Exception

Когда происходит что-то необычное во время работы программы, возникают исключения. Все исключения, требующие специальной обработки, наследуются от класса Exception. Метод должен явно указывать в своем объявлении (с помощью throws), какие исключения он может выбросить, за исключением исключений, происходящих от RuntimeException.

Классы исключений, входящие в состав пакета java.lang, приведены на рисунке 2.

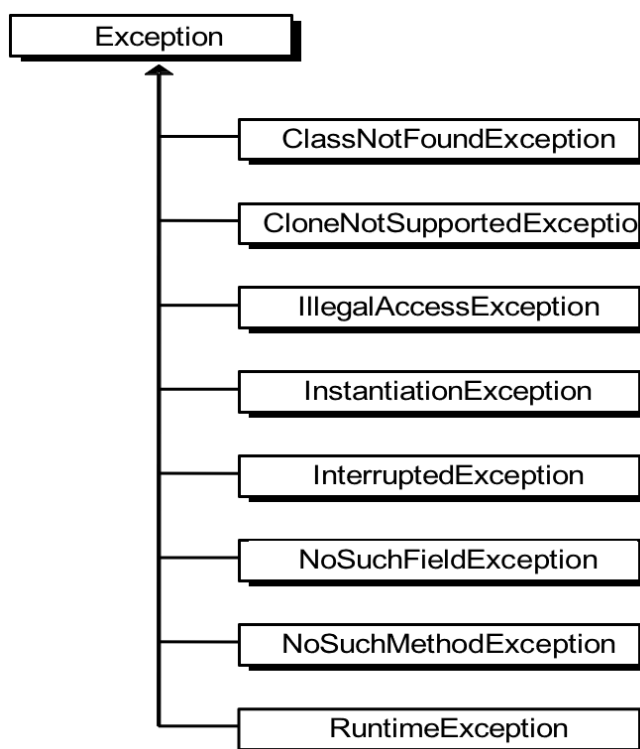


Рисунок 2. – Исключения пакета java.lang

Обработка исключений производится с помощью блока try-catch-finally.

RuntimeException

RuntimeException представляет собой особый вид исключений в Java, которые возникают во время выполнения программы. Главная их особенность в том, что их не обязательно указывать в секции throws метода. Компилятор Java не требует обязательной обработки этих исключений, поэтому их называют «непроверяемыми» или «неконтролируемыми». Они сигнализируют о проблемах, которые, как правило, являются следствием ошибок в коде, но могут возникнуть в любой части программы.

Error

Класс Error представляет собой ошибки, которые обычно не предназначены для перехвата и обработки в программе. Они указывают на серьезные проблемы в системе, такие как нехватка памяти или ошибки виртуальной машины Java. Как правило, возникновение Error приводит к немедленному и аварийному завершению работы приложения. Попытки перехватить и обработать Error обычно нецелесообразны.

Основные классы ошибок приведены на рисунке 3.

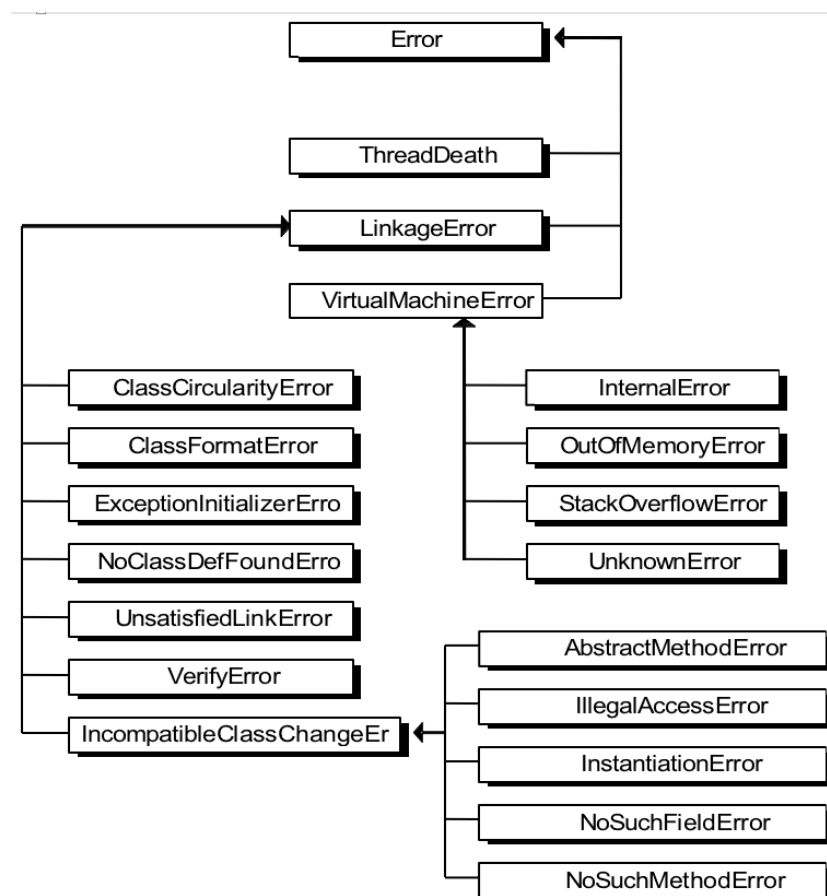


Рисунок 3. – Ошибки пакета java.lang

Множественная обработка исключений

В Java 7 и более поздних версиях можно обрабатывать несколько разных типов исключений в одном блоке `catch`. Это упрощает код, т.к. позволяет избежать дублирования и снижает риск перехвата слишком широкого класса исключений.

Следующий код содержит повторяющиеся фрагменты в каждом блоке `catch`:

```
catch (IOException ex) {  
    logger.log(ex);  
    throw ex;  
catch (SQLException ex) {  
    logger.log(ex);  
    throw ex;  
}
```

В предыдущих версиях Java было сложно создать метод, который бы позволил не дублировать код, т.к. переменная **ex** разного типа.

Ниже приведен пример для Java 7 и выше, решающий проблемы дубликации кода:

```
catch (IOException | SQLException ex)  
    { logger.log(ex);  
    throw ex;  
}
```

В блоке `catch` можно указать несколько типов исключений, которые он будет перехватывать. Эти типы перечисляются через символ `|` (вертикальная черта). Важно помнить, что если в `catch` указано несколько типов исключений, то переменная, представляющая пойманное исключение (например, `ex`), автоматически становится `final`. Это значит, что внутри блока `catch` вы не сможете изменить значение этой переменной.

Проброс исключений более узких типов

В Java 7 улучшен анализ исключений, что позволяет разработчикам более точно контролировать, какие исключения метод может выбросить, делая код более понятным и надежным.

Рассмотрим пример (листинг 20).

Листинг 20

```
static class FirstException extends Exception { }
static class SecondException extends Exception { }

public void rethrowException(String exceptionName) throws Exception {
    try {
        if(exceptionName.equals("First")){
            throw new FirstException();
        } else {
            throw new SecondException();
        }
    } catch (Exception e) {
        throw e;
    }
}
```

До Java 7 при перехвате исключений в блоке catch с общим типом, например, Exception, и последующем их пробрасывании в сигнатуре метода можно было указать только Exception. Это происходило потому, что компилятор не мог определить, какое конкретно исключение было выброшено в блоке try.

```
public void rethrowException(String exceptionName) throws FirstException,
SecondException {
    try {
        // ...
    }
    catch (Exception e) { throw e;
    }
}
```

В Java 7 эта проблема была решена. Теперь компилятор способен анализировать блок try и определять, какие типы исключений могут быть выброшены. В нашем примере, если блок try может выбросить только FirstException и SecondException, то даже если блок catch перехватывает исключение типа Exception, компилятор сможет определить, что пробрасывается либо FirstException, либо SecondException. Это позволяет указывать в блоке throws только эти два типа исключений, а не общий Exception.

1. Изменение переменной исключения в catch блоке:

Если в блоке catch переменной e (представляющей пойманное исключение) присваивается другое значение, то проверка типа исключения, которое будет выброшено дальше, не производится. Т.е. компилятор не будет следить за тем, что вы выбрасываете, если вы изменили значение переменной исключения.

2. Проверка типа исключения при повторном выбрасывании (re-throwing) в catch блоке (Java 7+):

Когда вы повторно выбрасываете исключение (`throw e;`) в блоке `catch`, который обрабатывает определенные типы исключений, компилятор Java (начиная с версии 7) выполняет следующие проверки:

- Возможность выброса из `try`: `try` блок, в котором возникло исключение, должен иметь возможность выбросить исключение такого типа. Это значит, что тип исключения должен быть совместим с типами исключений, которые могут быть сгенерированы в `try` блоке.

- Отсутствие перехвата другими `catch`: не должно быть других `catch` блоков, которые могли бы перехватить это исключение раньше, чем текущий. Порядок `catch` блоков важен.

- Совместимость типов с объявленными в `catch`: тип повторно выбрасываемого исключения должен быть либо тем же типом, что и пойманное исключение, либо его подклассом, либо его суперклассом. Это означает, что должно быть отношение наследования между типом пойманного исключения и типом повторно выбрасываемого исключения.

3. Особенности Java 7 и блока `throws`:

В Java 7 и выше, если метод объявлен с `throws` `FirstException`, `SecondException`, вы можете повторно выбросить любое исключение, которое является предком `FirstException` или `SecondException`. Компилятор понимает, что если метод заявляет о возможности выброса `FirstException` и `SecondException`, то он также может выбросить их общего предка (например, `Exception`).

4. Ограничения в более ранних версиях Java (до Java 7):

В версиях Java до 7, вы не могли повторно выбросить исключение, которое является *предком* исключений, указанных в параметрах блока `catch`. Например, если у вас был `catch (IOException e)`, вы не могли просто сделать `throw e`; если `e` фактически содержало объект типа `Exception` (который является предком `IOException`). Компилятор выдавал ошибку “unreported exception Exception; must be caught or declared to be thrown”, требуя либо поймать `Exception` в другом `catch` блоке, либо добавить `Exception` в список `throws` в объявлении метода. Это ограничение было снято в Java 7.

Выражение `try-with-resources`

Выражение `try-with-resources` позволяет безопасно работать с ресурсами, которые нужно закрывать после использования. Оно объявляет блок `try` и в нем же определяет один или несколько ресурсов (например, файлы или сетевые

соединения). Главное преимущество в том, что try-with-resources автоматически закрывает эти ресурсы после завершения блока try, даже если возникла ошибка. Ресурсом может быть любой объект, реализующий интерфейс java.lang.AutoCloseable (включая объекты, реализующие java.io.Closeable).

Следующий пример извлекает первую строку из файла. Для этого он использует объект класса BufferedReader, который обеспечивает чтение данных из файла. BufferedReader является ресурсом, требующим закрытия по завершении работы с ним.

```
static String readFirstLineFromFile(String path) throws IOException {  
    try (BufferedReader br =  
        new BufferedReader(new FileReader(path))) {  
        return br.readLine();  
    }  
}
```

В этом примере BufferedReader используется в конструкции try-with-resources. BufferedReader объявляется внутри скобок после try. Поскольку BufferedReader реализует интерфейс AutoCloseable (начиная с Java 7), он будет автоматически закрыт после завершения блока try, даже если внутри произойдет ошибка (например, br.readLine() выбросит исключение). Раньше для гарантированного закрытия ресурсов использовали блок finally, как показано в следующем примере (листинг 21):

Листинг 21

```
static String readFirstLineFromFileWithFinallyBlock(String path)  
    throws IOException {  
    BufferedReader br = new BufferedReader(new FileReader(path));  
    try {  
        return br.readLine();  
    } finally {  
        if (br != null) br.close();  
    }  
}
```

Если и чтение строки (br.readLine()), и закрытие потока (br.close()) вызывают исключения, то метод readFirstLineFromFileWithFinallyBlock сообщит только об исключении, возникшем в блоке finally. Исключение, возникшее при чтении строки, будет проигнорировано. При использовании try-with-resources, метод readFirstLineFromFile сообщит об исключении, возникшем при чтении строки

(в блоке try), а исключение, возникшее при закрытии потока (в блоке try-with-resources), будет подавлено.

Блок try-with-resources позволяет объявлять несколько ресурсов одновременно. В следующем примере показано, как извлечь имена файлов из ZIP-архива и записать их в новый текстовый файл (листинг 22).

Листинг 22

```
public static void writeToZipFileContents(String zipFileName,
                                         String outputFileName)
                                         throws
                                         java.io.IOException {

    java.nio.charset.Charset charset = java.nio.charset.StandardCharsets.US_ASCII;
    java.nio.file.Path outputPath = java.nio.file.Paths.get(outputFileName);

    try (
        java.util.zip.ZipFile zf =
            new java.util.zip.ZipFile(zipFileName); java.io.BufferedWriter writer
        =
            java.nio.file.Files.newBufferedWriter(outputPath,
                                                    charset)
    ) {
        for (java.util.Enumeration entries = zf.entries();
             entries.hasMoreElements();) { String
            newLine = System.getProperty("line.separator");
            String zipEntryName =
                ((java.util.zip.ZipEntry)entries.nextElement()).getName() + newLine;
            writer.write(zipEntryName, 0, zipEntryName.length());
        }
    }
}
```

В конструкции try-with-resources можно объявить несколько ресурсов, разделяя их точкой с запятой. После завершения блока кода, заключенного в try, ресурсы, объявленные в try-with-resources, будут автоматически закрыты. Это произойдет независимо от того, завершился ли блок успешно или с исключением. Порядок закрытия ресурсов обратен порядку их создания.

Как и обычный блок try, try-with-resources может включать блоки catch и finally. Важно отметить, что эти блоки будут выполнены только после того, как все ресурсы, объявленные в try-with-resources, будут закрыты.

Подавленные исключения

В Java, при использовании конструкции try-with-resources, могут возникать исключения как внутри блока try, так и при автоматическом закрытии ресурсов. Если оба типа исключений происходят одновременно, то исключение, возникшее при закрытии ресурса, будет «подавлено». Основное исключение, выброшенное из блока try, будет передано вызывающему коду. Однако, информацию о подавленных исключениях можно получить, используя метод `Throwable.getSuppressed()` у основного исключения.

Тема 3.3

БАЗОВЫЕ ОБЪЕКТЫ ЯЗЫКА JAVA. КЛАССЫ-ОБЕРТКИ. СТРОКИ

Класс Object - корень иерархии классов Java

В Java, даже если мы создаем класс без явного указания родительского класса, он автоматически наследуется от класса `Object`. Это означает, что все классы в Java, включая созданные нами, являются наследниками класса `Object`, что означает доступ всех классов к методам, определенным в классе `Object`. Рассмотрим один из таких важных методов.

Метод `toString()` - строковое представление объекта

Когда вы пытаетесь вывести объект как строку (например, с помощью `System.out.println()`), вызывается метод `toString()`. Если вы не определили свой собственный `toString()` для этого объекта, будет выведено его полное имя класса. Например:

Листинг 23

```
public class Program{
    public static void main(String[] args) {

        Person tom = new Person("Tom");
        System.out.println(tom.toString());
        // Будет выводить что-то наподобие Person@7960847b
    }
}
class Person {

    private String name;

    public Person(String name){

        this.name=name;
    }
}
```

Полученное значение (в данном случае `Person@7960847b`) вряд ли может служить хорошим строковым описанием объекта. Поэтому метод `toString()` нередко переопределяют. Например:

Листинг 24

```
public class Program{

    public static void main(String[] args) {

        Person tom = new Person("Tom");
        System.out.println(tom.toString()); // Person Tom
    }
}

class Person {

    private String name;

    public Person(String name){

        this.name=name;
    }

    @Override
    public String toString(){

        return "Person " + name;
    }
}
```

Метод hashCode

Метод `hashCode` в Java генерирует целочисленный хэш-код для объекта. Этот код используется для быстрого сравнения объектов, особенно в коллекциях, основанных на хэш-таблицах (например, `HashMap`, `HashSet`). Например, выведем представление вышеопределенного объекта:

```
Person tom = new Person("Tom");
System.out.println(tom.hashCode()); // 2036368507
```

Но мы можем задать свой алгоритм определения хэш-кода объекта:

Листинг 25

```
class Person {

    private String name;
```

Окончание листинга 25

```
public Person(String name){  
  
    this.name=name;  
}  
    @Override  
public int hashCode(){  
  
    return 10 * name.hashCode() + 20456;  
}  
}
```

Получение типа объекта и метод getClass

Метод getClass позволяет получить тип данного объекта:

```
Person tom = new Person("Tom");  
System.out.println(tom.getClass()); // class Person
```

Метод equals

Метод equals сравнивает два объекта на равенство (листинг 26):

Листинг 26

```
public class Program{  
  
    public static void main(String[] args) {  
  
        Person tom = new Person("Tom");  
        Person bob = new Person("Bob");  
        System.out.println(tom.equals(bob)); // false  
  
        Person tom2 = new Person("Tom");  
        System.out.println(tom.equals(tom2)); // true  
    }  
}  
class Person {  
    private String name;  
    public Person(String name){  
        this.name=name;  
    }  
  
    @Override  
    public boolean equals(Object obj){  
        if (!(obj instanceof Person)) return false;
```

```
Person p = (Person)obj;
return this.name.equals(p.name);
}
}
```

Метод `equals` используется для сравнения объектов на равенство. Он принимает объект любого типа в качестве аргумента. Сначала проверяется, является ли переданный объект экземпляром того же класса, что и текущий объект, используя оператор `instanceof`. Если это так, объект приводится к нужному типу и выполняется сравнение по значимым полям (в данном случае, по имени). Если имена совпадают, метод возвращает `true`, иначе – `false`.

Работа со строками. Класс `String`

Строки реализованы как объекты класса `String`. `String` представляет собой неизменяемую последовательность символов. Класс `String` предоставляет богатый набор методов для выполнения различных операций над строками, таких как поиск, замена, извлечение подстрок и т.д. Объект `String` фактически является ссылкой на область памяти, содержащую символы строки. Для инициализации строки можно использовать конструкторы класса `String` или литеральную запись в двойных кавычках (листинг 27):

Листинг 27

```
public static void main(String[] args) {

    String str1 = "Java";
    String str2 = new String(); // пустая строка
    String str3 = new String(new char[] {'h', 'e', 'l', 'l', 'o'});
    String str4 = new String(new char[] {'w', 'e', 'l', 'c', 'o', 'm', 'e'}, 3, 4); // 3 - начальный индекс,
    4 - кол-во символов

    System.out.println(str1); // Java
    System.out.println(str2); //
    System.out.println(str3); // hello
    System.out.println(str4); // come
}
```

При работе со строками в Java следует помнить об их неизменяемости. Любая операция, которая кажется, что изменяет строку, на самом деле создает новый объект `String`. Это важно учитывать, так как частое создание новых строк может

повлиять на производительность. Для определения количества символов в строке используется метод `length()`:

```
String str1 = "Java";  
System.out.println(str1.length()); // 4
```

С помощью метода `toCharArray()` можно обратно преобразовать строку в массив символов:

```
String str1 = new String(new char[] {'h', 'e', 'l', 'l', 'o'});  
char[] helloArray = str1.toCharArray();
```

Строка может быть пустой. Для этого ей можно присвоить пустые кавычки или удалить из строки все символы:

```
String s = ""; // пустая строка  
if(s.isEmpty()) System.out.println("String is empty");
```

Переменная `String` может не указывать на какой-либо объект и иметь значение `null`:

```
String s = null; // строка не указывает на объект  
if(s.length()==0) System.out.println("String is empty"); // ! Ошибка
```

Переменная не указывает ни на какой объект `String`, поэтому соответственно мы не можем обращаться к методам объекта `String`. Чтобы избежать подобных ошибок, можно предварительно проверять строку на `null`:

```
String s = null; // строка не указывает на объект  
if(s==null || s.length()==0) System.out.println("String is empty");
```

Основные методы класса `String`

В Java класс `String` предоставляет широкий набор методов для работы со строками. Вот некоторые из наиболее важных и часто используемых:

- **Объединение строк:**
 - `concat()`: Склеивает две строки в одну.
 - `join()`: Соединяет несколько строк, используя указанный разделитель.
- **Преобразование в строку:**
 - `valueOf()`: Преобразует объект любого типа в его строковое представление.

- **Сравнение строк:**
 - `compareTo()`: Сравнивает две строки лексикографически (с учетом регистра).
 - `equals()`: Проверяет, идентичны ли две строки (с учетом регистра).
 - `equalsIgnoreCase()`: Проверяет, идентичны ли две строки (без учета регистра).
 - `regionMatches()`: Сравнивает определенные участки (подстроки) двух строк.
- **Поиск в строке:**
 - `indexOf()`: Находит позицию первого вхождения указанной подстроки.
 - `lastIndexOf()`: Находит позицию последнего вхождения указанной подстроки.
 - `startsWith()`: Проверяет, начинается ли строка с указанной подстроки.
 - `endsWith()`: Проверяет, заканчивается ли строка указанной подстрокой.
- **Извлечение символов и подстрок:**
 - `charAt()`: Возвращает символ, находящийся по указанному индексу.
 - `getChars()`: Копирует группу символов из строки в массив символов.
 - `substring()`: Извлекает подстроку из строки, начиная с указанного индекса (и, опционально, заканчивая другим индексом).
- **Модификация строк:**
 - `replace()`: Заменяет все вхождения одной подстроки на другую.
 - `trim()`: Удаляет пробелы в начале и конце строки.
 - `toLowerCase()`: Преобразует все символы строки в нижний регистр.
 - `toUpperCase()`: Преобразует все символы строки в верхний регистр.

Эти методы позволяют выполнять разнообразные операции над строками, от простого объединения до сложного поиска и замены. Разберем работу этих методов.

Соединение строк

Для соединения строк можно использовать операцию сложения ("+"):

```
String strLeft = "Whole";
String strRight = "sentence";
String strRes = strLeft + " " + strRight;

System.out.println(strRes); //Напечатает строку «Whole sentence»
```

Когда вы используете оператор + для объединения строки с чем-то, что не является строкой, это "что-то" будет преобразовано в строковое представление:

```
String str3 = "Год " + 2015;
```

Это делается с помощью метода `String.valueOf()`, который умеет работать с разными типами. Для объектов, `String.valueOf()` вызывает метод `toString()` этого объекта, чтобы получить его строковое представление. Альтернативный способ объединения строк - использование метода `concat()`:

```
String strLeft = "Whole";  
String strRight = "Sentence";  
strLeft = strLeft.concat(strRight); // WholeSentence
```

Метод `concat()` используется для объединения двух строк в одну. Он берет строку, которую нужно добавить, и присоединяет её к исходной строке, возвращая результат.

Метод `join()` предлагает более гибкий способ объединения строк, позволяя указать разделитель, который будет вставлен между ними. В отличие от простого сложения, которое может привести к слитному результату (например, "WholeSentence "), `join()` позволяет вставить между объединяемыми строками нужный символ или последовательность символов. Например, чтобы разделить "Whole" и "Sentence" пробелом, можно использовать `join()`:

```
String strLeft = "Whole";  
String strRight = "Sentence";  
String strRes = String.join(" ", strLeft, strRight); // Whole Sentence
```

Статический метод `join` позволяет объединить несколько строк в одну. Первый аргумент `join` – это строка-разделитель, которая будет вставлена между объединяемыми строками. Последующие аргументы, переданные через запятую, представляют собой строки, которые необходимо объединить. Метод может принимать переменное количество строк для объединения, в данном случае их две.

Извлечение символов и подстрок

В Java, класс `String` предоставляет метод `charAt(int index)` для доступа к отдельным символам строки. Этот метод принимает целочисленный индекс (позицию символа) и возвращает символ, находящийся в этой позиции:

```
String testStr = "SomeString";  
char c = testStr.charAt(5);  
System.out.println(c); // t
```

Строки (объекты класса String) индексируются, начиная с 0, аналогично массивам. Для извлечения подстроки и записи её в массив символов используется метод `getChars()`. Этот метод выполняет копирование диапазона символов из исходной строки в предоставленный массив `char[]`. Параметры метода `getChars()` определяют границы копируемого диапазона и место назначения:

- `srcBegin`: Начальный индекс (включительно) в исходной строке.
- `srcEnd`: Конечный индекс (исключительно) в исходной строке.
- `dst`: Целевой массив `char[]`, в который будут скопированы символы.
- `dstBegin`: Начальный индекс в целевом массиве `dst`, определяющий позицию для вставки скопированных символов.

```
String testStr = "Whole Sentence";
int start = 1;
int end = 5;
char[] dst=new char[end - start];
testStr.getChars(start, end, dst, 0);
System.out.println(dst); // hole
```

Сравнение строк

Для сравнения строк используются методы `equals()` (с учетом регистра) и `equalsIgnoreCase()` (без учета регистра). Оба метода в качестве параметра принимают строку, с которой надо сравнить:

```
String str1 = "Hello";
String str2 = "hello";

System.out.println(str1.equals(str2)); // false
System.out.println(str1.equalsIgnoreCase(str2)); // true
```

Когда нужно сравнить две строки, важно помнить: оператор `==` проверяет, указывают ли две переменные на один и тот же объект в памяти. Чтобы сравнить, содержат ли строки одинаковый текст, используйте метод `equals()`.

Для более детального сравнения, позволяющего сопоставлять отдельные фрагменты строк, существует метод `regionMatches()`. Он имеет две перегруженные версии:

- `boolean regionMatches(int toffset, String other, int ooffset, int len)`
- `boolean regionMatches(boolean ignoreCase, int toffset, String other, int ooffset, int len)`.

Этот метод позволяет сравнить подстроку одной строки с подстрокой другой. Параметры метода определяют следующее:

- ignoreCase: логическое значение, указывающее, нужно ли игнорировать разницу в регистре символов при сравнении. true означает игнорировать регистр.
- toffset: индекс символа в первой строке (той, у которой вызывается метод regionMatches()), с которого начинается сравнение.
- other: строка, с которой производится сравнение.
- ooffset: индекс символа во второй строке (other), с которого начинается сравнение.
- len: количество символов, которые будут сравниваться в обеих строках.

Используем метод:

```
String str1 = "Hello world";  
String str2 = "I work";  
boolean result = str1.regionMatches(6, str2, 2, 3);  
System.out.println(result); // true
```

В данном случае метод сравнивает 3 символа с 6-го индекса первой строки ("wor") и 3 символа со 2-го индекса второй строки ("wor"). Эти подстроки одинаковы, поэтому возвращается true.

Есть еще методы compareTo(String str) и compareToIgnoreCase(String str) для сравнения строк. В отличие от простого определения равенства, эти методы позволяют установить лексикографический порядок между двумя строками.

– **Результат сравнения:** Возвращаемое значение типа int указывает на отношение между строками.

- Положительное значение означает, что первая строка лексикографически больше второй.
- Отрицательное значение означает, что первая строка лексикографически меньше второй.
- Нулевое значение означает, что строки равны.

– **Лексикографический порядок:** Сравнение основывается на порядке символов в алфавите (или, точнее, на их числовых значениях в кодировке Unicode). Строка "A" считается меньше строки "B", потому что символ 'A' предшествует символу 'B'. Если первые символы строк совпадают, сравнение продолжается по следующим символам, и т.д. Например:

```
String str1 = "hello";  
String str2 = "world";  
String str3 = "hell";  
  
System.out.println(str1.compareTo(str2)); // -15 - str1 меньше чем str2  
System.out.println(str1.compareTo(str3)); // 1 - str1 больше чем str3
```

Поиск в строке

Метод **indexOf()** находит индекс первого вхождения подстроки в строку, а метод **lastIndexOf()** - индекс последнего вхождения. Если подстрока не будет найдена, то оба метода возвращают -1:

```
String str = "Hello world";  
int index1 = str.indexOf('l'); // 2  
int index2 = str.indexOf("wo"); //6  
int index3 = str.lastIndexOf('l'); //9
```

Метод **startsWith()** позволяют определить начинается ли строка с определенной подстроки, а метод **endsWith()** позволяет определить заканчивается строка на определенную подстроку:

```
String str = "myfile.exe";  
boolean start = str.startsWith("my"); //true  
boolean end = str.endsWith("exe"); //true
```

Замена в строке

Метод **replace()** позволяет заменить в строке одну последовательность символов на другую:

```
String str = "Hello world";  
String replStr1 = str.replace('l', 'd'); // Heddo wordd  
String replStr2 = str.replace("Hello", "Bye"); // Bye world
```

Обрезка строки

Метод **trim()** позволяет удалить начальные и конечные пробелы:

```
String str = " hello world ";  
str = str.trim(); // hello world
```

Метод **substring()** возвращает подстроку, начиная с определенного индекса до конца или до определенного индекса:

```
String str = "Hello world";  
String substr1 = str.substring(6); // world  
String substr2 = str.substring(3,5); //lo
```

Изменение регистра

Метод `toLowerCase()` переводит все символы строки в нижний регистр, а метод `toUpperCase()` - в верхний:

```
String str = "Hello World";  
System.out.println(str.toLowerCase()); // hello world  
System.out.println(str.toUpperCase()); // HELLO WORLD
```

Метод Split

Метод `split()` позволяет разбить строку на подстроки по определенному разделителю. Разделитель- какой-нибудь символ или набор символов передается в качестве параметра в метод. Например, разобьем текст на отдельные слова:

```
String text = "FIFA will never regret it";  
String[] words = text.split(" ");  
for(String word : words){  
    System.out.println(word);  
}
```

В данном случае строка будет разделяться по пробелу. Консольный вывод:

```
FIFA  
will  
never  
regret  
it
```

StringBuffer и StringBuilder

Строки (`String`) являются неизменяемыми. Это означает, что любая операция, которая, казалось бы, изменяет строку, на самом деле создает новый объект `String`. Это может привести к снижению производительности, особенно при частом изменении строк.

Для решения этой проблемы в Java существуют классы `StringBuffer` и `StringBuilder`. Они представляют собой изменяемые последовательности символов, позволяющие эффективно модифицировать строки без создания новых объектов при каждой операции.

При работе со строками в Java, `StringBuffer` и `StringBuilder` предоставляют схожие возможности. Выбор между ними зависит от контекста. Если ваша программа многопоточная и несколько потоков могут изменять строку одновременно,

используйте `StringBuffer` для предотвращения ошибок. Если же ваша программа однопоточная, `StringBuilder` будет более эффективным, так как он не тратит ресурсы на синхронизацию.

`StringBuffer` определяет четыре конструктора:

```
StringBuffer()  
StringBuffer(int capacity)  
StringBuffer(String str)  
StringBuffer(CharSequence chars)
```

Аналогичные конструкторы определяет `StringBuilder`:

```
StringBuilder()  
StringBuilder(int capacity)  
StringBuilder(String str)  
StringBuilder(CharSequence chars)
```

Рассмотрим работу этих классов на примере функциональности `StringBuffer`.

При работе со строками, которые часто изменяются, использование обычного `String` может быть неэффективным из-за создания новых объектов при каждой модификации. Классы `StringBuffer` и `StringBuilder` решают эту проблему, динамически выделяя память и резервируя дополнительное пространство для будущих изменений. Конструктор по умолчанию создает буфер, способный вместить 16 символов без необходимости немедленного перераспределения памяти. Если требуется другой начальный размер, можно использовать конструктор, принимающий число символов. Конструкторы, принимающие строку или набор символов, также резервируют дополнительно 16 символов. Метод `capacity()` позволяет узнать текущий размер зарезервированной памяти, а `ensureCapacity()` – изменить минимальный размер:

```
String str = "Java";  
StringBuffer strBuffer = new StringBuffer(str);  
System.out.println("Емкость: " + strBuffer.capacity()); // 20  
strBuffer.ensureCapacity(32);  
System.out.println("Емкость: " + strBuffer.capacity()); // 42  
System.out.println("Длина: " + strBuffer.length()); // 4
```

Когда создается `StringBuffer` с начальным значением "Java", его начальная емкость устанавливается в 20 символов (длина строки "Java" плюс 16). Вызов `strBuffer.ensureCapacity(32)` гарантирует, что `StringBuffer` сможет вместить как

минимум 32 символа. Важно отметить, что фактическая емкость может быть больше 32 из-за внутренних оптимизаций Java, направленных на повышение производительности. В приведенном примере, емкость увеличивается до 42, а не до ожидаемых 32 или 36. Несмотря на изменение емкости, длина строки, хранящейся в `StringBuffer`, остается равной 4 (длина "Java"). Для извлечения строк из `StringBuffer` используется метод `toString()`:

```
String str = "Java";
StringBuffer strBuffer = new StringBuffer(str);
System.out.println(strBuffer.toString()); // Java
```

По всем своим операциям `StringBuffer` и `StringBuilder` похожи на класс `String`.

Получение и установка символов

Метод `charAt()` получает, а метод `setCharAt()` устанавливает символ по определенному индексу:

```
StringBuffer strBuffer = new StringBuffer("Java");
char c = strBuffer.charAt(0); // J
System.out.println(c);
strBuffer.setCharAt(0, 'c');
System.out.println(strBuffer.toString()); // cava
```

Метод `getChars()` получает набор символов между определенными индексами:

```
StringBuffer strBuffer = new StringBuffer("world");
int startIndex = 1;
int endIndex = 4;
char[] buffer = new char[endIndex-startIndex];
strBuffer.getChars(startIndex, endIndex, buffer, 0);
System.out.println(buffer); // orl
```

Добавление в строку

Метод `append()` добавляет подстроку в конец `StringBuffer`:

```
StringBuffer strBuffer = new StringBuffer("hello");
strBuffer.append(" world");
System.out.println(strBuffer.toString()); // hello world
```


Метод **insert()** добавляет строку или символ по определенному индексу в StringBuffer:

```
StringBuffer strBuffer = new StringBuffer("word");

strBuffer.insert(3, 'l');
System.out.println(strBuffer.toString()); //world

strBuffer.insert(0, "s");
System.out.println(strBuffer.toString()); //sworld
```

Удаление символов

Метод **delete()** удаляет все символы с определенного индекса до определенной позиции, а метод **deleteCharAt()** удаляет один символ по определенному индексу:

```
StringBuffer strBuffer = new StringBuffer("assembler");
strBuffer.delete(0,2);
System.out.println(strBuffer.toString()); //sembler

strBuffer.deleteCharAt(6);
System.out.println(strBuffer.toString()); //semble
```

Обрезка строки

Метод **substring()** обрезает строку с определенного индекса до конца, либо до определенного индекса:

```
StringBuffer strBuffer = new StringBuffer("hello java!");
String str1 = strBuffer.substring(6); // обрезка строки с 6 символа до конца
System.out.println(str1); //java!

String str2 = strBuffer.substring(3, 9); // обрезка строки с 3 по 9 символ
System.out.println(str2); //lo jav
```

Изменение длины

Для изменения длины StringBuffer (не емкости буфера символов) применяется метод **setLength()**. Если StringBuffer увеличивается, то его строка просто дополняется в конце пустыми символами, если уменьшается – то строка по сути обрезается:

```
StringBuffer strBuffer = new StringBuffer("hello");
strBuffer.setLength(10);
System.out.println(strBuffer.toString()); //"hello  "

strBuffer.setLength(4);
System.out.println(strBuffer.toString()); //"hell"
```

Замена в строке

Для замены подстроки между определенными позициями в StringBuffer на другую подстроку применяется метод **replace()**:

```
StringBuffer strBuffer = new StringBuffer("hello world!");
strBuffer.replace(6,11,"java");
System.out.println(strBuffer.toString()); //hello java!
```

Первый параметр метода `replace` указывает, с какой позиции надо начать замену, второй параметр - до какой позиции, а третий параметр указывает на подстроку замены.

Обратный порядок в строке

Метод **reverse()** меняет порядок в StringBuffer на обратный:

```
StringBuffer strBuffer = new StringBuffer("assembler");
strBuffer.reverse();
System.out.println(strBuffer.toString()); //relbmessa
```

Обертки, распаковка и упаковка

Java стремится к объектно-ориентированной парадигме, где все должно быть объектом. Однако примитивные типы данных (например, `int`, `double`) существуют для повышения производительности и эффективного использования памяти. Чтобы примитивы можно было использовать в контекстах, требующих объекты (например, в коллекциях), в Java предусмотрены классы-обертки. Класс-обертка (например, `Integer` для `int`) создает объект, который содержит значение примитивного типа. Таким образом, можно работать с примитивными значениями как с объектами, не отказываясь от преимуществ примитивных типов (таблица 24).

Таблица 24. – Классы оберток для примитивных типов

Примитивные типы данных	Классы оберток
int	Integer
short	Short
long	Long
byte	Byte
float	Float
double	Double
char	Char
boolean	Boolean

Объекты классов оберток создаются так же, как и любые другие:

```
public static void main(String[] args) {
    Integer i = new Integer(682);
    Double d = new Double(2.33);
    Boolean b = new Boolean(false);
}
```

Примитивные типы данных (вроде int, boolean, double) обладают ограничениями. Одно из главных — отсутствие методов. Это значит, что нельзя напрямую вызывать методы для примитивов, например, преобразовать число int в строку, используя метод toString().

Классы-обертки (например, Integer, Boolean, Double) решают эту проблему. Они "оборачивают" примитивные типы, предоставляя им функциональность объектов, включая методы. Благодаря классам-оберткам, становится возможным:

- Преобразовывать примитивные типы в строки (и обратно). Например, можно легко получить числовое значение типа int из строки, используя методы класса Integer.
- Использовать примитивные типы в коллекциях, которые работают только с объектами.

В общем, классы-обертки расширяют возможности работы с примитивными типами в Java, предоставляя необходимые методы и позволяя использовать их в объектно-ориентированном контексте (листинг 28).

Листинг 28

```
public static void main(String[] args) {
    Integer i = new Integer(123);
    String s = i.toString();
    System.out.println(s);
}
```

```
s = "1453674";  
i = Integer.parseInt(s);  
System.out.println(i);  
}
```

Автоупаковка/автораспаковка

Для обеспечения возможности работы с примитивами как с объектами в Java реализован механизм автоупаковки и автораспаковки.

Автоупаковка (Autoboxing) представляет собой неявное преобразование значения примитивного типа в экземпляр соответствующего класса-обертки. Например, присваивание значения типа `int` переменной типа `Integer` автоматически создает объект `Integer`, содержащий это значение.

Автораспаковка (Autounboxing) – это обратное неявное преобразование, при котором объект класса-обертки автоматически преобразуется в соответствующее значение примитивного типа. Например, присваивание переменной типа `int` объекта типа `Integer` автоматически извлекает значение `int` из объекта `Integer` (листинг 29).

Листинг 29

```
public class Main {  
  
    public static void main(String[] args) {  
        int x = 7;  
        Integer y = 111;  
        x = y;  
        y = x * 34;  
    }  
}
```

В строке 6 мы присваиваем примитиву `x` значение `y`, который является объектом `Integer`. Никаких дополнительных действий для этого не нужно: **компилятор знает, что `int` и `Integer`, по сути, одно и то же**. Это и есть автораспаковка. Так же происходит и автоупаковка в строке 7: объекту `y` легко присваивается значение примитивов (`x*34`). Это пример автоупаковки. Именно поэтому добавляется слово «авто»: для присваивания ссылок-примитивов объектам их классов-обертки (и наоборот) не требуется ничего делать, все происходит автоматически.

Еще одно удобство автоупаковки/автораспаковки проявляется в работе методов. Дело в том, что **параметры методов тоже подлежат автоупаковке и автораспаковке**. И, например, если какой-то из них принимает на вход два объекта `Integer` — мы легко можем передать туда обычные примитивы `int`.

```

public class Main {
    public static void main(String[] args) {
        printNumber(7);
    }

    public static void printNumber(Integer i) {
        System.out.println("Вы ввели число " + i); //7
    }
}

```

Точно так же работает и наоборот:

```

public class Main {
    public static void main(String[] args) {
        printNumber(new Integer(632));
    }

    public static void printNumber(int i) {
        System.out.println("Вы ввели число " + i);
    }
}

```

Важный момент, о котором нужно помнить: **автоупаковка и распаковка не работают для массивов!**

```

public class Main {
    public static void main(String[] args) {
        int[] i = {1,2,3,4,5};
        printArray(i);
    }
    public static void printArray(Integer[] arr) {
        System.out.println(Arrays.toString(arr));
    }
}

```

Попытка передать массив примитивов в метод, который принимает на вход массив объектов, вызовет ошибку компиляции. Еще раз кратко сравним примитивы и обертки.

Примитивы:

- имеют преимущество в производительности.

Обертки:

- позволяют не нарушать принцип «все является объектом», благодаря чему числа, символы и булевы значения true/false не выпадают из этой концепции;
- расширяют возможности работы с этими значениями, предоставляя удобные методы и поля;
- необходимы, когда какой-то метод может работать исключительно с объектами.

Тема 3.4

СБОРКА МУСОРА. ОСНОВНЫЕ ПОНЯТИЯ.

ОСОБЕННОСТИ НАСТРОЙКИ И РЕАЛИЗАЦИИ В ЯЗЫКЕ JAVA

В Java, как и в любом приложении, для работы необходима память. Но поскольку память компьютера не бесконечна, важно автоматически освобождать ее от данных, которые больше не используются. Этот процесс называется сборкой мусора (Garbage Collection). Сборка мусора нужна для того, чтобы программа не исчерпала всю доступную память и могла продолжать работать.

Кто занимается этой очисткой? Как и когда очищается память? Как выглядит структура памяти? Давайте разберем с этим подробнее.

Структура памяти Java

Память в Java состоит из следующих областей (рисунок 4):

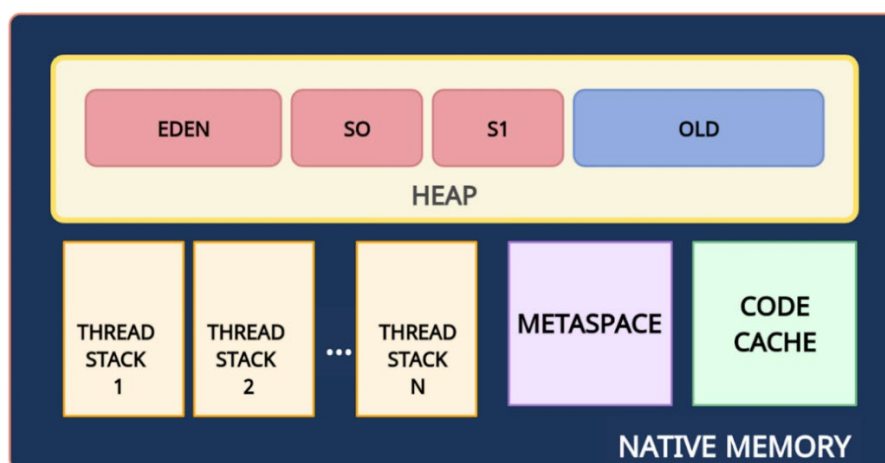


Рисунок 4. – Структура памяти Java

В сборщике мусора можно выделить следующие области памяти:

- **Системная память (Native Memory):** Это вся оперативная память, доступная Java Virtual Machine (JVM) в системе.
- **Куча (Heap):** Область памяти, выделенная JVM из системной памяти, предназначенная для хранения объектов Java. Она является общей для всех потоков приложения. Размер кучи регулируется параметрами `-Xms` (начальный размер) и `-Xmx` (максимальный размер).
- **Стек (Stack):** Каждый поток в Java имеет свой собственный стек, используемый для хранения локальных переменных и информации о вызовах методов.

- **Metaspace (Метапространство):** Область памяти, где JVM хранит метаданные классов (структуру классов, методы и т.д.) и статические переменные. Metaspace является общей для всех потоков и, будучи частью системной памяти, ограничена доступными ресурсами платформы. Максимальный размер Metaspace можно ограничить с помощью параметра `MaxMetaspaceSize`.
- **PermGen (Постоянное поколение):** Область памяти, аналогичная Metaspace, но использовавшаяся в версиях Java до 8. В Java 8 и более поздних версиях PermGen была заменена Metaspace.
- **Кэш кода (CodeCache):** Область памяти, используемая JIT-компилятором (Just-In-Time compiler) для хранения скомпилированного машинного кода, полученного в результате оптимизации часто используемых участков Java-кода. Это позволяет ускорить выполнение программы. CodeCache также является частью системной памяти.

Сборка мусора: введение

В Java сборщик мусора (garbage collector) занимается автоматическим управлением памятью. «Мусор» – это объекты, на которые больше нет ссылок из активного кода программы. Поскольку эти объекты больше не нужны, сборщик мусора освобождает занимаемую ими память, чтобы она могла быть использована повторно.

Например, на диаграмме (рисунок 5) объект `fruit2` может быть удален из памяти, поскольку на него нет ссылок.

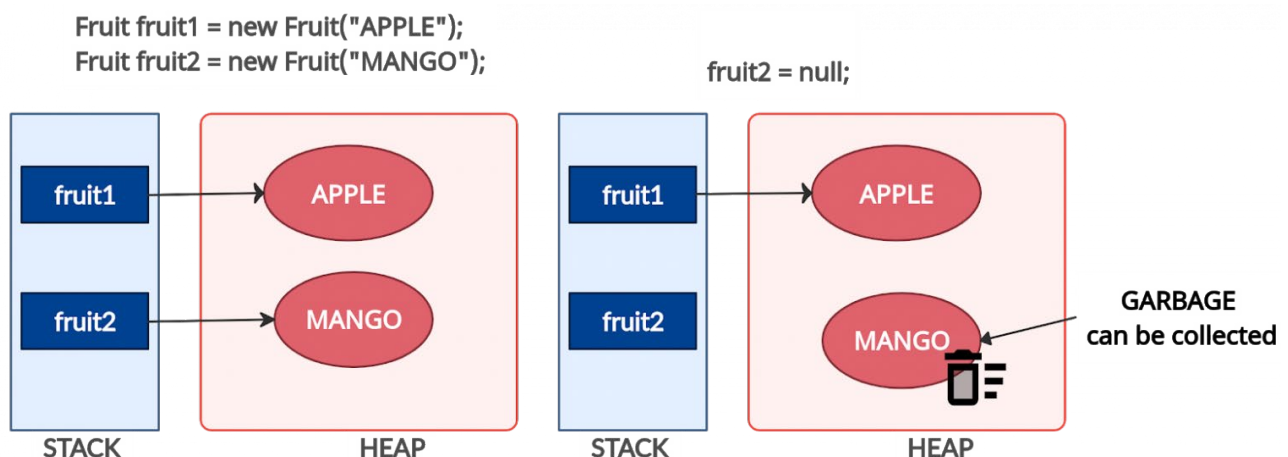


Рисунок 5. – Мусор

Автоматическое управление памятью обеспечивается механизмом сборки мусора (Garbage Collection, GC). GC является частью JVM и отвечает за автоматическое освобождение памяти, занимаемой более не используемыми объектами. Разработчикам не требуется явно управлять выделением и освобождением памяти.

Процесс сборки мусора обычно реализуется с использованием алгоритма Mark & Sweep (пометить и очистить), который включает в себя следующие этапы:

1. **Mark (Маркировка):** GC проходит по всем объектам в куче и помечает объекты, на которые есть ссылки (живые объекты). Этот этап требует приостановки работы приложения ("Stop the World").

2. **Sweep (Очистка):** Память, занимаемая объектами, которые не были помечены как живые, освобождается.

3. **Compact (Уплотнение):** Живые объекты перемещаются в куче, чтобы создать непрерывный блок свободной памяти. Это уменьшает фрагментацию и повышает эффективность выделения памяти.

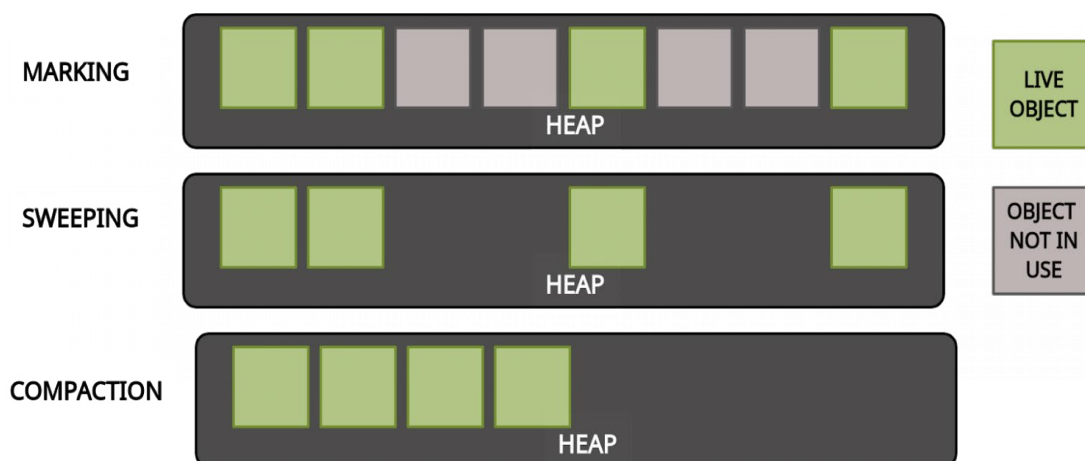


Рисунок 6. – Mark & Sweep GC

Поколения объектов

В Java для более эффективной работы сборщика мусора память кучи разделена на несколько областей, называемых поколениями (рисунок 7). Объекты перемещаются между этими поколениями в зависимости от того, как долго они существуют в программе. **Молодое поколение (Young Generation)** – это место, где создаются все новые объекты. Оно состоит из трех частей: **Eden** (где объекты появляются изначально) и двух областей **Survivor Space (S0 и S1)**, куда перемещаются объекты, пережившие первую сборку мусора. Объекты, которые «живут» достаточно долго, перемещаются в **Старое поколение (Old Generation)**.

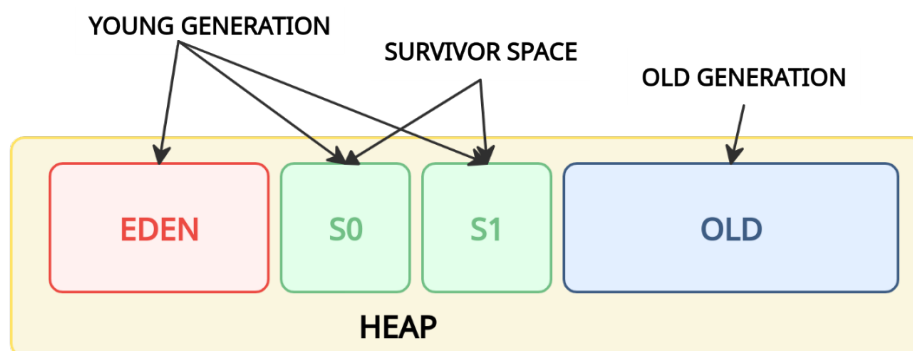


Рисунок 7. – Поколения в куче

Stop-the-World (STW): В контексте сборки мусора Java, "Stop-the-World" (STW) означает приостановку выполнения всех потоков приложения. Это происходит во время фазы маркировки (mark) сборщика мусора. После завершения маркировки, приложение продолжает свою работу. Любая операция сборки мусора, требующая приостановки приложения, называется "Stop-the-World".

Гипотеза о поколениях (Generational Hypothesis):

Для повышения эффективности сборки мусора, часто используется концепция поколений объектов. Гипотеза о поколениях основана на следующих наблюдениях:

1. Большинство объектов в Java-приложениях имеют короткий жизненный цикл.
2. Объекты, которые пережили несколько циклов сборки мусора, с большей вероятностью будут существовать долго.
3. Процессы маркировки и очистки (mark & sweep) выполняются быстрее, когда в рассматриваемой области памяти много «мусора» (неиспользуемых объектов). Другими словами, чем меньше область памяти и чем больше в ней мертвых объектов, тем быстрее происходит маркировка.

Исходя из этой гипотезы, алгоритмы сборки мусора, использующие поколения, строятся, как показано на рисунке 8.

Принцип работы сборки мусора по поколениям

Сборщик мусора (Garbage Collector, GC) использует концепцию поколений для более эффективного управления памятью. Объекты делятся на поколения, исходя из предположения, что большинство объектов «живут» недолго.

– **Молодое поколение (Young Generation):** Состоит из области Eden и двух областей Survivor (S0 и S1).

1. **Создание объектов:** Новые объекты изначально размещаются в области Eden. Области Survivor в начале пусты.

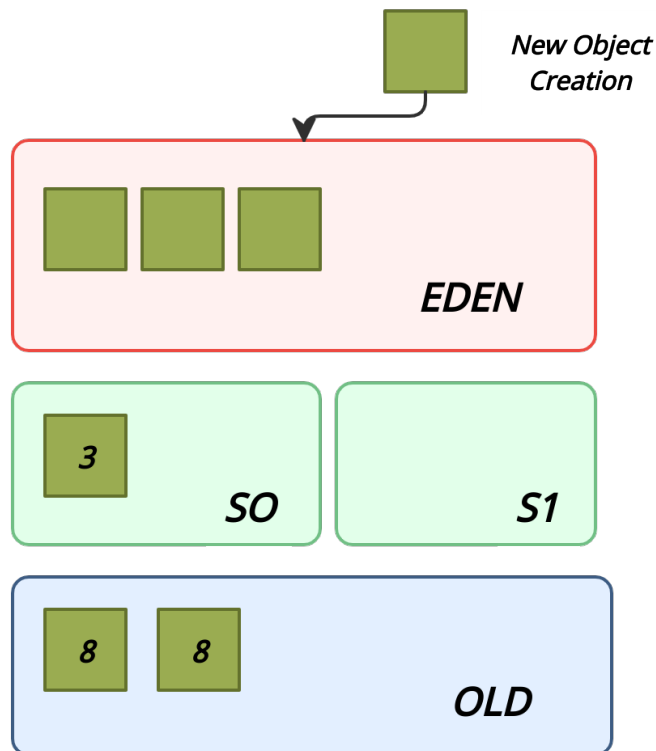


Рисунок 8. – Сборка мусора поколениями

2. **Minor GC:** Когда область Eden заполняется, запускается Minor GC. Это процесс, который идентифицирует и удаляет неиспользуемые объекты (мусор) только в молодом поколении.

3. **Перемещение выживших:** Объекты, которые «выжили» после Minor GC (т.е. все еще используются), перемещаются в одну из областей Survivor (например, S0). Объекты, признанные мусором, удаляются из Eden.

4. **Чередование Survivor:** При последующих Minor GC, процесс повторяется, но области Survivor меняются ролями. Например, если выжившие объекты были перемещены в S0, то при следующем Minor GC, выжившие объекты из Eden и S0 перемещаются в S1. При этом у выживших объектов увеличивается счетчик «возраста», показывающий, сколько Minor GC они пережили.

5. **Повышение в звании (Promotion):** После нескольких циклов перемещения между областями Survivor (или если в Survivor не хватает места) объекты, которые все еще «живы», перемещаются в область старого поколения (Old Generation).

– **Старое поколение (Old Generation):** Содержит объекты, которые «живут» дольше.

1. **Major GC (Full GC):** Когда старое поколение заполняется, запускается Major GC (иногда называемый Full GC). Этот процесс анализирует и очищает

память во всем старом поколении. Major GC обычно занимает больше времени, чем Minor GC, т.к. в старом поколении, как правило, больше «живых» объектов.

Преимущества:

- **Эффективность Minor GC:** Minor GC выполняется быстрее, потому что обрабатывает только небольшую часть кучи (молодое поколение), и в этой области обычно много мусора. Это позволяет снизить время, затрачиваемое на сборку мусора, и повысить производительность приложения.

Недостатки:

- **Неиспользуемое пространство:** В каждый момент времени одна из областей Survivor (S0 или S1) остается пустой, что означает неиспользование части выделенной памяти.

Сборка мусора: флаги

В этом разделе приведены некоторые важные флаги, которые можно использовать для настройки процесса сборки мусора (таблица 25).

Таблица 25

Флаг	Описание
-Xms	Первоначальный размер кучи
-Xmx	Максимальный размер куча
-XX:NewRatio=n	Отношение размера Old Generation к Young Generation
-XX:SurvivorRatio=n	Отношение размера Eden к Survivor
-XX:MaxTenuringThreshold=n	Возраст объекта, когда объект перемещается из области Survivor в область Old Generation

Типы сборщиков мусора

- Serial GC подходит для простых приложений с небольшими объемами данных, где не критичны паузы GC. (Включение: «-XX:+UseSerialGC»).
- Parallel GC обеспечивает высокую пропускную способность, но может приводить к более длительным паузам. (Включение: «-XX:+UseParallelGC»).
- G1 GC является хорошим компромиссом между пропускной способностью и временем отклика. (Включение: «-XX:+UseG1GC»).
- ZGC предназначен для приложений, требующих минимальных задержек, даже при работе с очень большими объемами данных. (Включение: «-XX:+UseZGC»).

Сборщики мусора (GC) автоматически освобождают память, занятую неиспользуемыми объектами. Однако процесс сборки мусора может приводить

к паузам в работе приложения ("stop the world"), поэтому важно следить за его эффективностью. Для этого используются инструменты мониторинга GC, которые позволяют отслеживать:

1. **Частоту сборок мусора:** чем реже запускается GC, тем лучше для производительности приложения.
2. **Длительность сборок мусора:** чем меньше время, затрачиваемое на каждую сборку, тем меньше влияние на отзывчивость приложения.

Для мониторинга GC можно использовать такие инструменты, как VisualVM. Также можно включить логирование событий GC, добавив специальные параметры при запуске JVM:

```
-XX:+PrintGCDateStamps -verbose:gc -XX:+PrintGCDetails -
Xloggc:/tmp/[Application-Name]-[Application-port]-%t-gc.log -
XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=20 -
XX:GCLogFileSize=100M
```

Логи GC выглядят следующим образом:

```
[15,651s][info ][gc] GC(36) Pause Young (G1 Evacuation Pause) 239M->57M(307M) (15,646s,
15,651s) 5,048ms
```

Объяснение:

[Время старта приложения = 15,651s]

[Уровень сообщения = info]

[Тег = gc]

[Тип GC = Pause Young]

[Причина запуска GC = G1 Evacuation Pause]

[Информация о потреблении памяти : "занято до GC" = 239M -> "занято после GC" = 57M
(Размер кучи = 307M)]

[Время начала и окончания GC = 15,646s, 15,651s]

[Общее время GC = 5,048ms]

Тема 3.5 КОЛЛЕКЦИИ ЯЗЫКА JAVA, ОСНОВНЫЕ ИНТЕРФЕЙСЫ

Коллекции

Коллекции – это классы для сохранения, получения, манипулирования множеством объектов.

Иерархия коллекций представлена на рисунке 9.

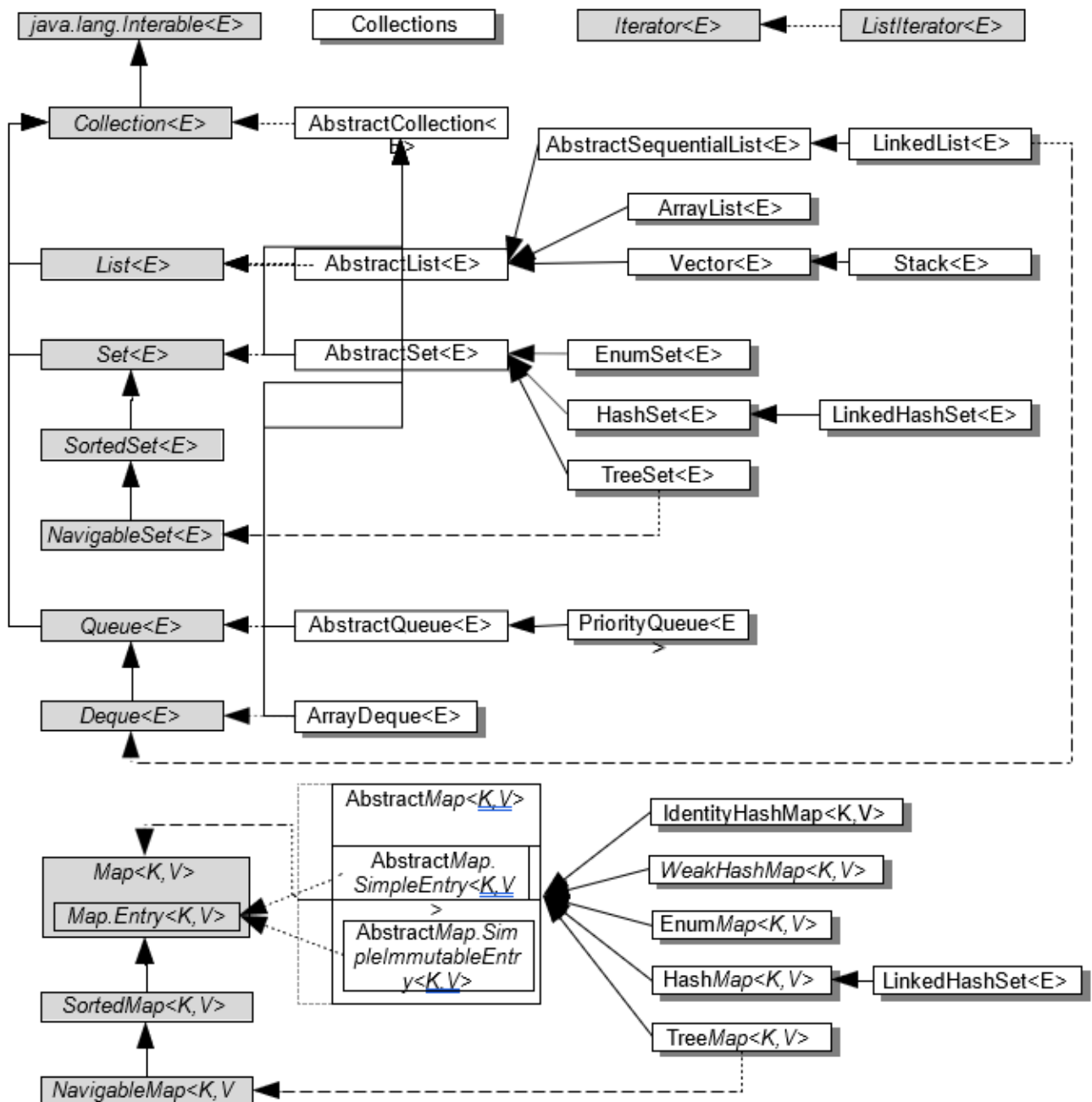


Рисунок 9. – Коллекции

Интерфейс Iterator

Iterator – интерфейс, позволяющий выполнять обход элементов коллекции. Является переработкой и развитием интерфейса Enumeration (таблица 26).

Таблица 26

boolean hasNext();	возвращает true, если в коллекции есть еще элементы
E next();	следующий элемент
void remove();	удалить элемент

Рекомендуется использовать Iterator вместо Enumeration. Iterator предоставляет более лаконичный интерфейс и позволяет удалять элементы из коллекции во время итерации. ListIterator расширяет возможности Iterator, обеспечивая двунаправленный обход списка.

Интерфейс Collection

Интерфейс Collection определяет общий способ работы с группами объектов. Конкретные классы и интерфейсы, реализующие Collection, определяют, как именно эти объекты хранятся и организуются.

Таблица 27

Метод	Действие
int size()	Возвращает размер коллекции
boolean isEmpty()	Возвращает true, если в коллекции нет элементов
boolean contains(Object o)	Возвращает true, если коллекция содержит элемент
Iterator<E> iterator();	Возвращает Iterator
Object[] toArray();	Преобразует коллекцию в массив объектов
<T> T[] toArray(T[] a);	Преобразует коллекцию в массив типа T, размещая в массиве a
boolean add(E e);	Добавляет элемент в коллекцию. Возвращает true, если элемент добавлен; false — если коллекция запрещает дубликаты элементов и элемент не уникален
boolean remove(Object o);	Удаляет <i>один</i> элемент (даже если таких несколько). Возвращает true, если элемент удален
boolean containsAll(Collection<?> c);	Возвращает true, если коллекция содержит все элементы заданной коллекции
boolean addAll(Collection<? extends E> c);	Добавляет элементы в коллекцию. Возвращает true, если коллекция изменена в результате операции

Окончание таблицы 27

Метод	Действие
<code>boolean removeAll(Collection<?>c);</code>	Удаляет все элементы, заданные в коллекции с
<code>boolean retainAll(Collection<?>c);</code>	Удаляет из коллекции все элементы, кроме тех, которые заданы в коллекции с
<code>void clear();</code>	Удаляет из коллекции все элементы
<code>boolean equals(Object o);</code>	Проверяет коллекцию на равенство другой коллекции
<code>int hashCode();</code>	Возвращает хеш-код коллекции

Конкретные реализации интерфейса `Collection` могут не поддерживать все методы, определенные в этом интерфейсе. Если метод не поддерживается, он должен выбрасывать исключение `UnsupportedOperationException`. При этом любая коллекция обязана предоставлять возможность итерации по своим элементам, т.е. должна возвращать объект `Iterator`.

Интерфейсы коллекций (таблица 28)

Таблица 28

Название	Функционал
<code>Set</code>	Множество. Элементы уникальны и, возможно, отсортированы
<code>List</code>	Сортированная последовательность элементов, с возможностью дублирования и позиционного доступа
<code>Queue</code>	Очередь, предназначенная для размещения элемента перед его обработкой. Расширяет коллекцию методами для вставки, выборки и просмотра элементов
<code>Deque</code>	Двунаправленная очередь, позволяющая вставку и удаление в два конца очереди
<code>Map</code>	Карта, это соответствие ключ — значение. Каждому ключу соответствует одно значение
<code>SortedSet</code>	Множество, элементы которого автоматически сортируются либо в их натуральном порядке (интерфейс <code>Comparable</code>), либо с использованием <code>Comparator</code>
<code>SortedMap</code>	Автоматически сортированная карта (см. <code>SortedSet</code>)
<code>NavigableSet</code>	<code>SortedSet</code> , расширенный методами кратчайшего доступа к искомому элементу. В <code>NavigableSet</code> элементы могут быть доступны в порядке увеличения и уменьшения
<code>NavigableMap</code>	<code>SortedMap</code> , расширенный методами кратчайшего доступа к искомому элементу

В пакете `java.util.concurrent`, предлагаются специализированные интерфейсы для работы с многопоточными коллекциями:

- `BlockingQueue` и `BlockingDeque`: Эти интерфейсы позволяют создавать очереди и двусторонние очереди с ограниченной вместимостью. Если очередь заполнена, потоки, пытающиеся добавить элементы, будут заблокированы до тех пор, пока не освободится место.

- `ConcurrentMap` и `ConcurrentNavigableMap`: Эти интерфейсы предоставляют ассоциативные массивы (карты), в которых операции добавления, удаления и изменения элементов выполняются атомарно, что обеспечивает безопасную работу в многопоточной среде.

Для упрощения процесса создания собственных коллекций Java предоставляет абстрактные классы, реализующие базовую функциональность:

- `AbstractCollection`, `AbstractMap` и другие подобные классы: Они предоставляют готовую, хотя и простую, реализацию основных методов интерфейсов коллекций, что позволяет разработчикам сосредоточиться на реализации специфической логики, а не на написании стандартного кода.

Коллекции общего назначения (таблица 29)

Таблица 29

Название	Функционал
<code>HashSet</code>	Реализация множества с использованием хеш-таблиц, обладающая самым высоким быстродействием
<code>TreeSet</code>	Реализация <code>NavigableSet</code> интерфейса с использованием раскрашенных деревьев
<code>LinkedHashSet</code>	Реализация множества в виде хеш таблицы и двусвязанного списка с заданным порядком элементов
<code>ArrayList</code>	Массив переменного размера. Не является потокобезопасным <code>Vector</code> . Наиболее высокое быстродействие
<code>ArrayDeque</code>	Эффективная реализация интерфейса <code>Deque</code> переменного размера
<code>LinkedList</code>	Двусвязная реализация интерфейса <code>List</code> . Быстрее <code>ArrayList</code> , если элементы часто добавляются и удаляются. В дополнение реализует интерфейс <code>Deque</code>
<code>PriorityQueue</code>	Реализация очереди с приоритетами
<code>HashMap</code>	Реализация интерфейса <code>Map</code> с использованием хеш-таблиц
<code>TreeMap</code>	Реализация <code>NavigableMap</code> интерфейса с использованием раскрашенных деревьев
<code>LinkedHashMap</code>	Реализация карты в виде хеш-таблицы и двусвязанного списка с заданным порядком элементов

Существуют еще прародители коллекций — классы Vector и Hashtable, которые были обновлены с использованием шаблонов.

Специальные коллекции (таблица 30)

Таблица 30

Название	Функционал
WeakHashMap	Карта, сохраняющие слабые ссылки на ключи. Позволяет сборщику мусора уничтожить пару ключ-значение, когда на на ключ более нет внешних ссылок
IdentityHashMap	Реализация интерфейса Mapс использованием хеш таблицы и сравнением объекта на равенство по ссылке (key1==key2) вместо сравнения по значению (key1.equals(key2))
CopyOnWriteArrayList	Реализация List, в которой операции – мутаторы (add, set, remove) реализуются путем создания новой копии List. В результате нет необходимости в синхронизации
CopyOnWriteArraySet	Реализация Setс созданием новой копии по операции изменения (см. CopyOnWriteArrayList)
EnumSet	Высокопроизводительная реализация множества с использованием битового вектора. Все элементы должны быть элементами одного Enum
EnumMap	Высокопроизводительная реализация карты с использованием битового вектора. Все ключи должны быть элементами одного Enum

Пакет java.util.concurrent дополняет реализации коллекций классами ConcurrentLinkedQueue, LinkedBlockingQueue, ArrayBlockingQueue,

PriorityBlockingQueue, DelayQueue, SynchronousQueue, LinkedBlockingDeque, ConcurrentHashMap, ConcurrentSkipListSet, ConcurrentSkipListMap, которые подготовлены для использования в многопоточных программах и реализуют различную дополнительную функциональность.

Сортировка элементов в коллекциях Java (например, в SortedMap)

При работе с коллекциями, поддерживающими автоматическую сортировку (например, SortedMap), элементы упорядочиваются одним из двух способов:

1. **Естественный порядок:** Если элементы коллекции реализуют интерфейс Comparable, то используется метод compareTo() для определения порядка. Этот метод позволяет каждому элементу «знать», как его сравнивать с другими элементами того же типа.

- compareTo(E other): Этот метод сравнивает текущий объект (this) с другим объектом (other). Он возвращает:

- Отрицательное число: если this меньше other.
- Ноль: если this равен other.
- Положительное число: если this больше other.

Пример для Byte: Реализация compareTo в классе Byte сравнивает числовые значения двух объектов Byte.

2. Внешний компаратор: Если естественный порядок не подходит или элементы не реализуют Comparable, можно использовать интерфейс Comparator. Comparator предоставляет отдельный объект, который определяет логику сравнения.

- java.util.Comparator содержит следующие методы:
 - compare(T o1, T o2): Сравнивает два объекта (o1 и o2) и возвращает отрицательное, нулевое или положительное значение, как и compareTo.
 - equals(Object obj): Проверяет, равен ли данный Comparator другому объекту obj. Два компаратора считаются равными, если они используют одинаковую логику сравнения. Важно отметить, что переопределение этого метода не всегда требуется, и часто достаточно полагаться на реализацию по умолчанию из класса Object.

Важные моменты:

- SortedMap и другие подобные интерфейсы автоматически поддерживают сортировку элементов.
- Можно использовать либо естественный порядок (через Comparable), либо внешний компаратор (через Comparator).
- Comparator предоставляет большую гибкость, позволяя определять различные стратегии сортировки для одной и той же коллекции.
- compareTo и compare должны быть согласованы с методом equals для обеспечения корректной работы коллекций, основанных на сравнении (например, TreeSet, TreeMap).

Пример сортировки по умолчанию:

```
import java.util.SortedMap;
import java.util.TreeMap;

public class SortedMapExample {
    public static void main(String[] args) {
        SortedMap<String, Integer> sortedMap = new TreeMap<>();
        sortedMap.put("banana", 3);
```

```

sortedMap.put("apple", 2);
sortedMap.put("cherry", 1);

// Вывод отсортированных элементов
for (String key : sortedMap.keySet()) {
    System.out.println(key + ": " + sortedMap.get(key));
}
// Ожидаемый вывод:
// apple: 2
// banana: 3
// cherry: 1

}
}

```

Пример сортировки с использованием компаратора:

```

import java.util.Comparator;
import java.util.SortedMap;
import java.util.TreeMap;

public class SortedMapComparatorExample {
    public static void main(String[] args) {
        // Сортировка в обратном алфавитном порядке
        SortedMap<String, Integer> sortedMap = new
            TreeMap<>(Comparator.reverseOrder());

        sortedMap.put("banana", 3);
        sortedMap.put("apple", 2);
        sortedMap.put("cherry", 1);

        // Вывод отсортированных элементов
        for (String key : sortedMap.keySet()) {
            System.out.println(key + ": " + sortedMap.get(key));
        }
        // Результат сортировки:
        // cherry: 1
        // banana: 3
        // apple: 2
    }
}

```

Класс Collections

Collections — класс, состоящий из статических методов, осуществляющих различные служебные операции над коллекциями (таблица 31).

Таблица 31

Название	Функционал
sort(List)	Сортировать список, используя merge sort алгоритм, с гарантированной скоростью $O(n \cdot \log n)$
binarySearch(List, Object)	Бинарный поиск элементов в списке
reverse(List)	Изменить порядок элементов в списке на противоположный
shuffle(List)	Случайно перемешать элементы
fill(List, Object)	Заменить каждый элемент заданным
copy(List dest, List src)	Скопировать список src в dst
min(Collection)	Вернуть минимальный элемент коллекции
max(Collection)	Вернуть максимальный элемент коллекции
rotate(List list, int distance)	Циклически повернуть список на указанное число элементов
replaceAll(List list, Object oldVal, Object newVal)	Заменить все объекты на указанные
indexOfSubList(List source, List target)	Вернуть индекс первого подсписка source, который эквивалентен target
lastIndexOfSubList(List source, List target)	Вернуть индекс последнего подсписка source, который эквивалентен target
swap(List, int, int)	Заменить элементы в указанных позициях списка
unmodifiableCollection (Collection)	Создает неизменяемую копию коллекции. Существуют отдельные методы для Set, List, Map, и т.д.
synchronizedCollection (Collection)	Создает потокобезопасную копию коллекции. Существуют отдельные методы для Set, List, Map, и т.д.
checkedCollection (Collection<E> of Class<E> type)	Создает типобезопасную копию коллекции, предотвращая появление неразрешенных типов в коллекции. Существуют отдельные методы для Set, List, Map, и т.д.
<T> Set<T> singleton(T o);	Создает неизменяемый Set, содержащую только заданный объект. Существуют методы для List и Map
<T> List<T> nCopies(int n, T o)	Создает неизменяемый List, содержащий n копий заданного объекта
frequency(Collection, Object)	Подсчитать количество элементов в коллекции
reverseOrder()	Вернуть Comparator, которые предполагает обратный порядок сортировки элементов
list(Enumeration<T> e)	Вернуть Enumeration в виде ArrayList
disjoint(Collection, Collection)	Определить, что коллекции не содержат общих элементов
addAll(Collection<? super T>, T[])	Добавить все элементы из массива в коллекцию
newSetFromMap(Map)	Создать Set из Map
asLifoQueue(Deque)	Создать Last In First Out Queue-представление из Deque

Тема 3.6

ОСНОВНЫЕ РЕАЛИЗАЦИИ КОЛЛЕКЦИЙ ЯЗЫКА JAVA

СПИСОК (LIST)

Списки реализуют интерфейс **List**, который, в свою очередь, расширяет интерфейс **Collection**. Список позволяет хранить любые значения, в том числе повторяющиеся. Итерация (обход) списка происходит в порядке добавления элементов. Т.е. элемент, добавленный первым, при итерации также будет первым.

```
List<String> list = new ArrayList<>();  
list.add("яблоко");  
list.add("ананас");  
list.add("яблоко");  
System.out.println(list); // На экране увидим: [яблоко, ананас, яблоко]
```

Две наиболее частые реализации интерфейса **List** – это **ArrayList** и **LinkedList**. Класс **ArrayList** построен на базе массива. Это означает, что доступ по индексу (порядковому номеру элемента) происходит очень быстро. А добавление элементов в середину списка в общем случае довольно затратно, т.к. нужно будет подвинуть вправо каждый элемент, который идет после добавляемого. С удалением такая же штука. Кроме того, массив, лежащий в основе этой структуры данных, имеет конечное количество свободных ячеек и, если их перестанет хватать, придется создать новый массив большего размера, перенеся в него все элементы из исходного. Но все это скрыто внутри реализации **ArrayList**.

Класс **LinkedList** представляет собой цепочку элементов, в которой каждый элемент имеет ссылку на предыдущий элемент и на следующий. Также имеется ссылка на начало и на конец списка, что позволяет быстро получать доступ к первому и к последнему элементу. При этом для доступа по индексу требуется пройти последовательно по всей цепочке, поэтому время доступа по индексу прямо пропорционально размеру списка. Однако сам процесс добавления и удаления элементов весьма прост: нужно всего лишь изменить пару ссылок.

Казалось бы, у каждой реализации списка свои плюсы и минусы, однако в последних версиях Java **ArrayList** был достаточно хорошо оптимизирован и на практике можно всё время выбирать его без ущерба для производительности в любых задачах.

Набор (set)

Интерфейс **Set** представляет собой набор уникальных значений и тоже наследуется от **Collection**. У этого интерфейса есть несколько реализаций, но каждая из них гарантирует, что каждое значение в наборе уникально.

Сравнение любых двух объектов между собой происходит при помощи методов **equals()** и **hashCode()** из базового класса **Object**. Метод **equals** выполняет полное сравнение элементов, тогда как **hashCode** только лишь вычисляет хеш-функцию, которая может принимать одинаковые значения для разных элементов. Соотношение между этими двумя методами всегда такое: если для двух объектов **hashCode** возвращает одинаковое значение, то **equals** при этом **может** быть **false**, однако если **equals** возвращает **true**, то **hashCode** **обязан** возвращать одинаковые значения. При создании собственных классов, если их предполагается использовать в наборах или словарях, вы должны переопределить эти два метода, чтобы коллекции работали с ними корректно.

Теперь рассмотрим три основные реализации интерфейса **Set**.

Первая из них – это **HashSet**. Когда мы будем выполнять проход по этому набору, то порядок элементов на первый взгляд нам покажется хаотичным. Однако на самом деле он будет зависеть от значения хэш-функции для каждого элемента. Также обратите внимание, что мы два раза добавляем «яблоко» в набор, однако в результате увидим его только один раз.

```
Set<String> hashSet = new HashSet<>();
hashSet.add("яблоко");
hashSet.add("яблоко"); // дубль
hashSet.add("ананас");
hashSet.add("банан");
System.out.println(hashSet); // На экране увидим: [банан, яблоко, ананас]
```

Следующая реализация – это **LinkedHashSet**, которая расширяет предыдущую. Основное различие заключается в том, что при обходе элементов мы будем видеть их в порядке добавления:

```
Set<String> linkedHashSet = new LinkedHashSet<>();
linkedHashSet.add("яблоко");
linkedHashSet.add("яблоко");
linkedHashSet.add("ананас");
linkedHashSet.add("банан");
System.out.println(linkedHashSet); // [яблоко, ананас, банан]
```

Третья реализация под названием **TreeSet** имеет в своей основе структуру данных «красно-черное дерево», что позволяет сортировать элементы автоматически:

```
Set<String> treeSet = new TreeSet<>();
treeSet.add("яблоко");
treeSet.add("яблоко");
treeSet.add("ананас");
treeSet.add("банан");
System.out.println(treeSet); // [ананас, банан, яблоко]
```

Поэтому если нужно сохранять порядок добавления элементов – подойдет **LinkedHashSet**, а если нужно отсортированный набор – тогда **TreeSet**.

Словарь (map)

Интерфейс **Map** представляет собой набор из пар элементов типа «ключ–значение». Гарантируется, что каждому ключу соответствует одно и только одно значение. Если по уже существующему ключу положить новое значение, то оно перезапишет старое. При работе с ключами также используются методы `equals` и `hashCode`. И по аналогии с **Set** здесь мы также имеем три основных реализации интерфейса **Map**.

Первая реализация – это **HashMap**, которая не гарантирует никакого порядка элементов при обходе. Обратите внимание, что при повторном добавлении элемента с тем же ключом мы теряем первое значение:

```
Map<String, Integer> hashMap = new HashMap<>();
hashMap.put("яблоко", 1);
hashMap.put("яблоко", 2); // повторное добавление
hashMap.put("ананас", 3);
hashMap.put("банан", 4);
System.out.println(hashMap); // На экране увидим: {банан=4, яблоко=2, ананас=3}
```

Ещё одна реализация – это **LinkedHashMap**, которая сохраняет порядок добавления:

```
Map<String, Integer> linkedHashMap = new LinkedHashMap<>();
linkedHashMap.put("яблоко", 1);
linkedHashMap.put("яблоко", 2);
linkedHashMap.put("ананас", 3);
linkedHashMap.put("банан", 4);
System.out.println(linkedHashMap); // {яблоко=2, ананас=3, банан=4}
```

Ну и третья популярная реализация интерфейса Map – это **TreeMap**, которая сортирует ключи по порядку:

```
Map<String, Integer> treeMap = new TreeMap<>();
treeMap.put("яблоко", 1);
treeMap.put("яблоко", 2);
treeMap.put("ананас", 3);
treeMap.put("банан", 4);
System.out.println(treeMap); // {ананас=3, банан=4, яблоко=2}
```

Поэтому, если нужно сохранять порядок добавления элементов, подходит **LinkedHashMap**, а если нужно получить отсортированный по ключам набор – тогда **TreeMap**.

Тема 3.7

ПОТОКИ ВВОДА-ВЫВОДА В ЯЗЫКЕ JAVA. РАБОТА С ФАЙЛАМИ

Для работы с данными, используются потоки (streams). Поток – это способ читать или записывать данные, неважно откуда они берутся или куда идут. Все потоки находятся в пакете `java.io`. Есть два основных типа: потоки ввода (для чтения данных) и потоки вывода (для записи данных). Также потоки делятся на байтовые (работают с байтами) и символьные (работают с текстом).

	Потоки ввода	Потоки вывода
байтовые	<code>InputStream</code>	<code>OutputStream</code>
символьные	<code>Reader</code>	<code>Writer</code>

При работе с потоками ввода-вывода (I/O), помимо базовых классов потоков, существуют:

- **Специализированные потоки:** Они оптимизированы для взаимодействия с конкретными типами источников данных (например, файлы, сокет) или для записи данных в определенные приемники.

- **Потоки-преобразователи:** Эти потоки выполняют обработку данных «на лету». Они принимают данные в одном формате, преобразуют их и выдают в другом, например, сжимают, шифруют или меняют кодировку текста.

Класс InputStream

Представляет абстрактный входной поток байтов и является предком для всех входных байтовых потоков.

Конструктор

InputStream()	Создает входной байтовый поток
---------------	--------------------------------

Методы (таблица 32)

Таблица 32

Название	Функционал
abstract int read() throws IOException	Читает очередной байт данных из входного потока. Значение должно быть от 0 до 255. При достижении конца потока возвращается -1. При ошибке ввода-вывода генерируется исключение. Подклассы должны обеспечить реализацию данного метода
int read(byte[] buf)	Читает данные в буфер и возвращает количество прочитанных байтов
int read(byte[] buf, int offset, int len)	Читает не более len байтов в буфер, заполняя его со смещением offset, и возвращает количество прочитанных байтов
void close()	Закрывает поток
int available()	Возвращает количество доступных на данный момент байтов для чтения из потока
long skip(long n)	Пропускает указанное количество байтов потока
boolean markSupported()	Проверка на возможность повторного чтения из потока
void mark(int limit)	Устанавливает метку для последующего повторного чтения; limit – размер буфера для операции повторного чтения
void reset()	Возвращает указатель потока на предварительно установленную метку. Дальнейшие вызовы метода read() будут снова возвращать данные, начиная с заданной метки

Класс OutputStream

Представляет абстрактный выходной поток байтов и является предком для всех выходных байтовых потоков.

Конструктор

OutputStream()	Создает выходной байтовый поток
----------------	---------------------------------

Методы (таблица 33)

Таблица 33

Название	Функционал
<code>abstract void write(int n) throws IOException</code>	Записывает очередной байт данных в выходной поток. Значащими являются 8 младших битов, старшие — игнорируются. При ошибке ввода-вывода генерируется исключение. Подклассы должны обеспечить реализацию данного метода
<code>void write(byte[] buf)</code>	Записывает в поток данные из буфера
<code>void write(byte[] buf, int offset, int len)</code>	Записывает в поток <code>len</code> байтов из буфера, начиная со смещения <code>offset</code>
<code>void close()</code>	Закрывает поток
<code>void flush()</code>	Заставляет освободить возможный буфер потока, отправляя на запись все записанные в него данные

Класс Reader

Представляет абстрактный входной поток символов и является предком для всех входных символьных потоков.

Конструктор

<code>Reader()</code>	Создает входной символьный поток
-----------------------	----------------------------------

Методы (таблица 34)

Таблица 34

Название	Функционал
<code>abstract int read() throws IOException</code>	Читает очередной символ Unicode из входного потока. При достижении конца потока возвращается -1. При ошибке ввода-вывода генерируется исключение. Подклассы должны обеспечить реализацию данного метода
<code>int read(char[] buf)</code>	Читает данные в буфер и возвращает количество прочитанных символов
<code>int read(char[] buf, int offset, int len)</code>	Читает не более <code>len</code> символов в буфер, заполняя его со смещением <code>offset</code> , и возвращает количество прочитанных символов
<code>void close()</code>	Закрывает поток
<code>int available()</code>	Возвращает количество доступных на данный момент символов для чтения из потока
<code>long skip(long n)</code>	Пропускает указанное количество символов из потока

boolean markSupported()	Проверка на возможность повторного чтения из потока
void mark(int limit)	Устанавливает метку для последующего повторного чтения; limit – размер буфера для операции повторного чтения
void reset()	Возвращает указатель потока на предварительно установленную метку. Дальнейшие вызовы метода read() будут снова возвращать данные, начиная с заданной метки

Класс Writer

Представляет абстрактный выходной поток символов и является предком для всех выходных символьных потоков.

Конструктор

Writer()	Создает выходной символьный поток
----------	-----------------------------------

Методы (таблица 35)

Таблица 35

Название	Функционал
abstract void write(int n) throws IOException	Записывает очередной символ Unicode в выходной поток. Значащими являются 16 младших битов, старшие — игнорируются. При ошибке ввода-вывода генерируется исключение. Подклассы должны обеспечить реализацию данного метода
void write(char[] buf)	Записывает в поток данные из буфера
void write(char[] buf, int offset, int len)	Записывает в поток len символов из буфера, начиная со смещения offset
void close()	Закрывает поток
void flush()	Заставляет освободить возможный буфер потока, отправляя на запись все записанные в него данные

В пакете java.io существуют специализированные классы потоков, предназначенные для работы с различными типами источников и получателей данных. Эти потоки делятся на две основные категории: байтовые и символьные, в зависимости от типа данных, с которыми они работают.

Таблица, показывающая соответствие между типом данных и классами потоков для чтения (входные) и записи (выходные), – таблица 36.

При создании этих потоков, в конструктор передается ссылка на источник данных (например, файл, массив) для входных потоков или на получатель данных

для выходных потоков. Для чтения данных используются методы `read()`, а для записи - `write()`.

Таблица 36

Тип данных	Байтовые потоки (чтение)	Байтовые потоки (запись)	Символьные потоки (чтение)	Символьные потоки (запись)
Файл	<code>FileInputStream</code>	<code>FileOutputStream</code>	<code>FileReader</code>	<code>FileWriter</code>
Массив	<code>ByteArrayInputStream</code>	<code>ByteArrayOutputStream</code>	<code>CharArrayReader</code>	<code>CharArrayWriter</code>
Строка	—	—	<code>StringReader</code>	<code>StringWriter</code>
Конвейер	<code>PipedInputStream</code>	<code>PipedOutputStream</code>	<code>PipedReader</code>	<code>PipedWriter</code>

Особый случай представляют собой конвейерные потоки (`PipedInputStream/PipedOutputStream` и `PipedReader/PipedWriter`). Они используются для организации канала связи между двумя потоками выполнения (`Thread`). Входной конвейерный поток получает данные из выходного конвейерного потока, и наоборот, создавая таким образом двунаправленную связь.

Пример чтения данных из файла:

```
FileReader f = new FileReader("myfile.txt");
char[] buffer = new char[512];
f.read(buffer);
f.close();
```

Преобразующие потоки

Java предоставляет механизм для расширения возможностей потоков ввода-вывода с помощью так называемых «преобразующих потоков». Эти потоки принимают в качестве аргумента другой поток и добавляют к нему функциональность, такую как буферизация (`BufferedInputStream/OutputStream/Reader/Writer`) для более эффективной работы с данными или преобразование кодировки (`InputStreamReader/OutputStreamWriter`) для работы с текстом в различных кодировках. Буферизованные потоки позволяют читать и записывать данные большими блоками, а потоки преобразования кодировки обеспечивают корректную интерпретацию символов. Пример использования (листинг 30):

Листинг 30

```
// Вариант 1
FileInputStream fis = new FileInputStream("file.txt"); InputStreamReader isr = new In-
putStreamReader(fis); BufferedReader br = new BufferedReader(isr);
br.readLine();

// Вариант 2
BufferedReader br = new BufferedReader(
    new InputStreamReader(
        new FileInputStream("file.txt")));
br.readLine();
```

`fis` – поток байтов из файла `file.txt`.

`isr` – поток символов, преобразованный из байтового с учетом системной кодировки.

`br` – поток символов с поддержкой буферизации.

DataInputStream и DataOutputStream

Эти классы служат для работы с двоичными данными. `DataOutputStream` позволяет записывать в поток данные примитивных типов (`boolean`, `int`, `double` и т.д.) в двоичном формате. `DataInputStream` предназначен для чтения этих двоичных данных из потока. Важно, чтобы данные читались в том же порядке и с использованием тех же типов, в котором они были записаны с помощью `DataOutputStream`, иначе чтение будет некорректным. Они предоставляют специализированные методы, такие как `readBoolean()`, `writeInt()`, `writeDouble()` и т.п., для работы с каждым типом данных.

PrintStream и PrintWriter

Эти классы используются для форматированного вывода текста в поток. Они предоставляют удобные методы `print()` и `println()`, которые могут принимать аргументы различных типов (примитивные типы, строки, объекты) и преобразовывать их в текстовое представление, используя системную кодировку. Метод `println()` автоматически добавляет символ новой строки в конце вывода. Это делает их удобными для восприятия информации при чтении.

Стандартные потоки ввода-вывода

Класс `java.lang.System` содержит 3 поля, представляющих собой стандартные консольные потоки (таблица 37).

Таблица 37

поле	класс	поток	по умолчанию
System.in	InputStream	стандартный поток ввода	клавиатура
System.out	PrintStream	стандартный поток вывода	окно терминала
System.err	PrintStream	стандартный поток ошибок	окно терминала

Имеется возможность перенаправлять данные потоки с помощью методов `System.setIn()`, `System.setOut()`, `System.setErr()`.

Пример чтения данных с клавиатуры и вывода в окно терминала:

```
BufferedReader br = new BufferedReader(
    new InputStreamReader(System.in)); String s =
br.readLine();
System.out.println("Введена строка : " + s);
System.out.println("Длина строки : " + s.length);
```

Пример чтения из файла:

```
try (FileInputStream fis = new FileInputStream("input.txt");
    InputStreamReader isr = new InputStreamReader(fis);
    BufferedReader reader = new BufferedReader(isr)) {

    String line;
    while ((line = reader.readLine()) != null) {
        System.out.println(line);
    }
} catch (IOException e) {
    e.printStackTrace();
}
```

В этом примере `FileInputStream` используется для чтения байтов из файла "input.txt". `InputStreamReader` преобразует байты в символы, а `BufferedReader` обеспечивает буферизованное чтение строк. Блок `try-with-resources` гарантирует автоматическое закрытие потоков.

Пример записи в файл:

```
try (FileOutputStream fos = new FileOutputStream("output.txt");
    OutputStreamWriter osw = new OutputStreamWriter(fos);
    BufferedWriter writer = new BufferedWriter(osw)) {
```

```
writer.write("Привет, мир!");  
writer.newLine();  
writer.write("Это пример записи в файл.");  
  
} catch (IOException e) {  
    e.printStackTrace();  
}
```

Здесь `FileOutputStream` используется для записи байтов в файл "output.txt". `OutputStreamWriter` преобразует символы в байты.

Тема 3.8

МНОГОПОТОЧНОЕ ПРОГРАММИРОВАНИЕ В ЯЗЫКЕ JAVA, ОСНОВНЫЕ ПОНЯТИЯ. СОЗДАНИЕ ПРОЦЕССОВ И ПОТОКОВ

Поток представляет собой независимую последовательность инструкций, выполняемую в рамках программы. Это один из способов реализации многозадачности, позволяющий программе одновременно выполнять несколько задач.

Примеры использования потоков:

- фоновое обновление и восстановление данных;
- ожидание и обработка сетевых данных;
- создание анимации;
- воспроизведение и обработка аудио.

Потоки vs. Процессы. В отличие от процессов, которые имеют собственное изолированное адресное пространство и обмениваются данными через механизмы межпроцессного взаимодействия, потоки одного процесса разделяют общее адресное пространство. Это позволяет потокам обмениваться данными через общие переменные. Переключение между потоками происходит быстрее и требует меньше ресурсов, чем переключение между процессами.

Реализация потоков в Java. Существует два основных способа создания потоков в Java:

1. Реализация интерфейса `java.lang.Runnable`: Класс реализует интерфейс `Runnable` и определяет метод `run()`, содержащий код, который должен выполняться в потоке. Затем экземпляр этого класса передается в конструктор класса `Thread`.

2. Наследование от класса `java.lang.Thread`: Класс наследуется от класса `Thread` и переопределяет метод `run()`. Затем создается экземпляр этого класса.

Метод run() является ключевым методом потока, определяющим его функциональность.

Запуск потока. Для запуска потока используется метод start() класса Thread. Этот метод выделяет необходимые ресурсы, создает новый поток и запускает выполнение метода run() в этом потоке. Пример (листинг 31):

Листинг 31

```
public class MyThread2 extends Thread {
    @Override
    public void run() {
        // тело потока
    }
}
...
MyThread2 t = new ldap.devMyThread2();
t.start();
```

Состояния потока

Потоки (threads) проходят через различные состояния, определяемые виртуальной машиной Java (JVM). Эти состояния, представленные в перечислении Thread.State, отражают текущий статус потока в процессе его жизненного цикла.

Состояния потока:

- **NEW (Новый):** Поток создан, но еще не начал свое выполнение. Это происходит до вызова метода start().
- **RUNNABLE (Выполняемый):** Поток готов к выполнению. Он либо активно использует процессор, либо ожидает своей очереди на процессорное время. Диспетчер потоков (thread scheduler) распределяет процессорное время между потоками в этом состоянии. JVM не делает различий между активным выполнением и ожиданием процессорного времени. Количество одновременно выполняющихся потоков ограничено количеством доступных процессорных ядер.
- **BLOCKED (Заблокированный):** Поток ожидает освобождения монитора (блокировки) для доступа к синхронизированному ресурсу.
- **WAITING (Ожидающий):** Поток приостановил свое выполнение и ждет, пока другой поток выполнит определенное действие, необходимое для его дальнейшей работы.
- **TIMED_WAITING (Временно ожидающий):** Аналогично WAITING, но поток ждет в течение определенного времени. Если за указанный период времени ожидаемое действие не произойдет, поток возобновит свое выполнение.

- **TERMINATED (Завершившийся):** Метод `run()` потока завершил свое выполнение, и поток больше не активен.

Приоритеты потоков. Каждому потоку можно назначить приоритет, который влияет на порядок его выполнения. В классе `java.lang.Thread` определены константы для представления приоритетов:

- `MIN_PRIORITY`: Минимальный приоритет.
- `NORM_PRIORITY`: Нормальный (по умолчанию) приоритет.
- `MAX_PRIORITY`: Максимальный приоритет.

При создании потоку присваивается приоритет `NORM_PRIORITY`. Изменить приоритет можно с помощью метода `setPriority(int)`. Потоки с более высоким приоритетом, как правило, получают больше процессорного времени.

Группы потоков (ThreadGroup). Класс `java.lang.ThreadGroup` позволяет объединять потоки в группы для упрощения управления ими. С группами потоков можно выполнять те же операции, что и с отдельными потоками, например, запуск, остановку и изменение приоритета. Группы потоков могут быть организованы в иерархическую структуру, где одна группа является родительской для других.

Методы класса `java.lang.Thread`. Методы класса `Thread` можно разделить на две категории: методы, возвращающие информацию о потоке, и методы, изменяющие его свойства.

Методы, возвращающие информацию:

- `activeCount()`: Возвращает количество активных потоков в группе.
- `currentThread()`: Возвращает ссылку на текущий выполняющийся поток.
- `getName()`: Возвращает имя потока.
- `getPriority()`: Возвращает приоритет потока.
- `getThreadGroup()`: Возвращает ссылку на группу, к которой принадлежит поток.
- `interrupted()`: (Статический метод) Возвращает `true`, если текущий поток был прерван.
- `isInterrupted()`: Возвращает `true`, если указанный поток был прерван.
- `isAlive()`: Возвращает `true`, если поток активен (не завершен).
- `isDaemon()`: Возвращает `true`, если поток является демоном.

Взаимодействие и синхронизация потоков

Для управления потоками и их взаимодействием, предусмотрены следующие механизмы:

- **Приостановка потока:** Метод `Thread.sleep(long ms)` позволяет потоку «уснуть» на указанное количество миллисекунд. Во время сна поток переходит в состояние `TIMED_WAITING`, освобождая процессорное время для других потоков.

– **Ожидание завершения другого потока:** Метод `join()` позволяет одному потоку дожидаться завершения работы другого потока. Поток, вызвавший `join()`, будет заблокирован до тех пор, пока поток, к которому был применен `join()`, не завершит свое выполнение.

– **Прерывание потока:** Метод `interrupt()` посылает потоку сигнал о прерывании. Этот сигнал устанавливает внутренний флаг прерывания потока. Поток может периодически проверять этот флаг с помощью метода `isInterrupted()`. Если флаг установлен, поток должен корректно завершить свою работу. Методы, такие как `sleep()` и `join()`, при получении сигнала прерывания выбрасывают исключение `InterruptedException`, которое можно обработать для завершения работы потока.

– **Синхронизация потоков (избежание гонок данных):** Когда несколько потоков одновременно работают с общими данными, может возникнуть ситуация «гонки данных». Это происходит, когда потоки читают и изменяют данные в непредсказуемом порядке, что приводит к некорректным результатам. Чтобы избежать этого, используется синхронизация.

- **synchronized:** Ключевое слово `synchronized` используется для защиты критических участков кода, где происходит доступ к общим данным. Существуют синхронизированные блоки и синхронизированные методы.
- **Мониторы:** Каждый объект в Java имеет связанный с ним монитор. При использовании синхронизированного блока, монитор объекта, указанного в качестве параметра блока, используется для блокировки доступа к этому блоку кода. Только один поток может владеть монитором в любой момент времени, что гарантирует эксклюзивный доступ к защищенным данным.

Пример использования синхронизированного блока:

```
synchronized (someObject) {  
    //здесь синхронизированный код  
}
```

Когда поток входит в `synchronized` блок, он получает эксклюзивный доступ к монитору объекта, указанного в `synchronized` конструкции (`synchronized(someObject)`). Пока поток владеет этим монитором, другие потоки, пытающиеся войти в этот же `synchronized` блок, блокируются и переходят в состояние `BLOCKED`. После завершения работы `synchronized` блока, поток освобождает монитор, позволяя одному из ожидающих потоков захватить его и продолжить

выполнение. Важно отметить, что поток, уже владеющий монитором, может повторно входить в `synchronized` блоки, связанные с этим же монитором (реентерабельность).

Если несколько `synchronized` блоков используют один и тот же объект для синхронизации, то они становятся взаимоисключающими: только один поток может выполнять любой из этих блоков в любой момент времени. Однако, если блоки используют разные объекты для синхронизации, они могут выполняться параллельно.

`Synchronized` методы в Java эквивалентны `synchronized` блокам, где в качестве объекта синхронизации используется `this` (текущий объект) для нестатических методов и объект `Class` для статических методов.

Для межпоточного взаимодействия и сигнализации о событиях в Java используются методы `wait()`, `notify()`, и `notifyAll()` класса `Object`. Эти методы позволяют потокам координировать свою работу, например, чтобы один поток ждал, пока другой подготовит данные.

Рассмотрим пример с методами `put()` и `get()`, где один поток записывает данные, а другой читает. Синхронизация необходима для предотвращения гонок данных и обеспечения корректной последовательности операций: запись должна происходить до чтения, и чтение должно происходить после записи.

Методы `wait()`, `notify()`, и `notifyAll()` тесно связаны с монитором объекта и должны вызываться только внутри `synchronized` блоков или методов. Вызов `wait()` приводит к тому, что поток освобождает монитор объекта, добавляется в список ожидания этого объекта и переходит в состояние `WAITING` или `TIMED_WAITING`. Это позволяет другим потокам захватить монитор и выполнить критическую секцию.

В листинге 32 представлен класс `A`, который реализует механизм синхронизации для обмена данными между потоками. Он содержит:

- **flag (boolean):** Флаг, указывающий, доступно ли значение `value` для чтения (`true`) или записи (`false`). Он используется для координации между потоками, читающими и записывающими данные.
- **value (int):** Целочисленное значение, которое передается между потоками.
- **put(int i) (synchronized):** Метод для записи значения `i` в `value`. Он использует `wait()` и `notifyAll()` для предотвращения перезаписи значения, пока оно не будет прочитано другим потоком.
- **get() (synchronized):** Метод для чтения значения из `value`. Он также использует `wait()` и `notifyAll()` для предотвращения чтения значения, пока оно не было записано.

Листинг 32

```
class A {  
    boolean flag = false; int  
    value;  
    synchronized void put(int i) {  
        while(flag) { wait(); }  
        flag = true; value = i;  
        notifyAll();  
    }  
    synchronized int get() {  
        while(!flag) { wait(); }  
        flag = false; notifyAll();  
        return value;  
    }  
}
```

Объяснение механизма синхронизации

Класс А использует монитор (встроенный в каждый Java-объект) и методы `wait()`, `notify()`, и `notifyAll()` для управления доступом к переменной `value` из разных потоков. Важные моменты:

1. **Синхронизированные методы:** Ключевое слово `synchronized` гарантирует, что только один поток может выполнять методы `put()` или `get()` одновременно. Это предотвращает состояние гонки (*race condition*) и обеспечивает целостность данных.

2. **Ожидание (`wait()`):** Если поток пытается записать значение в `value` (через `put()`), когда `flag` уже `true` (значение еще не прочитано), или пытается прочитать значение из `value` (через `get()`), когда `flag` `false` (значение еще не записано), он переходит в состояние ожидания (`wait()`). При этом поток *освобождает* монитор объекта А, позволяя другим потокам войти в синхронизированные методы.

3. **Уведомление (`notify()` / `notifyAll()`):** Когда поток завершает запись значения (в `put()`) или чтение значения (в `get()`), он вызывает `notifyAll()`. Это «будит» все потоки, которые находятся в состоянии ожидания на мониторе объекта А. `notify()` будит только один случайный поток.

4. **Возобновление работы:** Пробужденные потоки начинают конкурировать за захват монитора объекта А. Как только один из них получает монитор, он выходит из состояния ожидания и продолжает выполнение с того места, где был приостановлен (после вызова `wait()`). Он проверяет условие (например,

flag), чтобы убедиться, что теперь может безопасно записать или прочитать значение.

Этот код реализует шаблон «производитель-потребитель» (producer-consumer) с одним буфером. Один или несколько потоков («производители») записывают данные в value (через put()), а один или несколько потоков («потребители») читают данные из value (через get()). Синхронизация гарантирует, что производители не перезапишут данные, пока они не будут прочитаны, а потребители не прочитают данные, пока они не будут записаны. wait() и notifyAll() являются ключевыми механизмами для эффективной координации между этими потоками, позволяя им «спать», когда нет работы, и «проснуться», когда появляется новая работа.

Модификатор volatile. Модификатор volatile, применяемый к переменной, гарантирует, что каждый поток будет работать с самым актуальным значением этой переменной, хранящимся в основной памяти. Это достигается за счет того, что потоки не кэшируют volatile переменные, а всегда читают и записывают их непосредственно в память. Кроме того, volatile обеспечивает атомарность операций чтения и записи, даже для типов long и double.

Тема 3.9

УПРАВЛЕНИЕ ПОТОКАМИ ПРИ МНОГОПОТОЧНОМ ПРОГРАММИРОВАНИИ В ЯЗЫКЕ JAVA. СИНХРОНИЗАЦИЯ

При управлении потоками особое внимание следует уделить синхронизации потоков и возможностям, предоставляемым пакетом java.util.concurrent.

В отличие от низкоуровневых подходов к параллельной обработке данных, пакет java.util.concurrent, появившийся в Java 5, предлагает более удобные и мощные инструменты для разработки параллельных приложений. Мы рассмотрим основные классы и интерфейсы этого пакета, которые можно условно разделить на шесть групп, начиная с группы **Исполнителей (Executors)**.

Далее будут рассмотрены следующие интерфейсы:

- **Callable<V>**: Аналогичен Runnable, но позволяет возвращать результат (V) и выбрасывать исключения. Основной метод – V call().
- **Future<V>**: Предназначен для получения результата асинхронной задачи.

Предоставляет методы:

- V get(): Блокирует текущий поток до получения результата задачи.
- V get(long, TimeUnit): Блокирует поток до получения результата или истечения заданного времени ожидания.

- `boolean cancel(boolean)`: Пытается отменить выполнение задачи. Параметр определяет, следует ли прерывать выполняющуюся задачу.
- `boolean isDone()`: Возвращает `true`, если задача завершена (успешно, отменена или с ошибкой).
- `boolean isCancelled()`: Возвращает `true`, если задача была отменена.

– **Delayed**: Позволяет указать задержку перед началом выполнения задачи. Содержит метод `long getDelay(TimeUnit)`, возвращающий оставшееся время задержки.

– **RunnableFuture**: Комбинирует функциональность `Runnable` и `Future`, позволяя запустить задачу и получить ее результат.

– **ScheduledFuture**: Объединяет `Delayed` и `Future`, позволяя запланировать задачу на определенное время и получить ее результат.

– **RunnableScheduledFuture**: Сочетает в себе `Runnable` и `ScheduledFuture`, позволяя запланировать задачу на определенное время и получить ее результат. Дополнительно предоставляет метод `boolean isPeriodic()`, позволяющий создавать периодически повторяющиеся задачи.

Вот описание основных интерфейсов и классов, используемых для управления параллельным выполнением задач:

– **Executor**: Это базовый интерфейс, который абстрагирует процесс запуска задач от их фактического выполнения. Он принимает объекты `Runnable` и асинхронно запускает их через метод `execute(Runnable)`.

– **ExecutorService**: Расширяет `Executor`, предоставляя дополнительные возможности для управления жизненным циклом задач и получения результатов. Ключевые методы:

- `submit(Callable<T> task)`: Асинхронно запускает задачу, возвращая `Future<T>`, который позволяет получить результат выполнения.
- `invokeAll(Collection<Callable> tasks)`: Запускает несколько задач одновременно и возвращает список `Future<T>` для получения результатов после завершения всех задач.

– **ScheduledExecutorService**: Расширяет `ExecutorService`, добавляя возможность планировать задачи для выполнения с задержкой или периодически, используя метод `schedule()`, который возвращает `ScheduledFuture`.

– Класс **Executors** предоставляет набор статических методов для создания стандартных реализаций `ExecutorService` и `ScheduledExecutorService`, основанных на пулах потоков:

- **FixedThreadPool**: Пул с фиксированным количеством потоков, использующий общую неограниченную очередь задач.
- **WorkStealingPool**: Пул с динамически изменяемым количеством потоков и несколькими очередями задач. Обычно количество потоков соответствует числу доступных процессоров.
- **CachedThreadPool**: Пул, который повторно использует существующие потоки, если они доступны. Если поток простаивает более минуты, он уничтожается. Новые потоки создаются по мере необходимости. Подходит для обработки большого количества коротких задач.
- **ScheduledThreadPool**: Пул, предназначенный для выполнения задач с задержкой или периодически.
- **SingleThreadExecutor**: Однопоточный исполнитель с неограниченной очередью. Гарантирует последовательное выполнение задач в порядке их поступления.
- **SingleThreadScheduledExecutor**: Однопоточный исполнитель, позволяющий планировать задачи с задержкой.

Кроме того, Java предоставляет механизм **fork-join** для решения задач, которые можно рекурсивно разделить на подзадачи. Он использует классы `ForkJoinPool` и `ForkJoinTask`. Основная идея заключается в том, что поток выполняет задачу самостоятельно, если она достаточно мала. В противном случае задача делится на подзадачи, которые передаются другим потокам. Для реализации алгоритма **fork-join** обычно используются подклассы `ForkJoinTask`: `RecursiveAction` (для задач, не возвращающих результат) и `RecursiveTask` (для задач, возвращающих результат). Оба класса требуют реализации абстрактного метода `compute()`, в котором определяется логика выполнения задачи или ее разделения на подзадачи. Например, для вычисления чисел Фибоначчи – листинг 33:

Для выполнения задач параллельно с использованием пула потоков `ForkJoinPool`, создается экземпляр класса `ForkJoinPool`. Затем вызывается метод `invoke()`, которому передается задача, представленная классом, наследующим от `ForkJoinTask`.

Для разделения процесса отправки задач на выполнение и получения их результатов используется интерфейс `CompletionService`. Он предоставляет следующие методы:

- `submit(Callable<T>)`: отправляет задачу на выполнение и возвращает объект `Future`, который позволяет получить результат задачи позже.

– `poll()`: возвращает результат первой завершенной задачи, если такая есть. Если завершенных задач нет, возвращает `null`.

– `take()`: возвращает результат первой завершенной задачи. Если завершенных задач пока нет, метод будет ждать, пока одна из задач не завершится.

Листинг 33

```
public class FiboTask extends RecursiveTask<Integer> {  
    final int n;  
    public FiboTask(int number) { n = number; }  
    Integer compute() {  
        if (n < 2) { return n; }  
        FiboTask f1 = new FiboTask(n - 1);  
        f1.fork();  
        // выдача подзадачи другому потоку  
        FiboTask f2 = new FiboTask(n - 2);  
        return f2.compute() + f1.join();  
    }  
}
```

Очереди (Queues)

Пакет `java.util.concurrent` предоставляет набор реализаций для работы с очередями, предназначенных для многопоточного программирования. Эти реализации обеспечивают потокобезопасность и различные стратегии блокировки.

Вот основные типы очередей, доступные в этом пакете:

– **`ConcurrentLinkedQueue`**: Потокобезопасная очередь, построенная на основе связанного списка. Оптимальна для сценариев, где важна высокая производительность в многопоточной среде.

– **`ConcurrentLinkedDeque`**: Потокобезопасная двусторонняя очередь (дек), также основанная на связанном списке. Позволяет добавлять и удалять элементы с обоих концов.

– **`BlockingQueue`**: Интерфейс, определяющий поведение блокирующей очереди. В дополнение к стандартным операциям с очередями (таким как `add`, `remove`, `element`, `offer`, `poll`, `peek`), которые могут выбрасывать исключения или возвращать `false/null` при неудаче, `BlockingQueue` предоставляет методы, которые блокируют вызывающий поток, если очередь полна (при добавлении) или пуста (при извлечении). Существуют две группы таких методов:

- `put(e)` и `take()`: Блокируют поток до тех пор, пока не станет возможным добавить элемент в очередь или извлечь элемент из очереди, соответственно.
- `offer(e, time, unit)` и `poll(time, unit)`: Блокируют поток на указанное время, если добавление или извлечение элемента невозможно.

– **Реализации `BlockingQueue`:**

- `LinkedBlockingQueue` и `LinkedBlockingDeque`: Блокирующие очереди (одно- и двусторонние), реализованные на основе связного списка.
- `ArrayBlockingQueue`: Блокирующая очередь, использующая кольцевой буфер (массив) для хранения элементов.
- `SynchronousQueue`: Специальная блокирующая очередь, в которой каждая операция добавления должна дожидаться соответствующей операции извлечения, и наоборот. Фактически, она передает элемент напрямую от одного потока к другому.
- `PriorityBlockingQueue`: Блокирующая очередь, в которой элементы упорядочены в соответствии с их приоритетом, определяемым компаратором.
- `DelayQueue`: Блокирующая очередь, в которой элементы могут быть извлечены только после истечения определенной задержки, связанной с каждым элементом.

– **`TransferQueue` и `LinkedTransferQueue`:** `TransferQueue` расширяет интерфейс `BlockingQueue`, добавляя метод `transfer()`. Этот метод блокирует поток, добавляющий элемент, до тех пор, пока другой поток не извлечет этот элемент из очереди. `LinkedTransferQueue` является реализацией `TransferQueue` на основе связного списка.

Пример конкурентного доступа к очереди (`LinkedBlockingQueue`) – листинг 34.

Листинг 34

```
import java.util.concurrent.BlockingQueue;
import java.util.concurrent.LinkedBlockingQueue;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.TimeUnit;

public class LinkedBlockingQueueExample {

    public static void main(String[] args) throws InterruptedException {
        BlockingQueue<Integer> queue = new LinkedBlockingQueue<>(5);
```

```

ExecutorService executor = Executors.newFixedThreadPool(2);

executor.submit(() -> {
    for (int i = 0; i < 10; i++) {
        try {
            queue.put(i);
            System.out.println("Добавлено: " + i);
            Thread.sleep(100);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
});

executor.submit(() -> {
    while (true) {
        try {
            Integer element = queue.take();
            System.out.println("Извлечено: " + element);
            Thread.sleep(150);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
});

executor.shutdown();
executor.awaitTermination(1, TimeUnit.MINUTES);
}
}

```

В этом примере два параллельно выполняемых потока (запускаются по `executor.submit()`) имеют одновременный доступ к очереди (`queue`), первый в цикле `for` добавляет туда данные, второй – в `while` извлекает их.

Потокобезопасные коллекции (Concurrent Collections)

Помимо потокобезопасных очередей, в пакете представлены и другие коллекции, предназначенные для работы в многопоточной среде. В отличие от традиционных синхронизированных коллекций (создаваемых, например, с помощью `Collections.synchronizedList`), которые блокируют всю коллекцию при любой операции, потокобезопасные коллекции используют более тонкую систему блокировок. Это позволяет нескольким потокам одновременно работать с коллекцией, значительно повышая производительность при параллельном выполнении задач. В этот набор входят:

- `ConcurrentHashMap`: Обеспечивает высокую степень параллелизма, позволяя множеству потоков читать данные одновременно, а также ограниченному числу потоков параллельно изменять данные.
- `ConcurrentSkipListMap`: Ассоциативный массив, реализованный с использованием структуры данных «список с пропусками», что обеспечивает потокобезопасность.
- `ConcurrentSkipListSet`: Множество, также реализованное на основе списков с пропусками для обеспечения потокобезопасности.
- `CopyOnWriteArrayList`: Динамический массив, который при каждой операции изменения создает новую копию массива. Это гарантирует потокобезопасность, но может быть менее эффективным при частых изменениях.
- `CopyOnWriteArraySet`: Множество, реализованное на основе `CopyOnWriteArrayList`.

Синхронизаторы (Synchronizers)

Для управления взаимодействием между потоками существуют специальные инструменты, называемые синхронизаторами. Они предоставляют готовые решения для организации различных схем синхронизации, избавляя от необходимости писать сложные конструкции с `wait()` и `notify()`. Вот основные из них:

- **Semaphore**: Семафор контролирует доступ к общему ресурсу, ограничивая количество потоков, которые могут одновременно им пользоваться. Он работает с «разрешениями». `acquire()` пытается получить разрешение, блокируя поток, если их нет. `tryAcquire()` делает то же самое, но не блокируется, а возвращает `false`, если разрешение недоступно. `release()` возвращает разрешение, потенциально разблокируя ожидающий поток. Представьте его как ограниченное количество ключей от комнаты: потоки должны получить ключ, чтобы войти, и вернуть его, когда закончат.
- **Exchanger**: Этот класс позволяет двум потокам обмениваться данными. Каждый поток вызывает `exchange(V)`, передавая свои данные. Метод блокируется, пока второй поток не вызовет `exchange(V)`, после чего происходит обмен данными, и оба потока продолжают работу. Это как встреча двух курьеров, которые обмениваются пакетами.
- **CountDownLatch**: Это одноразовый триггер. При создании ему задается начальное значение счетчика. Потоки, вызывающие `await()`, блокируются до тех пор, пока счетчик не достигнет нуля (после вызова `countDown()` нужное количество раз). После этого все ожидающие потоки освобождаются. Представьте это как стартовую линию: гонка не начнется, пока все участники не будут готовы.

– **CyclicBarrier**: Циклический барьер похож на `CountDownLatch`, но его можно использовать несколько раз. При создании ему задается размер (количество потоков). Когда поток вызывает `await()`, он блокируется, пока не наберется заданное количество потоков. После этого все потоки освобождаются, и барьер можно использовать снова. Это как командная работа, где все должны дождаться друг друга, прежде чем перейти к следующему этапу.

– **Phaser**: `Phaser` – более гибкий синхронизатор, сочетающий в себе возможности `CountDownLatch` и `CyclicBarrier`. Он позволяет динамически регистрировать и deregистрировать потоки, участвующие в синхронизации. Основные методы:

- `register()`: увеличивает количество потоков, необходимых для преодоления барьера.
- `arriveAndAwaitAdvance()`: поток прибывает к барьеру и ждет, пока все остальные потоки не придут.
- `arriveAndDeregister()`: поток прибывает к барьеру и отменяет свое дальнейшее участие.
- `arrive()`: поток прибывает к барьеру, но не ждет его открытия.

`Phaser` позволяет реализовывать сложные сценарии синхронизации, например, когда количество участвующих потоков меняется в процессе выполнения. Он подходит для задач, где потоки работают поэтапно, и на каждом этапе нужно дождаться завершения работы всех участников.

Блокировки (Locks)

Для управления доступом к общим ресурсам в многопоточной среде, используется пакет `java.util.concurrent.locks`. Этот пакет предоставляет более гибкие и мощные инструменты, чем стандартные `synchronized` методы.

Основные компоненты пакета `java.util.concurrent.locks`:

– **Lock (Интерфейс)**: Представляет собой механизм блокировки, альтернативный `synchronized`. `Lock` предлагает расширенные возможности, такие как:

- Попытка захвата блокировки без блокировки текущего потока.
- Прерываемая блокировка (возможность прервать ожидание блокировки).
- Блокировка с ограничением по времени (таймаут).

Основные методы `Lock`:

- `lock()`: Захватывает блокировку. Если она свободна, текущий поток получает ее. Иначе поток переходит в состояние ожидания, пока блокировка не будет освобождена.
- `unlock()`: Освобождает блокировку.

- `lockInterruptibly()`: Захватывает блокировку, но позволяет прервать поток, ожидающий блокировку, с помощью `InterruptedException`.
- `tryLock()`: Пытается захватить блокировку немедленно. Возвращает `true`, если блокировка получена, и `false` в противном случае.
- `tryLock(time, unit)`: Пытается захватить блокировку в течение заданного времени. Если блокировка не получена за это время, поток прекращает ожидание.
- `newCondition()`: Создает объект `Condition`, связанный с этой блокировкой.

– **Condition (Интерфейс)**: Предоставляет механизм ожидания и уведомления, аналогичный `wait()` и `notify()`, но с большей гибкостью. Позволяет создавать несколько условий ожидания для одной блокировки.

Основные методы `Condition`:

- `await()`: Переводит текущий поток в состояние ожидания до получения сигнала или прерывания.
- `await(time, unit)`: Переводит текущий поток в состояние ожидания до получения сигнала, прерывания или истечения заданного времени.
- `awaitUntil(Date)`: Переводит текущий поток в состояние ожидания до получения сигнала, прерывания или наступления указанной даты.
- `awaitUninterruptibly()`: Переводит текущий поток в состояние ожидания до получения сигнала, не реагируя на прерывания.
- `signal()`: Пробуждает один поток, ожидающий на этом условии.
- `signalAll()`: Пробуждает все потоки, ожидающие на этом условии.

– **ReadWriteLock (Интерфейс)**: Определяет пару связанных блокировок: одна для чтения (разделяемая) и одна для записи (исключительная). Позволяет нескольким потокам читать данные одновременно, но только один поток может записывать данные в любой момент времени.

Основные методы `ReadWriteLock`:

- `readLock()`: Возвращает блокировку для чтения.
- `writeLock()`: Возвращает блокировку для записи.

– **ReentrantLock (Класс)**: Реализация интерфейса `Lock`, позволяющая потоку повторно захватывать уже удерживаемую им блокировку. Для освобождения блокировки необходимо вызвать `unlock()` столько раз, сколько раз она была захвачена.

– Класс `ReentrantReadWriteLock` реализует интерфейс `ReadWriteLock` и предоставляет следующие возможности:

- **Режим работы по умолчанию:** Используется «нечестный» режим, при котором нет гарантии, что потоки получают блокировку в порядке их запроса. Это обеспечивает более высокую производительность.
- **Опциональный «честный» режим:** Можно включить режим, в котором блокировки предоставляются потокам, ожидающим дольше всего. Однако, это может снизить общую производительность.
- **Блокировка на чтение:** Разрешается только тогда, когда нет потоков, ожидающих получения блокировки на запись. Это гарантирует, что данные не будут изменены во время чтения.
- **Повышение привилегий записи:** Поток, уже удерживающий блокировку на запись, может получить блокировку на чтение без необходимости повторного запроса.
- **Запрет понижения привилегий:** Поток, удерживающий блокировку на чтение, не может получить блокировку на запись, пока не освободит блокировку на чтение. Это предотвращает взаимоблокировки.
- **Условия для записи:** Блокировка на запись поддерживает использование объектов Condition для более сложной синхронизации. Блокировка на чтение такую возможность не предоставляет.

Пример использования ReentrantLock:

Листинг 35

```
import java.util.concurrent.locks.ReentrantLock;

class Counter {
    private final ReentrantLock lock = new ReentrantLock();
    private int count = 0;

    public void increment() {
        lock.lock(); // Захватываем блокировку
        try {
            count++;
            System.out.println(Thread.currentThread().getName() + " incremented count to: "
+ count);
        } finally {
            lock.unlock(); // Обязательно освобождаем блокировку в блоке finally
        }
    }
}
```

```

    public int getCount() {
        lock.lock();
        try {
            return count;
        } finally {
            lock.unlock();
        }
    }
}

class Worker extends Thread {
    private final Counter counter;

    public Worker(Counter counter) {
        this.counter = counter;
    }

    @Override

    public void run() {
        for (int i = 0; i < 5; i++) {
            counter.increment();
            try {
                Thread.sleep(100); // Имитация работы
            } catch (InterruptedException e) {
                Thread.currentThread().interrupt();
            }
        }
    }
}

public class Main {
    public static void main(String[] args) {
        Counter counter = new Counter();
        Worker worker1 = new Worker(counter);
        Worker worker2 = new Worker(counter);

        worker1.start();
        worker2.start();

        try {
            worker1.join();
            worker2.join();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
    }
}

```

```
}

    System.out.println("Final count: " + counter.getCount());
}
}
```

В примере выше присутствует три класса, **Counter**, в котором, в методе **increment()**, увеличивается переменная **count**. Этот метод будет вызываться двумя экземплярами класса **Worker**, которые и будут конкурировать за вызов этого метода. По этой причине весь код метода **increment()** заключен в блокировку повторного входа (**lock.lock()**; безопасный код **lock.unlock()**). Класс **Worker** (является наследником **Thread**, т.е. может выполнять код в параллельном потоке) принимает в конструкторе экземпляр класса **Counter** и в цикле отдельного потока выполнения вызывает **counter.increment()**. В третьем классе – **Main**, создается 2 экземпляра класса **Worker**, но только один экземпляр класса **Counter**, который и передается обоим **Worker**-ам. Вызов метода **join()** после запуска обоих параллельных потоков (**Worker**) вынуждает ожидать завершения работы обоих потоков (**worker1/worker2**) для того, чтобы корректно вывести результирующее значение переменной **count**. Метод **getCount()** также заключен в блокировку повторного входа, причем через тот же экземпляр класса **ReentrantLock** (переменная **lock**), что и метод **increment()**, что не позволит вызывать код метода **getCount()** не только разным потокам одновременно, но и также не позволит его вызывать одновременно с методом **increment()** (причина как раз в одном и том же экземпляре переменной **lock**, используемой в обоих методах для блокировки).

Результат работы программы приведен ниже:

```
Thread-0 incremented count to: 1
Thread-1 incremented count to: 2
Thread-0 incremented count to: 3
Thread-1 incremented count to: 4
Thread-0 incremented count to: 5
Thread-1 incremented count to: 6
Thread-0 incremented count to: 7
Thread-1 incremented count to: 8
Thread-0 incremented count to: 9
Thread-1 incremented count to: 10
Final count: 10
```


Виртуальные потоки

Виртуальные потоки (Virtual Threads) в Java призваны оптимизировать использование ресурсов, позволяя более эффективно использовать поток операционной системы. Стабильно виртуальные потоки поддерживаются с версии Java 21.

В отличие от традиционных потоков, виртуальные потоки сами по себе не выполняются напрямую. Они работают поверх так называемых «потоков-носителей» (Carrier Threads), которые являются обычными потоками платформы. Для работы виртуальных потоков используется пул потоков-носителей, размер которого по умолчанию составляет 256. При этом, по умолчанию, одновременно может выполняться столько виртуальных потоков, сколько ядер процессора доступно системе.

Процесс выполнения виртуального потока выглядит следующим образом:

1. **Создание и запуск:** Разработчик создает и запускает виртуальный поток в своем коде.

2. **Назначение потока-носителя:** Планировщик виртуальной машины Java (JVM) выбирает свободный поток-носитель из пула и назначает ему виртуальный поток для выполнения.

3. **Выполнение в потоке-носителе:** Виртуальный поток исполняется внутри потока-носителя до тех пор, пока не завершится или не столкнется с блокирующей операцией (например, ожидание ввода-вывода).

4. **Переключение при блокировке:** Если виртуальный поток сталкивается с блокирующей операцией, планировщик JVM приостанавливает его выполнение, освобождает поток-носитель и назначает ему другой виртуальный поток, готовый к выполнению.

5. **Проверка готовности:** Планировщик JVM регулярно проверяет, может ли приостановленный виртуальный поток продолжить свою работу (например, завершилось ожидание ввода-вывода).

6. **Возобновление выполнения:** Как только виртуальный поток готов к продолжению, планировщик JVM назначает его свободному потоку-носителю, и выполнение виртуального потока возобновляется до завершения или до следующей блокирующей операции.

Важные моменты:

- Виртуальные потоки используют потоки-носители для фактического выполнения.

- Планировщик JVM управляет распределением виртуальных потоков между потоками-носителями.

– Блокирующие операции приводят к приостановке виртуального потока и освобождению потока-носителя для других задач.

– Такой подход позволяет эффективно использовать ресурсы, поскольку потоки-носители не простаивают в ожидании завершения блокирующих операций виртуальных потоков.

Более подробно виртуальные потоки здесь рассматриваться не будут.

Тема 3.10

СЕТЕВОЕ ВЗАИМОДЕЙСТВИЕ

Сетевое взаимодействие обеспечивается пакетом `java.net`, который предоставляет инструменты для работы с сетью на разных уровнях абстракции. Эти инструменты можно условно разделить на две группы:

– **низкоуровневые средства:** Они позволяют напрямую взаимодействовать с сетевыми адресами, сокетами и сетевыми интерфейсами, предоставляя детальный контроль над сетевым соединением;

– **высокоуровневые средства:** Они упрощают работу с сетевыми ресурсами, предоставляя абстракции, такие как `URI`, `URL` и сетевые соединения.

Пакет `java.net` позволяет реализовать передачу данных по различным протоколам, включая `TCP`, `UDP` и `HTTP`. `TCP` и `UDP` – это протоколы транспортного уровня, каждый из которых имеет свои особенности (таблица 38).

Таблица 38

	TCP	UDP
Установление соединения	Требуется предварительное соединение	Не требует предварительного соединения
Подтверждение доставки	Гарантирует доставку сообщений	Не гарантирует доставку сообщений

Таким образом, `TCP` обеспечивает более надежную передачу данных, в то время как `UDP` обеспечивает более высокую скорость передачи, жертвуя надежностью.

Работа с адресами

Для работы с сетевыми адресами, такими как IP-адреса, используется класс `InetAddress`. IP-адреса, идентифицирующие хосты в сети, бывают двух основных

типов: IPv4 и IPv6. IPv4 представляется в виде четырех чисел (от 0 до 255), разделенных точками, а IPv6 – в виде восьми шестнадцатеричных чисел, разделенных двоеточиями.

Поскольку людям удобнее работать с текстовыми именами хостов, а не с числовыми IP-адресами, существует служба DNS, которая преобразует имена хостов в IP-адреса и наоборот.

Класс `InetAddress` в Java позволяет представлять IP-адреса и взаимодействовать с DNS. Хотя у него есть подклассы `Inet4Address` и `Inet6Address` для конкретных типов IP-адресов, обычно достаточно использовать базовый класс `InetAddress`. Объект `InetAddress` содержит как IP-адрес, так и имя хоста.

Для создания экземпляров `InetAddress` используются статические методы:

- `getLocalHost()`: Получает информацию о локальном компьютере.
- `getByAddress(String host, byte[] addr)`: Создает `InetAddress` по заданному IP-адресу (в виде массива байт) и имени хоста. Важно отметить, что проверка соответствия имени и адреса не производится.
- `getByAddress(byte[] addr)`: Создает `InetAddress` по заданному IP-адресу (в виде массива байт).
- `getByName(String host)`: Получает `InetAddress` по имени хоста, используя DNS для разрешения имени в IP-адрес.
- `getAllByName(String host)`: Получает массив всех IP-адресов, связанных с заданным именем хоста, также используя DNS.

Основные методы класса `InetAddress` для получения информации:

- `getAddress()`: Возвращает IP-адрес в виде массива байт.
- `getHostName()`: Возвращает имя хоста.

В целом, класс `InetAddress` предоставляет Java-программистам инструменты для работы с IP-адресами, именами хостов и DNS.

Передача данных по протоколу TCP

Для работы с TCP-соединениями ключевым элементом является класс `java.net.Socket`:

- **Создание соединения:** `Socket(InetAddress address, int port)` устанавливает TCP-соединение с указанным IP-адресом и портом.
- **Работа с данными:**
 - `getInputStream()` предоставляет возможность чтения данных, поступающих через сокет.
 - `getOutputStream()` позволяет отправлять данные через сокет.

- **Управление временем ожидания:** `setSoTimeout(int ms)` задает максимальное время (в миллисекундах) ожидания при чтении данных из сокета. Если данные не поступают за это время, возникает исключение `SocketTimeoutException`.

- **Завершение работы:** `close()` закрывает сокет и освобождает связанные ресурсы.

Для создания серверной части, которая принимает входящие соединения и обрабатывает запросы, используется класс **ServerSocket**.

Конструктор класса **ServerSocket**:

- `ServerSocket(int port)`: Инициализирует серверный сокет, который начинает прослушивать указанный порт для входящих соединений.

Методы класса **Socket** (обычно используются после установления соединения через **ServerSocket**):

- `Socket accept()`: Блокирует выполнение программы и ждет, пока клиент не попытается подключиться к серверу. Когда клиент подключается, метод возвращает новый объект `Socket`, представляющий соединение с этим клиентом.

- `void setSoTimeout(int ms)`: Задает максимальное время (в миллисекундах), в течение которого метод `accept()` будет ждать подключения клиента. Если в течение этого времени соединение не будет установлено, будет выброшено исключение `SocketTimeoutException`. Это позволяет избежать «зависания» программы в ожидании соединения.

- `void close()`: Закрывает сокет, освобождая ресурсы и завершая соединение. Важно закрывать сокеты после завершения работы с ними.

Другими словами, для создания сетевого сервера используется класс `ServerSocket`. Конструктор `ServerSocket(int port)` создает сервер, который «слушает» определенный порт. Метод `accept()` ждет, пока клиент подключится к этому порту. После подключения `accept()` возвращает объект `Socket`, через который сервер может общаться с клиентом. `setSoTimeout()` позволяет установить лимит времени ожидания подключения, чтобы избежать бесконечного ожидания. `close()` закрывает сокет, завершая соединение.

Последовательность создания TCP-соединения на стороне клиента

Клиентская сторона

Процесс установления TCP-соединения на стороне клиента включает следующие этапы:

1. **Определение адреса сервера:** Сначала необходимо получить объект `InetAddress`, представляющий IP-адрес сервера, к которому клиент хочет подключиться. Это может быть локальный адрес (как в примере) или адрес удаленного сервера.

2. **Создание сокета:** Создается объект `Socket`, который связывается с полученным IP-адресом и портом сервера. Этот шаг инициирует попытку установления соединения.

3. **Получение потоков ввода/вывода:** После успешного установления соединения из сокета извлекаются потоки `InputStream` и `OutputStream`. `InputStream` используется для чтения данных, отправленных сервером, а `OutputStream` – для отправки данных на сервер.

4. **Обмен данными:** Через полученные потоки осуществляется обмен данными между клиентом и сервером. Клиент читает данные из `InputStream` и записывает данные в `OutputStream`.

5. **Закрытие потоков:** После завершения обмена данными, необходимо закрыть потоки `InputStream` и `OutputStream`, чтобы освободить ресурсы.

6. **Закрытие сокета:** В завершение, необходимо закрыть сокет `Socket`, чтобы разорвать соединение с сервером и освободить системные ресурсы.

Серверная сторона

Процесс установления TCP-соединения на стороне сервера включает следующие этапы:

1. **Создание `ServerSocket`:** Создается объект `ServerSocket`, который «слушает» входящие соединения на определенном порту. Этот сокет отвечает за прием новых подключений.

2. **Принятие соединения:** Сервер ожидает подключения от клиента с помощью метода `accept()` объекта `ServerSocket`. Этот метод блокирует выполнение программы до тех пор, пока не будет установлено соединение. После установления соединения метод `accept()` возвращает объект `Socket`, представляющий соединение с конкретным клиентом.

3. **Получение потоков ввода/вывода:** Из полученного сокета `Socket` извлекаются потоки `InputStream` и `OutputStream`. `InputStream` используется для чтения данных, отправленных клиентом, а `OutputStream` – для отправки данных клиенту.

4. **Обмен данными:** Через полученные потоки осуществляется обмен данными между сервером и клиентом. Сервер читает данные из `InputStream` и записывает данные в `OutputStream`.

5. **Закрытие потоков:** После завершения обмена данными, необходимо закрыть потоки `InputStream` и `OutputStream`, чтобы освободить ресурсы.

6. **Закрытие сокета:** В завершение, необходимо закрыть сокет `Socket`, представляющий соединение с конкретным клиентом, чтобы разорвать соединение и освободить системные ресурсы. `ServerSocket` при этом продолжает слушать новые подключения.

Примеры кода работы с сокетами приведены в **лабораторной работе 3**.

Передача данных по протоколу UDP

Для передачи данных по протоколу UDP используются классы **DatagramPacket** и **DatagramSocket** из пакета **java.net**.

– **DatagramPacket**: Этот класс представляет собой UDP-дейтаграмму, т.е. единицу данных, передаваемую по UDP. Он содержит данные для передачи, адрес назначения и порт.

- **Создание DatagramPacket:**

- можно создать дейтаграмму, указав только массив байтов (`byte[]`) и его длину. В этом случае дейтаграмма предназначена для получения данных;
- можно создать дейтаграмму, указав массив байтов, его длину, IP-адрес (`InetAddress`) и порт назначения. Это используется для отправки данных.

- **Получение информации из DatagramPacket:**

- `getAddress()`: Возвращает IP-адрес назначения;
- `getPort()`: Возвращает порт назначения;
- `getData()`: Возвращает массив байтов, содержащий данные;
- `getLength()`: Возвращает длину данных в массиве.

– **DatagramSocket**: Этот класс представляет собой сокет UDP, через который происходит отправка и получение дейтаграмм.

- **Создание DatagramSocket:**

- можно создать сокет, который будет использовать любой доступный локальный порт;
- можно создать сокет, указав конкретный локальный порт, который он должен использовать.

- **Методы DatagramSocket:**

- `send(DatagramPacket p)`: Отправляет дейтаграмму `p` через сокет;
- `receive(DatagramPacket p)`: Принимает входящую дейтаграмму и сохраняет ее данные в объект `p`. Этот метод блокируется до тех пор, пока не будет получена дейтаграмма;
- `setSoTimeout(int ms)`: Устанавливает максимальное время ожидания (в миллисекундах) для операции `receive()`. Если дейтаграмма не будет получена в течение этого времени, будет выброшено исключение `SocketTimeoutException`;
- `close()`: Закрывает сокет, освобождая ресурсы.

Важные моменты:

- UDP – протокол без установления соединения. Это означает, что перед отправкой данных не устанавливается соединение между отправителем и получателем.
- DatagramPacket – это контейнер для данных, адреса и порта.
- DatagramSocket – это точка входа и выхода для UDP-трафика.
- `setSoTimeout()` важен для предотвращения зависания программы при ожидании данных.

Последовательность создания UDP-сокета на стороне сервера:

1. **Инициализация:** Создается объект `DatagramSocket`, который будет прослушивать входящие соединения на определенном порту (например, 7777). Также выделяются буферы (`byte[] ib`, `byte[] ob`) для хранения входящих и исходящих данных.
2. **Прием данных:** Сервер ожидает и принимает датаграмму (пакет данных) от клиента, используя метод `receive()`. Полученные данные помещаются в буфер `ib`.
3. **Обработка и подготовка ответа:** Формируется ответ клиенту. Этот шаг подразумевает обработку полученных данных и подготовку массива байтов `ob`, содержащего ответ.
4. **Отправка ответа:** Создается новая датаграмма `DatagramPacket` с данными из буфера `ob`, адресом и портом клиента (полученными из принятой датаграммы). Затем датаграмма отправляется клиенту с помощью метода `send()`.
5. **Завершение:** Сокет закрывается с помощью метода `close()`, освобождая ресурсы.

Последовательность создания UDP-сокета на стороне клиента:

1. **Инициализация:** Создается объект `DatagramSocket` без указания порта (система автоматически выделит свободный порт). Также выделяются буферы (`byte[] ib`, `byte[] ob`) для хранения входящих и исходящих данных.
2. **Подготовка запроса:** Формируется запрос серверу. Этот шаг подразумевает подготовку массива байтов `ob`, содержащего данные запроса.
3. **Отправка запроса:** Создается датаграмма `DatagramPacket` с данными из буфера `ob`, адресом сервера (полученным по имени `server_name`) и портом сервера (7777). Затем датаграмма отправляется серверу с помощью метода `send()`.
4. **Прием ответа:** Клиент ожидает и принимает ответную датаграмму от сервера, используя метод `receive()`. Полученные данные помещаются в буфер `ib`.
5. **Завершение:** Сокет закрывается с помощью метода `close()`, освобождая ресурсы.

Пример кода с использованием UDP:

Отправка датаграмм (Sender)

```
import java.net.*;
import java.io.*;

public class UDPSender {
    public static void main(String[] args) {
        try (DatagramSocket socket = new DatagramSocket()) {
            InetAddress address = InetAddress.getByName("localhost");
            byte[] buffer = "Привет, UDP!".getBytes();
            DatagramPacket packet = new DatagramPacket(buffer, buffer.length, address, 4445);
            socket.send(packet);
            System.out.println("Сообщение отправлено.");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Прием датаграмм (Receiver):

```
import java.net.*;
import java.io.*;

public class UDPReceiver {
    public static void main(String[] args) {
        try (DatagramSocket socket = new DatagramSocket(4445)) {
            byte[] buffer = new byte[256];
            DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
            socket.receive(packet);
            String received = new String(packet.getData(), 0, packet.getLength());
            System.out.println("Получено: " + received);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

- **DatagramSocket:** Представляет собой конечную точку для отправки и получения датаграмм.
- **InetAddress:** IP-адрес удаленного хоста.
- **DatagramPacket:** Пакет данных, содержащий данные, адрес отправителя/получателя и порт.
- **send():** Метод для отправки датаграммы.
- **receive():** Метод для приема датаграммы.

Запуск:

1. Сначала запустите UDPReceiver, чтобы он начал прослушивать порт 4445.
2. Затем запустите UDPSender, который отправит сообщение на порт 4445.
3. UDPReceiver получит сообщение и выведет его на консоль.

Работа с сетевыми ресурсами в Java: URI, URL и URLConnection

Для работы с сетевыми ресурсами используются классы из пакета java.net.

Ключевые понятия:

- **URI (Uniform Resource Identifier)**: Универсальный идентификатор ресурса. Это строка, которая однозначно идентифицирует ресурс, например, веб-страницу или файл.

- **URL (Uniform Resource Locator)**: Универсальный локатор ресурса. Это **специальный вид URI**, который не только идентифицирует ресурс, но и указывает, **где его найти** (например, по какому протоколу и на каком сервере). URL содержит информацию о протоколе, хосте, порте, пути к ресурсу и, опционально, параметры запроса и якорь (фрагмент). Пример:

`http://user:password@example.com:8080/path/to/resource?query=value#fragment.`

Основные классы:

- `java.net.URI`: Представляет URI;
- `java.net.URL`: Представляет URL;
- `java.net.URLConnection`: Абстрактный класс для установления соединения с ресурсом, на который указывает URL. Предоставляет методы для настройки соединения и получения информации о ресурсе;
- `java.net.HttpURLConnection`: Подкласс `URLConnection`, предназначенный для работы с HTTP-соединениями.

Рекомендуемый процесс работы с URL-соединениями:

1. **Создание URI**: Сначала создается объект `URI`, представляющий ресурс. Это можно сделать, указав строку URL или используя конструктор с отдельными компонентами.

2. **Преобразование URI в URL**: Объект `URI` преобразуется в объект `URL` с помощью метода `toURL()`.

3. **Установление соединения и получение потока данных**: Есть два основных способа:

- **Использование URLConnection**: Более гибкий подход. Сначала вызывается `url.openConnection()` для получения объекта `URLConnection`. Затем можно настроить соединение (например, установить заголовки

запроса, разрешить чтение/запись) и вызвать `connect()` для установления соединения. После этого можно получить входной поток данных с помощью `getInputStream()`.

- **Прямое открытие потока:** более простой способ. Вызывается `url.openStream()` для получения входного потока данных напрямую.

4. **Чтение данных:** Данные считываются из входного потока (`InputStream`).

5. **Закрытие соединения и потока:** После завершения работы с ресурсом необходимо закрыть входной поток (`is.close()`) и, если использовался `URLConnection`, закрыть соединение (`uc.disconnect()`).

Преимущества использования `URLConnection`:

Использование `URLConnection` (и особенно `HttpURLConnection`) предоставляет больше возможностей, чем простое открытие потока из URL. Это позволяет:

- **Настраивать параметры соединения:** например, устанавливать заголовки запроса, указывать метод HTTP (GET, POST и т.д.), разрешать чтение и запись данных.
- **Получать информацию о ресурсе:** например, дату создания и модификации, тип содержимого (`Content-Type`), длину содержимого (`Content-Length`), кодировку содержимого.
- **Обрабатывать ошибки:** `HttpURLConnection` позволяет получать код ответа HTTP (например, 200 OK, 404 Not Found) и обрабатывать ошибки.

Тема 3.11

ВВЕДЕНИЕ В LAMBDA -ВЫРАЖЕНИЯ

Lambda-выражения — это анонимные функции (может, и не 100% верное определение для Java, но зато привносит некоторую ясность). Проще говоря, это метод без объявления, т.е. без модификаторов доступа, возвращающие значение и имя. Они позволяют написать метод и сразу же использовать его. Особенно полезно в случае однократного вызова метода, т.к. сокращает время на объявление и написание метода без необходимости создавать класс.

Lambda-выражения в Java обычно имеют следующий синтаксис (**аргументы**) -> (**тело**). Например:

```
(arg1, arg2...) -> { тело }
```

```
(тип1 arg1, тип2 arg2...) -> { тело }
```

Далее идет несколько примеров настоящих Lambda-выражений:

```
(int a, int b) -> { return a + b; }  
() -> System.out.println("Hello World");  
(String s) -> { System.out.println(s); }  
() -> 42  
() -> { return 3.1415 };
```

Структура Lambda-выражений

Изучим структуру lambda-выражений:

- Lambda-выражения могут иметь от 0 и более входных параметров.
- Тип параметров можно указывать явно либо может быть получен из контекста. Например, (int a) можно записать и так (a).
- Параметры заключаются в круглые скобки и разделяются запятыми. Например (a, b) или (int a, int b) или (String a, int b, float c).
- Если параметров нет, то нужно использовать пустые круглые скобки. Например () -> 42.
- Когда параметр один, если тип не указывается явно, скобки можно опустить. Пример: a -> return a*a.
- Тело Lambda-выражения может содержать от 0 и более выражений.
- Если тело состоит из одного оператора, его можно не заключать в фигурные скобки, а возвращаемое значение можно указывать без ключевого слова return.
- В противном случае фигурные скобки обязательны (блок кода), а в конце надо указывать возвращаемое значение с использованием ключевого слова return (в противном случае типом возвращаемого значения будет void).

Функциональный интерфейс

В Java, маркерные интерфейсы (Marker interface) – это интерфейсы без объявления методов и полей. Другими словами, маркерные интерфейсы – это пустые интерфейсы. Точно также, функциональные интерфейсы (Functional Interface) – это интерфейсы только с одним абстрактным методом, объявленным в нем. java.lang.Runnable – это пример функционального интерфейса. В нем объявлен только один метод void run().

Также есть интерфейс ActionListener – тоже функциональный. Раньше приходилось использовать анонимные классы для создания объектов, реализующих функциональный интерфейс. С Lambda-выражениями все стало проще. Каждое

lambda-выражение может быть неявно привязано к какому-нибудь функциональному интерфейсу. Например, можно создать ссылку на Runnable интерфейс, как показано в следующем примере:

```
Runnable r = () -> System.out.println("hello world");
```

Подобное преобразование всегда осуществляется неявно, когда мы не указываем функциональный интерфейс:

```
new Thread(  
    () -> System.out.println("hello world")  
).start();
```

В примере выше компилятор автоматически создает lambda-выражение как реализацию Runnable интерфейса из конструктора класса Thread:

```
public Thread(Runnable r) { }.
```

Несколько примеров lambda-выражений и соответствующих функциональных интерфейсов:

```
Consumer<Integer> c = (int x) -> { System.out.println(x) };  
  
BiConsumer<Integer, String> b = (Integer x, String y) -> System.out.println(x + " : " + y);  
  
Predicate<String> p = (String s) -> { s == null };
```

Аннотация `@FunctionalInterface`, добавленная в Java 8 согласно Java Language Specification, проверяет, является ли объявляемый интерфейс функциональным. Кроме того, в Java 8 включен ряд готовых функциональных интерфейсов для использования с Lambda-выражениями. `@FunctionalInterface` выдаст ошибку компиляции, если объявляемый интерфейс не будет функциональным.

Далее приводится пример определения функционального интерфейса:

```
@FunctionalInterface  
public interface WorkerInterface {  
  
    public void doSomeWork();  
  
}
```

Как следует из определения, функциональный интерфейс может иметь только один абстрактный метод. Если попытаться добавить еще один абстрактный метод, то вылезет ошибка компиляции. Пример:

```
@FunctionalInterface
public interface WorkerInterface {

    public void doSomeWork();
    public void doSomeMoreWork();
}
```

Error

Unexpected @FunctionalInterface annotation

@FunctionalInterface ^ **WorkerInterface** is not a functional **interface** multiple non-overriding **abstract** methods found in **interface WorkerInterface** 1 error

После определения функционального интерфейса, мы можем его использовать и получать все преимущества **Lambda**-выражений.

Пример:// определении функционального интерфейса

```
@FunctionalInterface
public interface WorkerInterface {
    public void doSomeWork();
}
```

Листинг 36

```
public class WorkerInterfaceTest {

    public static void execute(WorkerInterface worker) {
        worker.doSomeWork();
    }

    public static void main(String [] args) {

        // вызов метода doSomeWork через анонимный класс
        // (классический способ)
        execute(new WorkerInterface() {
            @Override
            public void doSomeWork() {
                System.out.println("Worker вызван через анонимный класс");
            }
        });

        // вызов метода doSomeWork через Lambda-выражения
        // (нововведение Java 8)
        execute( () -> System.out.println("Worker вызван через Lambda") );
    }
}
```

Результат:

Worker вызван через анонимный класс
Worker вызван через **Lambda**

Здесь (см. листинг 36) мы определили свой собственный функциональный интерфейс и воспользовались lambda-выражением. Метод `execute()` способен принимать lambda-выражения в качестве аргумента.

Примеры Lambda-выражений

Лучший способ вникнуть в Lambda-выражения – это рассмотреть несколько примеров: Поток `Thread` можно проинициализировать двумя способами:

```
// Старый способ:  
new Thread(new Runnable() {  
    @Override  
    public void run() {  
        System.out.println("Hello from thread");  
    }  
}).start();
```

```
// Новый способ:  
new Thread(  
    () -> System.out.println("Hello from thread")  
).start();
```

Управление событиями в Java 8 также можно осуществлять через Lambda-выражения. Далее представлены два способа добавления обработчика события `ActionListener` в компонент пользовательского интерфейса:

```
// Старый способ:  
button.addActionListener(new ActionListener() {  
    @Override  
    public void actionPerformed(ActionEvent e) {  
        System.out.println("Кнопка нажата. Старый способ!");  
    }  
}));
```

```
// Новый способ:  
button.addActionListener( (e) -> {  
    System.out.println("Кнопка нажата. Lambda!");  
});
```

Простой пример вывода всех элементов заданного массива. Заметьте, что есть более одного способа использования lambda-выражения. Ниже мы создаем lambda-выражение обычным способом, используя синтаксис стрелки, а также мы используем оператор двойного двоеточия (::), который в Java 8 конвертирует обычный метод в lambda-выражение:

```
// Старый способ:  
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7);  
for(Integer n: list) {  
    System.out.println(n);  
}
```

```
// Новый способ:  
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7);  
list.forEach(n -> System.out.println(n));
```

```
// Новый способ с использованием оператора двойного двоеточия ::  
list.forEach(System.out::println);
```

В следующем примере (листинг 37) мы используем функциональный интерфейс Predicate для создания теста и печати элементов, прошедших этот тест. Таким способом вы можете помещать логику в lambda-выражения и делать что-либо на ее основе.

Листинг 37

```
import java.util.Arrays;  
import java.util.List;  
import java.util.function.Predicate;  
  
public class Main {  
  
    public static void main(String [] a) {  
  
        List<Integer> list = Arrays.asList(1, 2, 3, 4, 5, 6, 7);  
  
        System.out.print("Выводит все числа: ");  
        evaluate(list, (n)->true);  
  
        System.out.print("Не выводит ни одного числа: ");  
        evaluate(list, (n)->false);  
  
        System.out.print("Вывод четных чисел: ");
```

```

    evaluate(list, (n)-> n%2 == 0 );

    System.out.print("Вывод нечетных чисел: ");
    evaluate(list, (n)-> n%2 == 1 );

    System.out.print("Вывод чисел больше 5: ");
    evaluate(list, (n)-> n > 5 );

}

public static void evaluate(List<Integer> list, Predicate<Integer> predicate) {
    for(Integer n: list) {
        if(predicate.test(n)) {
            System.out.print(n + " ");
        }
    }
    System.out.println();
}
}

```

Результат:

```

Выводит все числа: 1 2 3 4 5 6 7
Не выводит ни одного числа:
Вывод четных чисел: 2 4 6
Вывод нечетных чисел: 1 3 5 7
Вывод чисел больше 5: 6 7

```

Можно, например, вывести квадрат каждого элемента списка. Здесь мы используем метод `stream()`, чтобы преобразовать обычный список в поток. Java 8 предоставляет шикарный класс `Stream` (`java.util.stream.Stream`). Он содержит много полезных методов, с которыми можно использовать `lambda`-выражения. Мы передаем `lambda`-выражение `x -> x*x` в метод `map()`, который применяет его ко всем элементам в потоке. После чего мы используем `forEach` для печати всех элементов списка.

```

// Старый способ:
List<Integer> list = Arrays.asList(1,2,3,4,5,6,7);
for(Integer n : list) {
    int x = n * n;
    System.out.println(x);
}

```



```
// Новый способ:  
List<Integer> list = Arrays.asList(1,2,3,4,5,6,7);  
list.stream().map((x) -> x*x).forEach(System.out::println);
```

Дан список, нужно вывести сумму квадратов всех элемента списка. Lambda-выражения позволяет достигнуть этого написанием всего одной строки кода. В этом примере применен метод свертки (редукции) `reduce()`. Мы используем метод `map()` для возведения в квадрат каждого элемента, а потом применяем метод `reduce()` для свертки всех элементов в одно число.

```
// Старый способ:  
List<Integer> list = Arrays.asList(1,2,3,4,5,6,7);  
int sum = 0;  
for(Integer n : list) {  
    int x = n * n;  
    sum = sum + x;  
}  
System.out.println(sum);
```

```
// Новый способ:  
List<Integer> list = Arrays.asList(1,2,3,4,5,6,7);  
int sum = list.stream().map(x -> x*x).reduce((x,y) -> x + y).get();  
System.out.println(sum);
```

Отличие Lambda-выражений от анонимных классов

Главное отличие состоит в использовании ключевого слова **this**. Для анонимных классов ключевое слово **'this'** обозначает объект анонимного класса, в то время как в lambda-выражении **'this'** обозначает объект класса, в котором lambda-выражение используется. Другое их отличие заключается в способе компиляции. Java компилирует lambda-выражения с преобразованием их в private-методы класса. При этом используется инструкция **invokedynamic**, появившаяся в Java 7 для динамической привязки метода.

Заключение

Марк Рейнхолд (Mark Reinhold - Oracle's Chief Architect) назвал Lambda-выражения самым значительным изменением в модели программирования, которое когда-либо происходило – даже более значительным, чем дженерики (generics). Должно быть он прав, т.к. они дают Java-программистам возможности функциональных языков программирования, которых так давно все ждали. Наряду с такими новшествами, как методы виртуального расширения (Virtual extension methods), Lambda-выражения позволяют писать очень качественный код.

ЛАБОРАТОРНЫЙ ПРАКТИКУМ

Лабораторная работа 1 КЛАССЫ И ОБЪЕКТЫ В JAVA

Цель работы: изучить структуру класса в языке программирования Java, познакомиться с различными типами объектов и классов.

Теоретическая часть

Классы и объекты

В Java, как в объектно-ориентированном языке, ключевыми элементами являются **классы** и **объекты**. Класс служит **определением** или **шаблоном** для создания объектов, задавая их атрибуты (данные) и методы (поведение). **Объект** – это **реализация** или **экземпляр** класса, представляющий собой конкретную сущность с определенным состоянием и способностью выполнять действия, описанные в классе. Java-программа организуется как набор взаимодействующих объектов. Аналогично, понятие «человек» с его общими характеристиками (две руки, две ноги, и т.д.) можно рассматривать как класс, а каждого отдельного человека – как объект, созданный на основе этого класса.

Структура класса

Класс определяется с помощью ключевого слова **class**:

```
1 public class Book
2 {
3
4 }
```

Рассмотрим составляющие нашего класса:

- 1) **public** – это модификатор доступа, определяющий, кто может использовать этот класс. В данном случае `public` означает, что класс доступен для всех. Java предлагает различные модификаторы доступа, позволяющие более точно контролировать, какие классы могут видеть и использовать другие классы;
 - 2) **class** – ключевое слово, говорящее о том, что это класс;
 - 3) **Book** – имя класса. Имена классов принято писать с заглавной буквы;
 - 4) **{ }** – фигурные скобки, между которыми разместится тело нашего класса.
- Внутри класса обычно определяются два основных типа членов: **поля**, представляющие собой переменные, которые характеризуют состояние

объекта (или класса), и **методы**, являющиеся именованными блоками кода, реализующими поведение. Класс также может содержать **вложенные классы** и **интерфейсы** для организации более сложной структуры. Модификаторы, ключевые слова, предшествующие объявлению класса, определяют его видимость и другие характеристики.

Например, класс **Book** мог бы иметь следующее определение:

```
1 class Book
2 {
3     public String name;
4     public String author;
5     public int year;
6
7     public void info()
8     {
9         System.out.printf("Book '%s' (author %s) was published in %d \n",
10             name, author, year);
11     }
12 }
13
```

Таким образом, в классе **Book** определены три переменных и один метод, который выводит значения этих переменных.

Функции и методы класса

Методы представляют собой подпрограммы, инкапсулированные в определениях классов. Они являются частью класса и определяются в его теле, наряду с полями (переменными экземпляра). Объявление метода включает в себя указание типа возвращаемого значения и спецификацию списка параметров, которые метод принимает.

Общая форма объявления метода такова:

```
1 [модификаторы] тип_возвращаемого_значения название_метода ([параметры]){
2     // тело метода
3 }
```

Модификаторы и параметры необязательны.

По умолчанию главный класс любой программы на Java содержит метод **main**, который служит точкой входа в программу:

```
1 public static void main(String[] args) {
2     System.out.println("привет мир!");
3 }
```

Ключевые слова `public` и `static` являются модификаторами. Далее идет тип возвращаемого значения. Ключевое слово `void` указывает на то, что метод ничего не возвращает.

Затем идут название метода – `main` и в скобках параметры метода – `String[] args`. В фигурные скобки заключено тело метода – все действия, которые он выполняет.

Условно методы, которые не возвращают никакого значения, называются **процедурами**.

Кроме `void` методы в Java могут возвращать конкретное значение. Такие методы также условно называют **функциями**.

Функции, возвращающие значения

В отличие от функций типа `void`, которые ничего не возвращают, функции, возвращающие значения, должны указывать конкретный тип данных, который они будут возвращать (например, `int`, `string`, `bool` и т.д.). Ключевым отличием является обязательное использование оператора `return`, за которым следует значение, которое функция возвращает. Тип этого возвращаемого значения должен строго соответствовать типу, указанному в определении функции. Например, если функция объявлена как возвращающая `int`, то оператор `return` должен возвращать целое число, которое будет интерпретировано как объект типа `int`.

Конструкторы в классах

Помимо обычных методов, классы могут содержать специальные методы, называемые конструкторами. Конструкторы отвечают за создание новых экземпляров (объектов) класса и обычно используются для инициализации этих объектов, т.е. присвоения им начальных значений. Важно отметить, что имя конструктора всегда должно совпадать с именем класса, например:

```
1  class Book
2  {
3      public String name;
4      public String author;
5      public int year;
6
7      public void info()
8      {
9          System.out.printf("Book '%s' (author %s) was published in %d \n",
10                           name, author, year);
11      }
12
13      public Book()
14      {
15      }
16
17      public Book(String name, String author, int year)
18      {
19          this.name = name;
20          this.author = author;
21          this.year = year;
22      }
23 }
```

Здесь у класса `Book` определено два конструктора. Первый конструктор без параметров присваивает неопределенные начальные значения полям. Второй конструктор присваивает полям класса значения, которые передаются через его параметры.

Т.к. имена параметров и имена полей класса в данном случае у нас совпадают – `name`, `author`, `year`, то мы используем ключевое слово **this**. Это ключевое слово представляет ссылку на текущий объект, поэтому в выражении `this.name = name`; первая часть `this.name` означает, что `name` – это поле текущего класса, а не название параметра `name`. Если бы у нас параметры и поля назывались по-разному, то использовать слово `this` было бы *необязательно*.

Мы можем определить несколько конструкторов для установки разного количества параметров и затем вызывать один конструктор из другого. Например:

```
1  class Book
2  {
3      public String name;
4      public String author;
5      public int year;
6
7      public void info()
8      {
9          System.out.printf("Book '%s' (author %s) was published in %d \n",
10             name, author, year);
11      }
12
13      public Book(String name, String author)
14      {
15          this.name = name;
16          this.author = author;
17      }
18
19      public Book(String name, String author, int year)
20      {
21          this(name, author);
22          this.year = year;
23      }
24  }
```

Допустим, нам нужно инициализировать объект, задав только два или три параметра. Вместо того, чтобы создавать конструктор, принимающий все три параметра и заставляющий нас указывать значения для всех полей, даже если они не нужны, мы можем использовать другой подход: создать конструктор, принимающий два параметра, и внутри него вызвать другой конструктор (с тремя параметрами), передав ему эти два параметра и, возможно, какое-то значение по умолчанию для третьего. Для вызова другого конструктора из текущего используется ключевое слово **this** с указанием списка параметров в скобках.

Создание объекта

Для работы с классом необходимо создать его экземпляр, называемый объектом. Сначала объявляется переменная, которая будет хранить ссылку на этот

объект. Затем, используя ключевое слово **new** и конструктор класса, создается сам объект, и ссылка на него присваивается объявленной переменной. Например:

```
1 public static void main(String[] args)
2 {
3     Book b;
4     b = new Book();
5 }
```

После объявления переменной `Book b`; эта переменная еще не ссылается ни на какой объект и имеет значение `null`. Затем мы создаем непосредственно объект класса `Book` с помощью одного из конструкторов и ключевого слова **new**.

Пакеты

В Java классы обычно группируются в пакеты. Пакеты служат для логической организации классов по функциональности. Java предоставляет множество встроенных пакетов, таких как `java.lang`, `java.util` и `java.io`. Пакеты могут быть вложенными друг в друга.

Использование пакетов предотвращает конфликты имен классов, которые могут возникнуть, если разные разработчики используют одинаковые имена. Пакет гарантирует уникальность имени класса.

Чтобы поместить класс в определенный пакет, используется ключевое слово **package**, за которым следует имя пакета. Например:

```
1 package by.psu.fit.cmsn.cross_pp.lab1.sample1;
```

Классы необязательно определять в пакеты. Если для класса пакет не определен, то считается, что данный класс находится в пакете по умолчанию, который не имеет имени.

Импорт пакетов и классов

Если нам надо использовать классы из других пакетов, то необходимо подключить эти пакеты и классы. Исключение составляют классы из пакета `java.lang` (например, `String`), которые подключаются в программу автоматически.

Например, класс `Scanner` находится в пакете `java.util`, поэтому мы можем получить к нему доступ следующим способом:

```
1 java.util.Scanner in = new java.util.Scanner(System.in);
```

Т.е. мы указываем полный путь к файлу в пакете при создании его объекта. Такое нагромождение имен пакетов не всегда удобно, и в качестве альтернативы мы можем импортировать пакеты и классы в проект с помощью директивы **import**, которая указывается после директивы **package**:

```
1 package by.psu.fit.cmsn.cross_pp.lab1.sample1;
2
3 import java.util.Scanner;
```

Директива **import** указывается в самом начале кода, после чего идет имя подключаемого класса (в данном случае класса `Scanner`).

В примере выше мы подключили только один класс, однако пакет `java.util` содержит еще множество классов, и чтобы не подключать по отдельности каждый класс, мы можем сразу подключить весь пакет:

```
1 import java.util.*; // импорт всех классов из пакета java.util
```

Теперь мы можем использовать любой класс из пакета `java.util`.

Иногда бывает, что два разных пакета содержат классы с одинаковым именем. Например, класс `Date` есть и в `java.util`, и в `java.sql`. Чтобы использовать оба этих класса одновременно, нужно указывать их полные имена, включая имя пакета:

```
1 java.util.Date utilDate = new java.util.Date();
2 java.sql.Date sqlDate = new java.sql.Date();
```

Статические члены и модификатор **static**

Кроме обычных методов и полей класс может иметь статические поля и методы. Например, главный класс программы имеет метод `main`, который является статическим:

```
1 public static void main(String[] args) {
2
3 }
```

Для объявления статических переменных и методов перед их объявлением указывается ключевое слово **static**. Статические члены класса могут использоваться без создания объектов класса.

Например, в классе `System` содержится статическая переменная `out`, с помощью которой выводятся данные на консоль. Так, создадим статическую переменную:

```

1 class Book{
2
3     private int id;
4     private static int counter=1;
5
6     public void displayId(){
7
8         System.out.printf("Id: %d \n", id);
9     }
10
11     private String author;
12     private int year;
13     private String name;
14
15     Book(String name, String author, int year){
16
17         this.name = name;
18         this.author = author;
19         this.year = year;
20         id=counter++;
21     }
22 }

```

Класс Book содержит статическую переменную counter, которая увеличивается в конструкторе, и ее значение присваивается переменной id.

После этого мы можем создать несколько объектов Book, и в каждом вызове конструктора переменная counter будет увеличиваться на единицу, т.к. она относится не к конкретному объекту, а ко всему классу Book в целом или всем объектам Book сразу:

```

1 Book b1 = new Book("Война и мир", "Л. Н. Толстой", 1863);
2 b1.displayId(); // выведет Id: 1
3
4 Book b2 = new Book("Отцы и дети", "И. Тургенев", 1862);
5 b2.displayId(); // выведет Id: 2

```

Хотя в примере выше мы сразу инициализировали статическую переменную, но иногда для инициализации статических полей применяется статический блок. Этот блок вызывается один раз в программе при создании первого объекта данного класса:

```

1 class Book{
2
3     private int id;
4     private static int counter;
5
6     static {
7
8         counter=1;
9         System.out.println("Вызов статического блока");
10    }
11    // остальной код класса
12 }

```


Константы на уровне класса часто объявляют как статические:

```
1 public final static double PI = 3.14;
```

Статические методы, как и переменные, также относятся ко всему классу. Например, создадим новый класс Algorithm и добавим в него две функции для вычисления числа Фибоначчи и факториала:

```
1 class Algorithm
2 {
3     public final static double PI = 3.14;
4
5     public static int factorial(int x)
6     {
7         if (x == 1)
8             return 1;
9         else
10            return x * factorial(x - 1); //рекурсия
11    }
12
13    public static int fibonacci(int x)
14    {
15        if (x == 0)
16            return 1;
17
18        if (x == 1)
19            return 1;
20        else
21            return fibonacci(x - 1) + fibonacci(x - 2); //рекурсия
22    }
23 }
```

Теперь используем их в программе:

```
1 public static void main(String[] args)
2 {
3     int num1 = Algorithm.factorial(5);
4     int num2 = Algorithm.fibonacci(5);
5
6     System.out.println(Algorithm.PI);
7 }
```

И поскольку методы factorial и fibonacci, а также поле PI являются статическими, то мы можем к ним обратиться напрямую без создания объекта класса: Algorithm.factorial(5).

ЗАДАНИЕ К ЛАБОРАТОРНОЙ РАБОТЕ

1. Ознакомиться с теоретической частью.
2. Реализовать класс, описывающий сущность согласно индивидуальному варианту, и учесть следующие требования:
 - в классе должно быть не менее 3 полей;
 - в классе должно быть предусмотрено 2 конструктора с параметрами и конструктор по умолчанию;

- доступ к членам класса должен осуществляться через методы `get` и `set`, а сами члены класса скрыты с помощью модификаторов доступа;
- реализовать подсчет созданных объектов класса с помощью статической переменной;
- по умолчанию переменные класса должны инициализироваться с помощью констант, определенных в данном классе;
- создать минимум 2 функции с несколькими параметрами (помимо геттеров и сеттеров) для манипуляции внутренними свойствами объекта;
- реализовать метод для вывода информации об объекте класса;
- в главном классе продемонстрировать работу всех методов класса, используя статический метод (помимо `main`).

ИНДИВИДУАЛЬНЫЕ ВАРИАНТЫ ЗАДАНИЙ

Номер варианта	Сущность, которую необходимо представить в виде класса
1	СТОЛ
2	РЕБЕНОК
3	ЧАСЫ
4	ЗДАНИЕ
5	СМАРТФОН
6	ТЕСТ
7	АДРЕС
8	ТОВАР
9	ПРИНТЕР
10	РУЧКА
11	ПЕРСОНА
12	АВТОМОБИЛЬ
13	КНИГА
14	ОЧКИ
15	КОРАБЛЬ
16	МИКРОСКОП
17	КАРТИНА
18	ФЛОМАСТЕР
19	ТЕЛЕВИЗОР
20	МОСТ

Порядок выполнения работы

1. Изучить основные теоретические положения.
2. Получить задание у преподавателя, выполнить задание.
3. Оформить отчет.

Требования к отчету

1. Цель работы.
2. Результаты работы.
3. Выводы.

Лабораторная работа 2

НАСЛЕДОВАНИЕ

Цель работы: изучить механизм наследования и реализации интерфейсов в языке программирования Java.

Теоретическая часть

Наследование. Наследование в Java – это механизм, позволяющий одному классу (подклассу) приобретать атрибуты и методы другого класса (суперкласса). Подкласс расширяет функциональность суперкласса, сохраняя при этом его базовые характеристики. Суперкласс является базовым классом, от которого происходит наследование, а подкласс – это класс, который наследует свойства суперкласса и может добавлять собственные. В результате наследования подкласс содержит все поля и методы суперкласса, а также может определять новые поля и методы, специфичные для его функциональности. Например, если класс «Транспортное средство» определяет свойства «скорость» и «вместимость», а класс «Скутер» наследует от «Транспортного средства», то класс «Скутер» автоматически получает доступ к свойствам «скорость» и «вместимость», и может дополнительно определять свои свойства.

Например, класс Person описывает отдельного человека:

```
1 public class Person {
2
3     private String name;
4     private String surname;
5
6     public String getName() { return name; }
7     public String getSurname() { return surname; }
8
9     public Person(String name, String surname){
10
11         this.name=name;
12         this.surname=surname;
13     }
14
15     public void displayInfo(){
16
17         System.out.println("Имя: " + name + " Фамилия: " + surname);
18     }
19 }
```

Если мы захотим добавить класс Employee, описывающий сотрудника, то, поскольку сотрудник является человеком, мы можем использовать механизм наследования. Класс Employee будет производным от класса Person, что позволит нам повторно использовать уже написанный код и избежать дублирования. Person в этом случае выступает в роли базового класса:

```

1 class Employee extends Person{
2
3 }

```

Механизм наследования позволяет одному классу (наследнику) расширять функциональность другого класса (базового). Для указания наследования используется ключевое слово `extends`. Синтаксис следующий: `class Наследник extends БазовыйКласс`. В случае, если `Employee` наследует от `Person`, это записывается как `class Employee extends Person`. При этом класс `Employee` автоматически получает доступ ко всем полям и методам, определенным в классе `Person`.

В классе `Employee` могут быть определены свои методы, включая конструктор, и поля:

```

1 class Employee extends Person{
2
3     private String company;
4
5     public Employee(String name, String surname, String company) {
6
7         super(name, surname);
8         this.company=company;
9     }
10
11     public void displayInfo(){
12
13         super.displayInfo();
14         System.out.println("Компания: " + company);
15     }
16 }

```

Класс `Employee` расширяет функциональность класса `Person`, добавляя поле `company` для хранения информации о месте работы сотрудника. Значение этого поля задается при создании объекта `Employee` в конструкторе.

Поля `name` и `surname`, унаследованные от класса `Person`, являются приватными, поэтому прямой доступ к ним из класса `Employee` невозможен. Вместо этого, для инициализации этих полей используется конструктор базового класса `Person`. Это делается с помощью ключевого слова `super()`, которому передаются необходимые аргументы (имя и фамилия).

Ключевое слово `super` позволяет обращаться к любым неprivатным членам (методам и полям) базового класса.

В классе `Employee` также переопределен метод `displayInfo()`. Чтобы избежать дублирования кода, в переопределенном методе сначала вызывается метод `displayInfo()` базового класса `Person` с помощью `super.displayInfo()`. Затем добавляется вывод информации, специфичной для сотрудника, например, название компании.

Используя обращение к методам базового класса, можно переопределить метод `displayInfo()`:

```
1 public void displayInfo(){
2
3     System.out.println("Имя: " + super.getName() + " Фамилия: "
4         + super.getSurname() + " Компания: " + company);
5 }
```

Конструкторы и наследование

Для вызова параметризованного конструктора суперкласса из конструктора подкласса требуется явный вызов `super()`. Если в конструкторе подкласса не указан явный вызов конструктора суперкласса, то компилятор автоматически вставляет вызов конструктора суперкласса по умолчанию (без аргументов) в качестве первой инструкции. Таким образом, явный вызов `super()` без аргументов эквивалентен его отсутствию:

```
1 public class Person
2 {
3     public Person()
4     {
5     }
6 }
7
8 // Между тем, в Employee.java
9 public class Employee extends Person
10 {
11     public Employee()
12     {
13     }
14 }
```

или

```
1 public class Person
2 {
3     public Person()
4     {
5     }
6 }
7
8 // Между тем, в Employee.java
9 public class Employee extends Person
10 {
11     public Employee()
12     {
13         super();
14     }
15 }
```

Конструкторы без аргументов

Создавая альтернативный конструктор, мы должны явно создать конструктор по умолчанию, иначе он будет недоступен. Например, следующий код вызовет ошибку компиляции:

```

1 public class Person
2 {
3     private String name;
4
5     public Person(String name)
6     {
7         this.name = name;
8     }
9 }
10
11 // Между тем, в Employee.java
12 public class Employee extends Person
13 {
14     public Employee() {}
15 }

```

Этот пример не содержит конструктора по умолчанию. Это связано с тем, что класс уже имеет другой конструктор. Конструктор по умолчанию, который также называют конструктором без аргументов, не создается компилятором, если определены другие конструкторы. Поэтому термин «конструктор по умолчанию» может быть немного вводящим в заблуждение, т.к. он не всегда присутствует.

Вызов конструктора из другого конструктора

Конструктор может быть вызван другим конструктором изнутри класса с использованием ключевого слова **this** и списка аргументов. Как и **super()**, вызов **this()** должен быть первой строкой конструктора. Например:

```

1 public class Person
2 {
3     private String name;
4
5     public Person()
6     {
7         this("Some reasonable default?");
8     }
9
10    public Person(String name)
11    {
12        this.name = name;
13    }
14 }

```

Мы будем часто встречать это выражение, когда один конструктор вызывает другой, передавая ему некоторое значение при вызове этого конструктора.

Наследование и абстракция

Абстрагирование в ООП — это процесс обобщения функциональности и данных, присущих нескольким классам, путем их переноса в общий родительский класс (суперкласс). Когда мы «абстрагируем» переменную или метод, мы перемещаем его вверх по иерархии наследования. Главная цель — избежать дублирования кода

и обеспечить его повторное использование. Централизация общего кода в супер-классе значительно упрощает его поддержку и модификацию.

Абстрактные классы и методы

Иногда возникает необходимость в создании классов, которые служат лишь в качестве шаблонов или общих концепций, а не для непосредственного создания объектов. Такие классы называются абстрактными классами. Они определяют общую структуру и поведение для группы связанных классов, но сами по себе не могут быть использованы для создания конкретных объектов.

Кроме того, часто встречаются ситуации, когда определенные действия (методы) должны выполняться по-разному в каждом классе, который наследует от общего родительского класса. В этом случае используются абстрактные методы. Абстрактный метод определяет только, что нужно сделать, но не как. Реализация (конкретный код) этого метода возлагается на классы-наследники.

Основные правила работы с абстрактными классами и методами:

- Любой класс может быть объявлен абстрактным, если это необходимо.
- Невозможно создать объект (экземпляр) абстрактного класса напрямую.
- Абстрактный метод не содержит реализации (тела метода). Он только объявляет сигнатуру метода.
- Если класс содержит хотя бы один абстрактный метод, то сам класс обязательно должен быть объявлен абстрактным.

Пример: предположим, мы хотим запретить создание объектов класса Employee напрямую. Чтобы сделать это, мы объявляем класс Employee как абстрактный, используя ключевое слово `abstract`. Это означает, что мы можем создавать только подклассы Employee (например, Manager, Developer), но не сам Employee.

```
1 public abstract class Employee extends Person
2 {
3     //...
4 }
```

Абстрактный класс может содержать не только обычные методы с реализацией, но и абстрактные методы. Абстрактные методы обозначаются ключевым словом `abstract` и представляют собой лишь объявление, без какой-либо конкретной логики. Класс, унаследовавший от абстрактного класса, должен обязательно реализовать все абстрактные методы, определенные в базовом абстрактном классе. Важно помнить, что наличие хотя бы одного абстрактного метода в классе автоматически делает этот класс абстрактным.

Представим разработку программного обеспечения для банковских операций. Мы определяем классы Person (человек), Employee (сотрудник банка) и Client (клиент банка). Классы Employee и Client логично наследовать от Person, т.к. они разделяют общие атрибуты и поведение. Поскольку мы не планируем создавать объекты непосредственно типа Person (только сотрудники и клиенты), целесообразно определить класс Person как абстрактный. Например:

```
1  abstract class Person
2  {
3      private String name;
4      private String surname;
5
6      public String getName()
7      {
8          return name;
9      }
10
11     public String getSurname()
12     {
13         return surname;
14     }
15
16     public Person(String name, String surname)
17     {
18         this.name = name;
19         this.surname = surname;
20     }
21
22     public abstract void displayInfo();
23 }
24
25 class Employee extends Person
26 {
27
28     private String bank;
29
30     public Employee(String name, String surname, String company)
31     {
32         super(name, surname);
33         this.bank = company;
34     }
35
36     public void displayInfo()
37     {
38         System.out.println("Имя: " + super.getName() + " Фамилия: "
39                             + super.getSurname() + " Работает в банке: " + bank);
40     }
41 }
42
43 class Client extends Person
44 {
45     private String bank;
46
47     public Client(String name, String surname, String company)
48     {
49         super(name, surname);
50         this.bank = company;
51     }
52
53     public void displayInfo()
54     {
55         System.out.println("Имя: " + super.getName() + " Фамилия: "
56                             + super.getSurname() + " Клиент банка: " + bank);
57     }
58 }
```

Другим примером являются системы фигур. В реальности не существует геометрической фигуры как таковой. Есть круг, прямоугольник, квадрат, но просто фигуры нет. Однако же и круг, и прямоугольник имеют что-то общее и являются фигурами:

```

1  //абстрактный класс фигуры
2  public abstract class Figure
3  {
4      float x; // x-координата точки
5      float y; // y-координата точки
6
7      public Figure(float x, float y)
8      {
9          this.x=x;
10         this.y=y;
11     }
12
13     // абстрактный метод для получения периметра
14     public abstract float getPerimeter();
15
16     // абстрактный метод для получения площади
17     public abstract float getArea();
18 }
19
20 // производный класс прямоугольника
21 class Rectangle extends Figure
22 {
23     private float width;
24     private float height;
25
26     // конструктор с обращением к конструктору класса Figure
27     public Rectangle(float x, float y, float width, float height)
28     {
29         super(x,y);
30         this.width = width;
31         this.height = height;
32     }
33
34     public float getPerimeter()
35     {
36         return width * 2 + height * 2;
37     }
38
39     public float getArea()
40     {
41         return width * height;
42     }
43 }

```

Когда использовать абстракции:

1. Отложите абстракции на потом. Начинать проект с абстрактных классов — плохая идея. Это может загнать вас в рамки и ограничить гибкость приложения. Лучше сначала реализовать конкретные решения, а уже потом, когда станет ясно, какое поведение действительно общее, вынести его в абстрактные классы. Абстрагирование «по требованию» обычно более эффективно.

2. Используйте абстракции с осторожностью. Даже если абстракции кажутся мощным инструментом, не спешите их применять. Если суперклассы не имеют значительного общего поведения, а сами по себе не очень важны, лучше обойтись без абстракций. Слишком сложные иерархии наследования могут усложнить поддержку кода. Важно найти баланс между большими, монолитными классами и удобством сопровождения.

Присвоения: классы

Присвоение ссылки из одного класса на переменную типа, принадлежащую к другому классу, допускается, но существуют определенные правила. Рассмотрим следующий пример:

```

class Person
{
    ...
}

abstract class Employee extends Person
{
    ...
}

class Manager extends Employee
{
    ...
}

Manager m = new Manager();
Employee e = new Employee();
Person p = m; // нормально
p = e; // тоже нормально
Employee e2 = e; // да, так можно
e = m; // тоже нормально
e2 = p; // Неверно!

```

Вызываемая переменная должна быть супертипом класса, принадлежащего к источнику ссылки, иначе компилятор выдаст сообщение об ошибке. В принципе, все, что справа от присвоения, должно быть подклассом или тем же классом того, что слева. В противном случае при присвоении объектов с разными графами наследования (например, Manager и Employee) можно присвоить переменную недопустимого типа.

Оператор instanceof

Оператор instanceof служит для определения принадлежности объекта к определенному типу данных. Он позволяет выяснить, является ли объект экземпляром конкретного класса или реализует указанный интерфейс.

При использовании instanceof слева указывается проверяемый объект, а справа – имя класса или интерфейса, с которым производится сравнение. Если объект принадлежит указанному классу (или его подклассу) или реализует указанный интерфейс, оператор вернет true; в противном случае – false.

instanceof также применяется для проверки, является ли объект массивом. В этом случае проверка включает два этапа: сначала определяется, является ли объект массивом вообще, а затем проверяется, является ли тип элементов массива совместимым с типом, указанным в правом операнде. Например, если правым операндом является класс Component, а левым – массив объектов Button, оператор вернет true, т.к. Button является подклассом Component. Тест для такого массива будет выглядеть так:

```
if (x instanceof Component[])
```

Невозможно провести тестирование для «любого массива с элементами любого типа», т.е. следующая строка недопустима:

```
if (x instanceof [])
```

Недопустимо также тестирование на массив с элементами типа Object:

```
if (x instanceof Object []),
```

поскольку элементы массива могут быть примитивного типа и тест попросту не сработает.

Если левый операнд равен null, то instanceof возвратит false. Никакого исключения не возникнет. Пример:

```
1  class Student
2  {
3      // some code
4  }
5
6  class Graduate extends Student
7  {
8      //some code
9  }
10
11 class UndergraduateStudent extends Student
12 {
13     // some code
14 }
15
16 public class st
17 {
18     public static void main(String []args)
19     {
20         Student x = new UndergraduateStudent();
21
22         if (x instanceof UndergraduateStudent)
23             System.out.println("Mr. X is an undergraduate student.");
24         else
25             System.out.println("Mr. X is an graduate student.");
26     }
27 }
```

Интерфейсы

Интерфейс – это контракт, который определяет, что должен уметь делать объект, не указывая, как именно он это делает.

Определить интерфейс можно так:

```
public interface interfaceName
{
    returnType methodName( argumentList );
}
```

Интерфейс объявляется подобно классу, но с использованием ключевого слова interface. Имя интерфейса выбирается по тем же правилам, что и имя класса и, как правило, следует тем же соглашениям об именовании.

В отличие от классов методы, объявленные в интерфейсе, не содержат реализации (тела метода). Реализация этих методов возлагается на класс, который «реализует» данный интерфейс. Это аналогично абстрактным методам в абстрактных классах.

Интерфейсы могут формировать иерархии, как и классы. Ключевое отличие состоит в том, что класс может реализовывать несколько интерфейсов одновременно, в то время как класс может наследовать только от одного класса. Если класс наследует от другого класса и реализует интерфейсы, то сначала указывается наследуемый класс, а затем перечисляются реализуемые интерфейсы, вот так:

```
1 public class Manager extends Employee implements BonusEligible, StockOptionRecipient
2 {
3     // и т.д.
4 }
```

Интерфейсы и доступность методов

Методы, объявленные в интерфейсе, не требуют явного указания модификатора доступа. Однако они всегда подразумеваются как `public`. Это связано с тем, что интерфейс определяет контракт, который реализующие классы должны выполнять. Чтобы обеспечить полную реализацию этого контракта, все методы интерфейса должны быть доступны извне.

Интерфейсы и файлы Java

При создании интерфейсов в файле `.java` следует помнить о следующих правилах:

- Только один интерфейс в файле может быть объявлен с модификатором доступа `public`.
- Имя этого `public` интерфейса должно совпадать с именем файла.
- Остальные интерфейсы в файле (если они есть) не должны иметь никаких модификаторов доступа (т.е. иметь `package-private` доступ).

Интерфейс без тела

Интерфейс может быть объявлен и при этом не содержать никаких методов или полей.

Например:

```
1 public interface BookPrintable
2 {
3 }
```

Такие интерфейсы называются **интерфейсами-маркерами** (marker interfaces), потому что они маркируют класс как реализацию этого интерфейса, но не предлагают особого явного поведения.

ЗАДАНИЕ К ЛАБОРАТОРНОЙ РАБОТЕ

1. Ознакомьтесь с теоретическим материалом, необходимым для выполнения работы, получите индивидуальное задание у преподавателя.

2. Проектирование иерархии классов и интерфейсов:

- Создайте иерархию классов, представляющую сущности вашего варианта.
- Включите в иерархию как минимум один интерфейс, определяющий общую логику для классов.

- В иерархии должен быть как минимум один абстрактный класс.

3. Реализация классов:

- **Конструкторы:** В каждом базовом классе реализуйте два конструктора с параметрами и один конструктор по умолчанию (без параметров).

- **Вызов конструкторов:** В классах-наследниках продемонстрируйте вызов конструктора базового класса и конструктора текущего класса (использование `super()` или аналогичного механизма).

- **Инкапсуляция:** Сделайте поля классов приватными (используйте модификаторы доступа, например, `private`). Доступ к полям и их изменение должны осуществляться только через методы `get` (геттеры) и `set` (сеттеры).

- **Инициализация по умолчанию:** При объявлении полей класса, присваивайте им значения по умолчанию, используя константы, определенные в базовом классе или интерфейсе.

4. Демонстрация работы иерархии:

- **Инициализация объектов:** В главном классе (или в отдельном тестовом классе) создайте экземпляры (объекты) различных классов из вашей иерархии.

- **Оператор `instanceof`:** Продемонстрируйте использование оператора `instanceof` для проверки принадлежности объекта к определенному классу.

Порядок выполнения работы

1. Изучить основные теоретические положения.
2. Получить задание у преподавателя, выполнить задание.
3. Оформить отчет.

Требования к отчету

1. Цель работы.
2. Результаты работы.
3. Выводы.

Лабораторная работа 3

СЕТЕВОЕ ПРОГРАММИРОВАНИЕ С СОКЕТАМИ И КАНАЛАМИ

Цель работы: приобретение навыков проектирования и разработки приложений в архитектуре клиент-сервер.

Теоретические сведения

Клиент-серверная архитектура – это способ организации вычислительных систем или сетей, при котором задачи и обработка данных распределяются между двумя типами участников: серверами (поставщиками услуг) и клиентами (потребителями этих услуг). Клиенты запрашивают ресурсы или сервисы, а серверы предоставляют их. Обычно взаимодействие между клиентами и серверами происходит по сети, и они могут быть реализованы как на разных компьютерах, так и в виде программ на одном устройстве.

Для установления связи, сервер сначала «слушает» входящие соединения через специальный объект. Клиент, в свою очередь, пытается подключиться к серверу, используя другой объект. После успешного установления соединения, обе стороны (клиент и сервер) получают возможность обмениваться данными, как если бы они работали с файлом, используя стандартные команды ввода/вывода.

Для однозначной идентификации сервера в сети недостаточно знать только его IP-адрес, т.к. на одном компьютере может работать несколько серверов. Поэтому используется понятие порта. При установлении соединения и клиент, и сервер должны указать номер порта, через который будет осуществляться обмен данными.

Порт – это своего рода «виртуальный канал» внутри компьютера. Клиент знает IP-адрес сервера, но для доступа к конкретной службе на этом сервере ему нужен номер порта. Каждая служба обычно «слушает» определенный порт. Клиент должен заранее знать, на каком порту работает нужная ему служба, чтобы установить с ней соединение. Номер порта служит вторым уровнем адресации, позволяя клиенту обратиться к конкретной службе на сервере.

Системные службы зарезервировали использование портов с номерами от 1 до 1024, так что вы не можете использовать этот или любой другой порт, про который вы знаете, что он задействован.

Сокет – это программный интерфейс, предназначенный для передачи данных между приложениями. Что же касается типов сокетов, то их два – потоковые и датаграммные.

Сокет создается, чтобы установить соединение с другой машиной, затем получается `InputStream` и `OutputStream` (или, с соответствующими конверторами `Reader` и `Writer`) из сокета, чтобы иметь возможность трактовать соединение как объект потока ввода/вывода. Существует два класса сокетов, основанных на потоках: `ServerSocket`, который использует сервер для «прослушивания» входящих соединений, и `Socket`, который использует клиент для инициализации соединения. Как только клиент создаст сокетное соединение, `ServerSocket` возвратит (посредством метода `accept()`) соответствующий `Socket`, через который может происходить коммуникация на стороне сервера. На этой стадии используются методы `getInputStream()` и `getOutputStream()` для получения соответствующих объектов `InputStream`'а и `OutputStream`'а для каждого сокета. Они должны быть обернуты внутрь буферных и форматирующих классов точно так же, как и другие объекты потоков.

Когда создается `ServerSocket`, то ему дается только номер порта, но не дается IP адрес, поскольку он уже есть на той машине, на которой он установлен. Однако когда создается `Socket`, то необходимо передать ему и IP-адрес, и номер порта, с которым необходимо соединиться. `Socket`, который возвращается из метода `ServerSocket.accept()`, уже содержит всю эту информацию.

Представленный ниже пример покажет использование серверного и клиентского сокета. Все, что делает сервер, это ожидает соединения, затем использует сокет, полученный при соединении для создания нового экземпляра `ClientHandler`, которому этот сокет и передается, после чего этот клиент запускается в отдельном потоке (`Thread`). Таким образом, все клиенты, подключенные к серверу, будут одновременно обслуживаться сервером – каждый в своем потоке.

Клиент создает соединение с сервером, затем создает `OutputStream` и создает некоторую обертку, как и в сервере. Строки текста посылаются через полученный `PrintWriter`. Клиент также создает `InputStream` с соответствующей конвертацией и оберткой, чтобы «слушать», что «говорит» сервер, который, в данном случае, просто отсылает слова назад.

Пример сервера, который создает новый экземпляр `ClientHandler`, – листинг 38.

Листинг 38

```
import java.io.*;
import java.net.*;
import java.util.Scanner;
import java.util.concurrent.CopyOnWriteArrayList;

public class MyServer {
    public static final int PORT = 3000;
        public static CopyOnWriteArrayList<ClientHandler> clients = new
            CopyOnWriteArrayList<>();
    public static ServerSocket serverSocket;
    public static Socket clientSocket;
    public static ClientHandler clientHandler;

    // ПОДКЛЮЧЕНИЕ НОВЫХ КЛИЕНТОВ
    public static void main(String[] args) {

        try {
            serverSocket = new ServerSocket(PORT);
            System.out.println("Chat started, waiting for clients...");
            while (true) {
                clientSocket = serverSocket.accept();
                System.out.println("New client: " + clientSocket);
                clientHandler = new ClientHandler(clientSocket);
                clients.add(clientHandler);
                new Thread(clientHandler).start();
            }
        }
        catch (IOException e) { System.out.println("Ups.. something is broken"); }
    }

    // РАССЫЛКА ЧАТА
    public static void broadcast(String message, ClientHandler sender) {
        for (ClientHandler client : clients) {
            if (client != sender) {
                client.sendMessage(message);
            }
        }
    }
}
```

Код соответствующего ClientHandler-а (листинг 39):

Листинг 39

```
class ClientHandler implements Runnable {
    Socket clientSocket; PrintWriter out; BufferedReader in; String username;

    // СОЗДАНИЕ ВВОДА И ВЫВОДА ДЛЯ КАЖДОГО КЛИЕНТА
    public ClientHandler(Socket socket) {
        this.clientSocket = socket;

        try {
            out = new PrintWriter(clientSocket.getOutputStream(), true);
            in = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
        } catch (IOException e) { e.printStackTrace(); }
    }

    // ОБРАБОТЧИК КЛИЕНТА
    @Override public void run() {
        try {
            username = getUsername(); MyServer.broadcast(username + " connected!", this);
            System.out.println(username + " connected!");
            System.out.println("Clients list");
            for (ClientHandler client : MyServer.clients)
            {
                System.out.println("\t- " + client.username);
            }
            out.println("Hi, " + username + "!\n");
            String inputLine;
            while ((inputLine = in.readLine()) != null) {
                System.out.println "["+username+" ":"+inputLine); MyServer.broadcast(
                "["+username+" ":"+inputLine, this);
            }
            MyServer.clients.remove(this); in.close(); out.close(); clientSocket.close();
        } catch (IOException e) {
            System.out.println(username + " left us...");
        } finally{
            MyServer.clients.remove(this); out.close();
        }
    }
}
```

Таким образом, для ServerSocket'a необходим только номер порта, а не IP-адрес (т.к. он запускается на локальной машине!). Когда вызывается accept(), метод блокирует выполнение до тех пор, пока клиент не попытается подсоединиться

к серверу. Т.е. сервер ожидает соединения, но другой процесс может выполняться. Когда соединение установлено, метод `accept()` возвращает объект `Socket`, представляющий это соединение.

В описанном механизме обработана ответственность за очистку сокета. Если конструктор `ServerSocket` завершится неудачей, программа просто завершится. Если конструктор `ServerSocket` завершится успешно, то все вызовы методов должны быть помещены в блок «try-finally», чтобы убедиться, что блок не будет покинут ни при каких условиях и `ServerSocket` будет правильно закрыт.

Аналогичная логика используется для сокета, возвращаемого из метода `accept()`. Если метод `accept()` завершится неудачей, то необходимо предположить, что сокет не существует и не удерживает никаких ресурсов, так что он не нуждается в очистке. Однако если он закончится успешно, то следующие выражения должны быть помещены в блок «try-finally», чтобы при каких-либо ошибках все равно произошла очистка. Позаботится об этом необходимо, потому что сокеты используют важные ресурсы, не относящиеся к памяти.

Код класса `ClientHandler` принимает в конструкторе клиентский сокет и сохраняет его. Далее создается `PrintWriter`, который ассоциируется с выходным потоком (`OutputStream`) этого клиентского сокета – для отсылки сообщений клиенту. Также создается `BufferedReader`, ассоциируемый с входным потоком (`InputStream`) того-же сокета, для чтения сообщений клиента. Далее ключевым моментом является вызов `in.readLine()`, это ожидание данных во входном потоке. Отсылка данных в этот поток выполняется с помощью `out.println()`. Остальной код достаточно понятен.

Пример клиента (листинг 40):

Листинг 40

```
class MyClient {
    static final String SERVER_ADDRESS = "localhost";
    static final int SERVER_PORT = 3000;
    static Socket socket;
    static PrintWriter out;
    static BufferedReader in;

    // ПОДКЛЮЧЕНИЕ
    public static void main(String[] args) {
        try {
            socket = new Socket(SERVER_ADDRESS, SERVER_PORT);
            System.out.println("Connection succeed!");
            out = new PrintWriter(socket.getOutputStream(), true);
            in = new BufferedReader(new
                InputStreamReader(socket.getInputStream()));
            new Thread(() -> {
```

```

    try {
        String serverResponse;
        while ((serverResponse = in.readLine()) != null) {
            System.out.println(serverResponse);
        }
    } catch (IOException e) {
        System.out.println("Everything is broken(((");
    }
}).start();
Scanner scanner = new Scanner(System.in);
String userInput;
while (true) {
    userInput = scanner.nextLine();
    out.println(userInput);
}
} catch (IOException e) { System.out.println("Connection lost!!!"); }
}
}

```

Здесь Socket создается при указании IP-адреса и номера порта. Интернет-соединение уникально определяется четырьмя параметрами: клиентским хостом, клиентским номером порта, серверным хостом и серверным номером порта. Когда запускается сервер, он получает назначаемый порт (3000) на localhost (127.0.0.1). Теперь, чтобы передать данные между клиентом и сервером, каждая сторона знает, куда посылать их. Поэтому, в процессе соединения с «известным» сервером клиент посылает «обратный адрес», чтобы сервер знал, куда посылать данные.

Как только объект Socket будет создан, создается PrintWriter для выходного потока данных сокета, а также BufferedReader для входного потока, после чего в отдельном потоке запускается цикл ожидания/чтения сообщений от сервера и вывода их в консоль. После чего в основном потоке (параллельно с ожиданием данных от сервера) запрашиваем имя пользователя с консоли ввода и отправляем это имя на сервер для регистрации нового пользователя.

ЗАДАНИЯ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ.

Создать на основе сокетов клиент-серверное приложение:

1. Клиент посылает через сервер сообщение другому клиенту.
2. Клиент посылает через сервер сообщение другому клиенту, выбранному из списка.
3. Чат. Клиент посылает через сервер сообщение, которое получают все клиенты. Список клиентов хранится на сервере в файле.

4. Сервер рассылает сообщения выбранным из списка клиентам. Список хранится в файле.

5. Сервер рассылает сообщения в определенное время определенным клиентам.

6. Сервер рассылает сообщения только тем клиентам, которые в настоящий момент находятся в on-line.

7. Чат. Сервер рассылает всем клиентам информацию о клиентах, вошедших в чат и покинувших его.

8. Клиент выбирает изображение из списка и пересылает его другому клиенту через сервер.

9. Игра по сети в «Морской бой».

10. Написать программу, сканирующую сеть в указанном диапазоне IP-адресов на наличие активных компьютеров.

11. Прокси. Программа должна принимать сообщения от любого клиента, работающего на протоколе TCP, и отсылать их на соответствующий сервер. При передаче изменять сообщение.

Вариант назначается преподавателем.

ВОПРОСЫ ДЛЯ КОНТРОЛЯ ЗНАНИЙ

1. Что такое архитектура клиент-сервер?
2. Назначение порта, зарезервированные номера портов.
3. Что такое сокет, назначение сокета?
4. Объясните механизм создания входного и выходного потоков.
5. Объясните механизм работы метода accept().

Порядок выполнения работы

1. Изучить основные теоретические положения.
2. Получить задание у преподавателя, выполнить задание.
3. Оформить отчет.

Требования к отчету

1. Цель работы.
2. Результаты работы.
3. Выводы.

Лабораторная работа 4

МНОГОПОТОЧНОЕ ПРОГРАММИРОВАНИЕ, БАЗОВЫЕ ПОНЯТИЯ

Цель работы: приобретение базовых навыков многопоточного программирования.

Теоретические сведения

Многопоточность в Java позволяет добиться параллельного выполнения нескольких потоков в рамках одной программы, эффективно используя ресурсы процессора. Вместо последовательного выполнения задач потоки работают как бы одновременно, разделяя общее адресное пространство процесса. Это позволяет ускорить выполнение сложных задач, поскольку переключение между потоками происходит быстрее, чем между отдельными процессами.

Поток – это минимальная единица исполнения в Java, часто называемая «легковесным процессом». Потоки существуют внутри процесса и совместно используют его память.

Для координации и взаимодействия между потоками в Java предусмотрены механизмы, основанные на методах `wait()`, `notify()` и `notifyAll()`. Это позволяет потокам сигнализировать друг другу о событиях и синхронизировать свою работу.

Жизненный цикл потока в Java включает несколько состояний:

- **New (Новый):** Поток создан, но еще не запущен.
- **Running (Выполняется):** Поток активен и выполняется процессором.

В процессе выполнения поток может переходить в другие состояния.

- **Suspended (Приостановлен):** Поток временно приостановлен, но может быть возобновлен с точки остановки.

- **Blocked (Заблокирован):** Поток ожидает освобождения ресурса или завершения операции ввода-вывода, не потребляя процессорное время.

- **Terminated (Прерван):** Поток принудительно завершен. Возобновление невозможно. Причины могут быть разными: необработанное исключение или ошибка.

- **Dead (Мертвый):** Поток завершил свою работу (успешно или аварийно) и больше не может быть использован. Это конечное состояние жизненного цикла потока.

Способы запуска потоков

Приложение, создающее экземпляр класса Thread, должно предоставить код, который будет работать в этом потоке. Существует два способа, чтобы добиться этого:

– Предоставить реализацию объекта Runnable. Интерфейс Runnable определяет единственный метод – run, который должен содержать код, выполняющийся в потоке. Объект Runnable передается конструктору Thread. Например:

```
public class ExampleRunnable implements Runnable {
    public void run() {
        System.out.println("Hello!");
    }
    public static void main(String args[]) {
        (new Thread(new ExampleRunnable ())).start();
    }
}
```

– Использовать подкласс Thread. Класс Thread сам реализует Runnable, хотя его метод run не делает ничего. Можно объявить класс Thread подклассом, предоставляя собственную реализацию метода run, например:

```
public class ExampleThread extends Thread {
    public void run() {
        System.out.println("Hello!");
    }
    public static void main(String args[]) {
        (new ExampleThread()).start();
    }
}
```

Метод Thread.sleep()

Поток может остановиться сам, вызвав статический метод Thread.sleep(). Thread.sleep() принимает в качестве параметра количество миллисекунд. Метод sleep() попытается заснуть на это количество времени перед возобновлением выполнения. Thread sleep() не гарантирует абсолютной точности.

Завершение работы потока

Поток будет автоматически завершен после выхода из метода run(), того самого, что определен в интерфейсе Runnable. Это наиболее распространенный способ завершения работы потока.

Разберем следующий пример (листинг 41):

Листинг 41

```
public class Main extends Thread {
    public static int amount = 0;

    public static void main(String[] args) {
        Main thread = new Main();
        thread.start();
        // Wait for the thread to finish
        while(thread.isAlive()) {
            System.out.println("Waiting...");
        }
        // Update amount and print its value
        System.out.println("Main: " + amount);
        amount++;
        System.out.println("Main: " + amount);
        System.out.println("Thread is dead...");
    }
    public void run() {
        amount++;
    }
}
```

Здесь класс Main наследует Thread, и, соответственно, переопределяет метод run(), в котором и будет выполняться код параллельного потока. В нем просто увеличиваем статическую переменную amount на 1 для демонстрации того, что код сработал. В методе main() создаем экземпляр класса Main и запускаем параллельный поток (thread.start()). Далее идет цикл ожидания, в котором наш основной поток ожидает окончания работы параллельного и только после смерти этого параллельного потока идет выполнять дальнейший код.

Вывод в консоль (результат работы программы) будет следующим:

```
Waiting...
Main: 1
Main: 2
Thread is dead...
```

Из этого очевидно, что значение переменной amount при первом выводе в консоль уже увеличено на 1 в параллельном потоке, поскольку изначально amount=0. Также, этот пример показывает, что при окончании работы метода run(), метод thread.isAlive() (т.е. проверка на состояние нити) больше не выдает true, т.е. нить благополучно умирает и будет удалена из памяти сборщиком мусора.

Разберем еще один пример (листинг 42).

Листинг 42

```
public class Main {
    public static void main(String[] args) throws Exception{
        TestClass thread = new TestClass("Thread 1", 100);
        thread.start();
        TestClass thread2 = new TestClass("Thread 2", 100);
        thread2.start();
        TestClass thread3 = new TestClass("Thread 3", 100);
        thread3.start();
        // Wait for the thread to finish
        while(thread.isAlive() || thread2.isAlive() || thread3.isAlive()) {
            Thread.sleep(100);
            System.out.println("->Waiting...");
        }
        System.out.println("All threads are dead");
    }
}

class TestClass extends Thread{
    private int amount;
    private String id;
    private int time;
    public TestClass(String id, int time){
        this.id = id;
        this.time = time;
    }

    public void run() {
        // Update amount and print its value
        for(int i=0;i<3;i++){
            try{
                Thread.sleep(time);
            }catch(Exception e){
                ;
            }
            amount++;
            System.out.println(id + "->" + amount);
        }
    }
}
```

В этом примере 2 класса: основной (Main) и класс для реализации параллельного потока (TestClass).

Класс TestClass принимает на вход конструктора свой идентификатор (будет позже выводиться в консоль для идентификации экземпляра потока) и время задержки в ms. для большей наглядности работы. Сам метод run(), содержит простой цикл на 3 итерации, в каждой из которых спит некоторое время (в данном случае 100 ms), потом увеличивает локальную переменную на 1 и выводит в консоль запись, содержащую идентификатор потока и текущее значение amount. Это выглядит приблизительно так: «Thread 1->1».

В классе Main последовательно создаются 3 экземпляра класса TestClass и запускаются, затем идет цикл (while()), работающий пока хотя бы один из экземпляров TestClass еще жив. В этом цикле засыпает на те же 100 ms., и пишет в консоль “->Waiting”. После того как все экземпляры TestClass завершат свою работу, основной поток сообщает в консоль, что все параллельные потоки мертвы и сам завершает свою работу.

Результат в консоли выглядит следующим образом (листинг 43):

Листинг 43

```
->Waiting...
Thread 1->1
Thread 3->1
Thread 2->1
->Waiting...
Thread 1->2
Thread 3->2
Thread 2->2
->Waiting...
Thread 1->3
Thread 3->3
Thread 2->3
->Waiting...
All threads are dead
```

Порядок выполнения работы

1. Создать класс, расширяющий Thread.

- 1) Создать класс **MyThread** наследующий класс Thread.
- 2) Переопределить метод **run()**. В цикле for вывести на консоль символ 50 раз.
- 3) Создать экземпляр класса **MyThread** и запустить новый поток.

2. Создать класс реализующий интерфейс Runnable.

- 1) Создать класс, реализующий интерфейс **Runnable**.
- 2) Переопределить **run()** метод. Создать цикл **for**. В цикле распечатываем значения от 0 до 120 делящиеся на 9 без остатка.
- 3) Используем статический метод **Thread.sleep()**, чтобы сделать паузу.
- 4) Создать три потока, выполняющих задачу распечатки значений.

3. Вывод символа в трех потоках.

Создать три потока, которые изменяют один и тот же объект. Каждый поток должен вывести на экран одну букву 120 раз, затем увеличить значение символа на 1.

- 1) Создать класс, расширяющий **Thread**.
- 2) Переопределить метод **run()** – здесь будет находиться синхронизированный блок кода.
- 3) Для того чтобы три объекта потока имели доступ к одному объекту, создаем конструктор принимающий на вход объект **StringBuilder**.
- 4) Синхронизированный блок кода будет получать монитор на объект **StringBuilder** из п. 3.
- 5) Внутри синхронизированного блока кода выведите **StringBuilder** на экран 100 раз, а потом увеличьте значение символа на 1.
- 6) В методе **main()** создайте один объект класса **StringBuilder**, используя символ 'а'.
- 7) Затем создайте три экземпляра объекта нашего класса и запустите потоки.

Порядок выполнения работы

1. Изучить основные теоретические положения.
2. Получить задание у преподавателя, выполнить задание.
3. Оформить отчет.

Требования к отчету

1. Цель работы.
2. Результаты работы.
3. Выводы.

Лабораторная работа 5

МНОГОПОТОЧНОЕ ПРОГРАММИРОВАНИЕ.

СИНХРОНИЗАЦИЯ ПОТОКОВ

Цель работы: приобретение навыков синхронизации нитей в многопоточном программировании.

Теоретические сведения

Для синхронизации выполнения различных нитей существуют методы **wait**, **notify**, **notifyAll**. Эти методы были придуманы как часть механизма синхронизации нитей.

Java предоставляет встроенные средства для синхронизации доступа к общим ресурсам (объектам) в многопоточной среде. Механизм `synchronized` позволяет потоку получить эксклюзивный доступ к объекту, предотвращая одновременный доступ других потоков. Пока объект «захвачен» одним потоком, другие потоки, пытающиеся получить к нему доступ, будут заблокированы до тех пор, пока первый поток не освободит объект. Пример использования может выглядеть следующим образом:

```
public void doSomething()
{
    Object monitor = getMonitor();
    synchronized(monitor)
    {
        System.out.println("text");
    }
}
```

В методе `doSomething()` реализована синхронизация доступа к ресурсу. Получив объект `monitor`, метод использует блок `synchronized(monitor)` для создания критической секции. Когда поток входит в этот блок, он «захватывает» монитор, делая его недоступным для других потоков. Если другой поток попытается вызвать `doSomething()` в это же время, он будет заблокирован и будет ждать, пока первый поток не выйдет из `synchronized` блока и не освободит монитор. Важно отметить, что один и тот же объект `monitor` может быть использован для синхронизации доступа к другим ресурсам в программе, обеспечивая согласованность данных.

Монитором называется объект, который хранит состояние занят/свободен. Для управления этими состояниями используются методы **wait** и **notify**.

В многопоточных программах иногда возникает такая ситуация: нить захватывает монитор (входит в `synchronized` блок), но не может продолжить работу из-за отсутствия необходимых данных (например, ожидается загрузка файла). Реализация ожидания в цикле с периодической проверкой условия (например, «спим секунду и проверяем») неэффективна. Пока нить удерживает монитор, другие нити, готовые к работе, вынуждены простаивать.

Для решения этой проблемы используется механизм `wait()`. Вызов `wait()` у объекта-монитора (внутри `synchronized` блока, связанного с этим монитором) приводит к тому, что текущая нить временно приостанавливает свое выполнение и освобождает монитор. Это позволяет другим нитям получить доступ к монитору и продолжить работу.

Таким образом, несколько нитей могут оказаться в состоянии ожидания, вызванном `wait()` для одного и того же монитора.

Для возобновления работы нитей, ожидающих на мониторе, используются методы `notify()` и `notifyAll()`. Эти методы также должны вызываться внутри `synchronized` блока, связанного с тем же монитором, на котором нити ожидают. `notifyAll()` пробуждает все нити, находящиеся в состоянии ожидания на данном мониторе, а `notify()` – только одну, выбранную случайным образом.

Пример кода – листинг 44.

Листинг 44

```
import java.util.*;
public class Test{
    private static Vector<Task> tasks = new Vector<Task>();
    public static void main(String[] args) {
        Worker worker = new Worker(tasks);
        worker.start();
        for(int i = 1; i <= 3; i++){
            tasks.add(new Task(i));
            notifyWorker(worker);
            sleep(200);
        }
        System.out.println("is worker alive: " + worker.isAlive());
        System.out.println("\n=====");
        for(int i = 4; i <= 6; i++){
            tasks.add(new Task(i));
            notifyWorker(worker);
        }
        for(int i = 1; i <= 6; i++){
            System.out.println("Main loop...");
            sleep(50);
        }
    }
}
```

```

    }
        sleep(1000);
        worker.terminate();
        notifyWorker(worker);
        System.out.println("is worker alive: " + worker.isAlive());
        sleep(1000);
        System.out.println("is worker alive: " + worker.isAlive());
    }
    private static void notifyWorker(Worker worker){
        synchronized (worker){
            worker.notify();
        }
    }
    public static void sleep(int time){
        try {
            Thread.sleep(time);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
//-----класс Task-----
private static class Task{
    int id;
    public Task(int id){
        this.id = id;
    }
    public void doTask(){
        Test.sleep(100);
        System.out.println("Task "+ "" + id + " is done");
    }
}
//-----класс Worker-----
private static class Worker extends Thread{
    private boolean isTerminated = false;
    private List<Task> tasks;
    public Worker(List<Task> tasks){
        this.tasks = tasks;
    }
    public void terminate(){
        isTerminated = true;
    }
    @Override
    public void run(){
        while (!isTerminated) {
            if (tasks.isEmpty()) {
                synchronized (this) {
                    try {

```

```

        System.out.println("Worker is sleeping now");
        wait();
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    }
    } else{
        tasks.removeFirst().doTask();
    }
    }
    System.out.println("Worker is down");
}
}
}
}

```

В этом примере (см. листинг 44) 3 класса: основной класс Test, классы Task и Worker. Класс Task содержит некую задачу, которая должна быть выполнена, в основном классе Test происходит накопление задач (экземпляров класса Task) в контейнере tasks. Класс Worker служит для асинхронного выполнения задач (обработки экземпляров класса Task).

Приложение работает следующим образом: по мере появления новых задач (объекты класса Task) они добавляются в контейнер tasks, после чего класс Worker получает вызов notify(), извещающий его о том, что есть новые данные для обработки. Worker просыпается, смотрит в список задач (ссылка на контейнер tasks была передана Worker-у в конструкторе) и выполняет все доступные в списке задачи, после чего снова засыпает до появления новых задач.

После выполнения всех задач главного класса (Test), работающий экземпляр класса Worker уничтожается вызовом Worker.terminate(), после чего работа программы завершается.

Порядок выполнения работы

1. Изучить основные теоретические положения.
2. Получить задание у преподавателя, выполнить задание.
3. Оформить отчет.

Требования к отчету

1. Цель работы.
2. Результаты работы.
3. Выводы.

Лабораторная работа 6

МНОГОПОТОЧНОЕ ПРОГРАММИРОВАНИЕ.

СИНХРОНИЗАЦИЯ РЕЗУЛЬТАТОВ РАБОТЫ НИТЕЙ

Цель работы: приобретение навыков синхронизации результатов работы нитей в многопоточном программировании с использованием интерфейса обратного вызова.

Теоретические сведения

Одним из распространенных способов использования интерфейсов в Java является создание обратного вызова. Суть обратного вызова состоит в том, что мы создаем действия, которые вызываются при других действиях. Т.е. одни действия вызываются другими действиями. В случае многопоточного программирования зачастую нужно в главном потоке программы оперативно получать результаты выполнения множества одновременно выполняемых параллельных потоков. Интерфейсы обратного вызова помогают решать эту проблему.

Рассмотрим следующий пример (листинг 45).

Листинг 45

```
import java.util.*;
import java.util.concurrent.locks.*;
public class Main extends Thread {
    public static int amount = 0;
    public static List<Integer> resList;
    public static void main(String[] args) throws Exception{
        resList = new ArrayList<Integer>();
        Result result = new Result();
        Worker thread = new Worker(result, "Thread 1", 300);
        thread.start();
        Worker thread2 = new Worker(result, "Thread 2", 300);
        thread2.start();
        Worker thread3 = new Worker(result, "Thread 3", 300);
        thread3.start();
        // Wait for the thread to finish
        while(thread.isAlive() || thread2.isAlive() || thread3.isAlive()) {
            Thread.sleep(300);
            System.out.println("->Waiting...");
        }
        System.out.println("->All threads done...");
    }
}
```



```

        System.out.println("Result list content->");
        System.out.println(Arrays.toString(resList.toArray(new Integer[resList.size()]));
    }
}
//-----Result-----
class Result implements TaskDoneI{
    Lock lock;
    public Result(){
        lock = new ReentrantLock();
    }
    public void taskIsDone(String id, int res){
        try{
            lock.lock();
            Main.resList.add(res);
            System.out.println("Result class: new result registered from " + id + "->" + res);
        }finally{
            lock.unlock();
        }
    }
}
//-----Worker-----
class Worker extends Thread{
    private int amount;
    private String id;
    private int time;
    private TaskDoneI result;
    public Worker(TaskDoneI result, String id, int time){
        this.result = result;
        this.id = id;
        this.time = time;
    }

    public void run() {
        // Update amount and print its value
        for(int i=0;i<3;i++){
            try{
                Thread.sleep(time);
            }catch(Exception e){
                ;
            }
            amount++;
            result.taskIsDone(id, amount);
            System.out.println(id + "->" + amount);
        }
    }
}

```

```

}
//-----Interface-----
interface TaskDoneI{
    void taskIsDone(String id, int res);
}

```

Результат работы программы в консоли вывода выглядит следующим образом (листинг 46):

Листинг 46

```

->Waiting...
Result class: new result registered from Thread 1->1
Result class: new result registered from Thread 2->1
Thread 1->1
Result class: new result registered from Thread 3->1
Thread 3->1
Thread 2->1
->Waiting...
Result class: new result registered from Thread 1->2
Thread 1->2
Result class: new result registered from Thread 3->2
Thread 3->2
Result class: new result registered from Thread 2->2
Thread 2->2
->Waiting...
Result class: new result registered from Thread 1->3
Thread 1->3
Result class: new result registered from Thread 3->3
Thread 3->3
Result class: new result registered from Thread 2->3
Thread 2->3
->Waiting...
->All threads done...
Result list content->
[1, 1, 1, 2, 2, 2, 3, 3, 3]

```

Здесь имеется главный класс Main, из которого организуется вся работа. В качестве интерфейса обратного вызова объявлен интерфейс **TaskDoneI** с единственным методом **void taskIsDone(String id, int res)**, а также класс, его реализующий – **Result**. Далее, есть класс **Worker**, который может выполнять некую задачу, ему на вход конструктора подаются исходные данные для его работы, а также ссылка на экземпляр класса Result. При этом следует обратить внимание на то, что вместо типа **Result**, в конструкторе указан интерфейс **TaskDoneI**, что существенно

увеличивает гибкость в использовании класса `Worker`, поскольку теперь для регистрации результатов можно использовать любую реализацию интерфейса `TaskDoneI`, не ограничиваясь только классом `Result`.

В главном классе для демонстрации работы создается 3 экземпляра класса `Worker`. На вход каждому передается уникальный идентификатор нити (по нему можно определить, какая из нитей выполнила какие-то действия исходя из сообщений в логге консоли), также передается ссылка на единственный объект класса `Result`, который и будет собирать результаты работы всех нитей. Далее все 3 `Worker`-а одновременно запускаются, после чего в главном классе происходит ожидание прекращения работы всех нитей `Worker`.

По окончании работы каждой нити, для регистрации результатов, происходит вызов метода регистрации через интерфейс – `result.taskIsDone(id, amount)`.

Следует обратить внимание на особенность организации метода `taskIsDone()` в классе `Result`. Очевидно, что этот метод должен быть синхронизирован (например, помечен как `synchronuzed`), поскольку его одновременно могут вызывать несколько потоков. В данном случае используется другой, более современный и эффективный, способ синхронизации с использованием механизма блокировок (локов – `Locks`). Здесь используется реализация `ReentrantLock`.

`ReentrantLock` выполняет те же функции, что и ключевое слово `synchronized`, обеспечивая взаимное исключение при доступе к общим ресурсам. Поток, пытающийся получить доступ к ресурсу, защищенному `ReentrantLock`, приостанавливается на вызове `lock()`, пока не придет его очередь на владение этим «замком». Как и в случае с `synchronized`, только один поток может владеть `ReentrantLock` в любой момент времени. Вызов `unlock()` освобождает «замок», позволяя другим потокам претендовать на него, аналогично выходу из `synchronized` блока.

Однако, в отличие от `synchronized`, `ReentrantLock` предоставляет более богатый набор инструментов для мониторинга состояния блокировки. Можно узнать, занят ли «замок» в данный момент, сколько потоков ожидают его освобождения и сколько раз текущий поток уже захватил этот «замок».

`ReentrantLock` также предлагает более гибкие режимы блокировки. Помимо стандартного `lock()`, который блокирует поток до получения «замка», метод `tryLock()` позволяет потоку попытаться получить «замок» в течение определенного времени или немедленно, если «замок» свободен.

Наконец, `ReentrantLock` может быть настроен как «справедливый» (`fair`). В этом режиме потоки получают доступ к «замку» в порядке их прибытия в очередь ожидания. `synchronized` не предоставляет никаких гарантий относительно порядка доступа потоков к ресурсу.

Также важным моментом является использование последовательности вызова `lock()/unlock()` в блоке `try/finally`. Вызов `unlock()` должен происходить именно в `finally` потому, что в этом случае можно гарантировать вызов `unlock()` несмотря ни на какие возможные выбросы исключений в предваряющем коде. В противном случае, этот участок кода может оказаться навсегда заблокирован в случае возникновения исключения.

Порядок выполнения работы

1. Изучить основные теоретические положения.
2. Получить задание у преподавателя, выполнить задание.
3. Оформить отчет.

Требования к отчету

1. Цель работы.
2. Результаты работы.
3. Выводы.

Лабораторная работа 7

РАБОТА С ФАЙЛАМИ

Цель работы: изучить базовые возможности языка программирования Java по работе с файлами.

Теоретическая часть

Для организации потоков ввода-вывода используются четыре базовых абстрактных класса: `InputStream` и `OutputStream` для работы с байтами, а `Reader` и `Writer` – для работы с символами.

Для взаимодействия с файлами на основе этих абстрактных классов созданы классы `FileInputStream`, `FileOutputStream`, `FileReader` и `FileWriter`. Эти классы выступают в роли адаптеров, позволяющих объектам класса `File` использовать методы, определенные в интерфейсах `InputStream`, `OutputStream`, `Reader` и `Writer`.

Рассмотрим паттерн проектирования «Адаптер». Адаптер (класс A) реализует интерфейс B, подстраивая под него объект другого класса (C). В классе-адаптере A содержится поле, хранящее экземпляр класса C.

В нашем случае объект `File` адаптируется к потоку ввода `InputStream`. Это означает, что данные из `File` извлекаются через `InputStream`. Фактически мы работаем с `InputStream` через адаптер `FileInputStream`, который, с одной стороны, является наследником `InputStream`, а с другой – содержит ссылку на объект `File`.

Адаптер берет на себя задачу извлечения данных из файла и преобразования их в формат, совместимый с методами `InputStream`. Класс `FileInputStream` (в данном примере – адаптер) переопределяет методы `InputStream`, добавляя в них логику, необходимую для работы с файлом.

Сначала создается объект, для которого требуется адаптер. Затем создается переменная класса, к которому выполняется адаптация. Этой переменной присваивается объект класса-адаптера, в конструктор которого передается адаптируемый объект.

```
File file = new File("/home/user/pic.jpg");
InputStream fIn = new FileInputStream(file);
```

Также, переменную можно определить самим классом-адаптером:

```
FileInputStream fIn = new FileInputStream(file);
```

В конструктор можно передать строку-путь к файлу. Объект File будет создан внутри адаптера. Пример побайтового копирования файла (рисунок 10):

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class InputOutputStream {
    public static void main(String[] args)
        throws IOException {

        FileInputStream fileIn =
            new FileInputStream(
                "src/file/pets.png");
        FileOutputStream fileOut =
            new FileOutputStream(
                "src/file/pets2.png");

        while (fileIn.available() > 0) {
            int oneByte = fileIn.read();
            fileOut.write(oneByte);
        }
        fileIn.close();
        fileOut.close();
    }
}
```

Рисунок 10

Если используются относительные адреса, они должны начинаться от корня проекта. При работе с файлами класс FileOutputStream позволяет не только создавать новые файлы для записи, но и добавлять данные в существующие. Если при создании FileOutputStream указать true в качестве второго аргумента, то новые данные будут дописаны в конец файла, а не перезапишут его содержимое.

Класс FileInputStream предоставляет методы для чтения данных из файла. Метод available() позволяет узнать, сколько байт еще не прочитано. Метод read()

может читать данные двумя способами: по одному байту (возвращая его в виде `int`) или блоками, записывая прочитанные байты в массив и возвращая количество фактически прочитанных байт. Аналогично, `write()` позволяет записывать данные в файл не только по одному байту, но и блоками.

```
byte[] blockBytes = new byte[100];

while (fileIn.available() > 0) {
    int qtyBytes = fileIn.read(blockBytes);
    fileOut.write(blockBytes, 0, qtyBytes);
}
```

При чтении конца файла блок может содержать меньше прочитанных байт, чем размерность массива. Поэтому `write()` позволяет указывать срез массива.

При работе с файлами, для записи данных можно использовать `FileOutputStream`. Метод `flush()` этого класса позволяет немедленно сбросить данные из буфера на диск, а метод `close()` делает это автоматически при закрытии потока. Альтернативно для записи в файлы можно использовать класс `PrintStream`, который является потомком `OutputStream` через `FilterOutputStream` (рисунок 11).

```
import java.io.FileNotFoundException;
import java.io.PrintStream;

public class PrintStreamTest {
    public static void main(String[] args)
        throws FileNotFoundException {

        PrintStream fileOut = new PrintStream(
            "src/file/text.txt");
        fileOut.println(10.5);
        fileOut.printf(
            "%s - %d - %f", "hi", 10, 1.1);
        fileOut.close();
    }
}
```

Рисунок 11

При работе с файлами, часто используется паттерн проектирования «Декоратор» (или «Обертка»). Этот паттерн позволяет добавлять новую функциональность к существующим объектам, не изменяя их структуру. Вместо того чтобы адаптировать объект к другому интерфейсу, декоратор расширяет его возможности.

Класс-декоратор наследуется от того же класса или интерфейса, что и декорируемый объект. Он переопределяет методы этого объекта, добавляя к ним новую логику. При этом, внутри переопределенных методов, декоратор вызывает оригинальные методы декорируемого объекта, а затем выполняет дополнительные действия.

Для использования декоратора, создается экземпляр декорируемого класса и передается в конструктор класса-декоратора. Внутри декоратора хранится ссылка на декорируемый объект.

Примером такого декоратора является `BufferedInputStream`, который расширяет возможности `InputStream`. `BufferedInputStream` позволяет читать данные из файла большими блоками в буфер, что повышает производительность за счет уменьшения количества обращений к файловой системе. Аналогично существует `BufferedOutputStream` для записи данных (рисунок 12).

```
import java.io.BufferedInputStream;
import java.io.FileInputStream;
import java.io.IOException;

public class BufferStream {
    public static void main(String[] args)
        throws IOException {

        FileInputStream fileIn =
            new FileInputStream("src/file/text.txt");
        BufferedInputStream bufIn =
            new BufferedInputStream(fileIn, 100);

        int i;
        while((i = bufIn.read()) != -1){
            System.out.print((char)i);
        }
    }
}
```

Рисунок 12

Конструктор класса `BufferedInputStream` принимает объект `InputStream`.

Данные считываются блоками, но метод `read()` извлекает их по одному, но уже из буфера.

С помощью классов `FileReader` и `FileWriter` выполняется ввод-вывод в текстовые файлы (рисунок 13).

```
FileReader reader = new FileReader(
    "src/file/text.txt");
FileWriter writer = new FileWriter(
    "src/file/text2.txt");

while (reader.ready()) {
    int c = reader.read();
    writer.write(c);
}

reader.close();
writer.close();
```

Рисунок 13

Метод `ready()` возвращает `true`, если не все символы прочитаны.

Читать и писать можно блоками. Также методу `write()` можно передать строку (рисунок 14):

```
FileReader reader = new FileReader(
    "src/file/text.txt");
FileWriter writer = new FileWriter(
    "src/file/text3.txt");

char[] buff = new char[10];

while (reader.ready()) {
    int qtySymbols = reader.read(buff);
    writer.write(buff, 0, qtySymbols);
}

writer.write("Halo");

reader.close();
writer.close();
```

Рисунок 14

Порядок выполнения работы

1. Изучить основные теоретические положения.
2. Получить задание у преподавателя, выполнить задание.
3. Оформить отчет.

Требования к отчету

1. Цель работы.
2. Результаты работы.
3. Выводы.

Лабораторная работа 8

ИСКЛЮЧЕНИЯ В JAVA

Цель работы: изучить принципы обработки, создания, использования исключений в Java. Научиться работать со строками.

Теоретическая часть

В Java исключения – это способ обработки нештатных ситуаций или ошибок, возникающих во время выполнения программы. Вместо ручной проверки на каждую возможную ошибку Java предоставляет механизм исключений, который делает код более чистым, надежным и удобным в сопровождении. Когда в программе возникает ошибка (например, попытка деления на ноль), создается объект исключения. Этот объект может быть обработан непосредственно в месте возникновения ошибки, либо «переброшен» выше по стеку вызовов. В любом случае, исключение должно быть «поймано» и обработано.

Исключения могут быть двух типов:

- **системные исключения:** возникают из-за ошибок в использовании языка Java или при попытке выполнить недопустимые операции;
- **пользовательские исключения:** создаются программистом для обработки специфических ошибок, возникающих в конкретной программе.

Вместо того чтобы пытаться «заглушить» ошибку (например, при делении на ноль), часто более правильно «выбросить» исключение, сигнализируя о том, что операция не может быть выполнена корректно.

Для работы с исключениями в Java используются пять ключевых слов: try, catch, throw, throws и finally.

Пример кода:

```
1  try
2  {
3      // блок кода, где отслеживаются ошибки
4  }
5  catch (тип_исключения_1 exceptionObject)
6  {
7      // обрабатываем ошибку
8  }
9  catch (тип_исключения_2 exceptionObject)
10 {
11     // обрабатываем ошибку
12 }
13 finally
14 {
15     // код, который нужно выполнить после завершения блока try
16 }
```

Процесс обработки исключений выглядит следующим образом:

1. Код, который может вызвать исключение, помещается в блок try.
2. Если в блоке try возникает исключение, создается объект исключения и происходит его «выброс».
3. Блок catch перехватывает исключение и выполняет код, необходимый для его обработки.
4. Для ручного «выброса» исключения используется ключевое слово throw.
5. Метод, который может «выбросить» исключение, должен указать это в своей сигнатуре с помощью ключевого слова throws.
6. Блок finally содержит код, который должен быть выполнен в любом случае, независимо от того, было ли выброшено исключение или нет.

В Java все исключения являются потомками класса Throwable. Этот класс имеет два основных подкласса:

- **Exception:** используется для обработки исключений, возникающих в программе. Можно создавать собственные классы исключений, наследуясь от Exception. Существует также подкласс RuntimeException для распространенных ошибок, таких как деление на ноль или выход за границы массива.
- **Error:** используется для обработки ошибок, возникающих в самой виртуальной машине Java. Как правило, программисту не требуется обрабатывать ошибки этого типа.

Прежде чем начать обрабатывать исключения, посмотрим на реакцию виртуальной машины на необработанное исключение, например, при делении на ноль. Фрагмент кода:

```
1 package by.psu.fit.cmsn.cross_pp.sample1.main;
2
3 public class SampleApp
4 {
5     public static void main(String... args)
6     {
7         int catNumber;
8         int zero;
9         catNumber = 1; // у меня один кот
10        zero = 0; // ноль, он и в Африке ноль
11        int result = catNumber / zero;
12    }
13 }
```

При обнаружении операции деления на ноль JVM генерирует исключение. Если в программе не предусмотрен блок catch для обработки этого исключения, управление передается стандартному обработчику исключений Java.

Этот обработчик завершает выполнение программы и выводит информацию об ошибке в стандартный поток вывода (обычно терминал).

```
run:
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at by.psu.fit.cmsn.cross_pp.sample1.main.SampleApp.main(SampleApp.java:11)
Java Result: 1
BUILD SUCCESSFUL (total time: 0 seconds)
```

Как видно, созданный объект исключения принадлежит к классу `ArithmeticException`. Далее следует краткое описание ошибки и место возникновения ошибки.

Поместим проблемный код в блок `try`, а в блоке `catch` обработаем исключение в следующем фрагменте кода:

```
1  public class SampleApp
2  {
3      public static void main(String... args)
4      {
5          int catNumber;
6          int zero;
7
8          try
9          {
10             // мониторим код
11             catNumber = 1; // у меня один кот
12             zero = 0; // ноль, он и в Африке ноль
13             int result = catNumber / zero;
14             System.out.println("Не увидите это сообщение!");
15         }
16         catch (ArithmeticException e)
17         {
18             System.out.println("Нельзя котов делить на ноль!");
19         }
20
21         System.out.println("Жизнь продолжается");
22     }
23 }
```

Теперь программа аварийно не закрывается, т.к. мы обрабатываем ситуацию с делением на ноль.

Поскольку мы уже знали, к какому классу принадлежит исключение, то в блоке `catch` сразу указали конкретный тип. Обратите внимание, что последний оператор в блоке `try` не срабатывает, поскольку ошибка происходит раньше строчкой выше. Далее выполнение передается в блок `catch`, после чего выполняются следующие операторы в обычном порядке.

Операторы `try` и `catch` работают совместно в паре. Хотя `catch` может обрабатывать несколько вложенных операторов `try`.

Несколько исключений

Один блок try может содержать код, который потенциально может выбросить несколько разных типов исключений. Чтобы обработать их все, необходимо использовать несколько блоков catch, каждый из которых предназначен для конкретного типа исключения. Эти блоки catch проверяются последовательно. Если исключение соответствует типу, указанному в первом блоке catch, то выполняется код именно этого блока. Остальные блоки catch при этом игнорируются, и выполнение программы продолжается с кода, следующего сразу после всей конструкции try-catch.

Пример:

```
1 public class SampleApp
2 {
3     public static void main(String... args)
4     {
5         int catNumber;
6         int zero;
7
8         try
9         {
10            catNumber = 1; // у меня один кот
11
12            zero = 1; // ноль, он и в Африке ноль
13
14            int result = catNumber / zero;
15
16            // Создадим массив из трёх котов
17            String[] catNames = {"Васька", "Барсик", "Мурзик"};
18
19            catNames[3] = "Рыжик";
20
21            System.out.println("Не увидите это сообщение!");
22        }
23        catch (ArithmeticException e)
24        {
25            System.out.println(e.toString() + ": Нельзя котов делить на ноль!");
26        }
27        catch (ArrayIndexOutOfBoundsException e)
28        {
29            System.out.println("Ошибка: " + e.toString());
30        }
31
32        System.out.println("Жизнь продолжается");
33    }
34 }
```

Можно обрабатывать несколько типов исключений в одном блоке try-catch. Представьте, у вас есть код, который может вызвать ошибку деления на ноль (ArithmeticException) или выход за границы массива (ArrayIndexOutOfBoundsException).

Пример: мы пытаемся обратиться к четвертому элементу массива из трех элементов (индексы начинаются с нуля). Если при этом еще и происходит деление на ноль, то сработает только первый catch, если он стоит первым.

Важно помнить порядок catch-блоков. Обработчики более конкретных исключений (например, `ArithmeticException`) должны идти перед обработчиками более общих исключений (например, `Exception`). Если первым будет `catch (Exception e)`, то он перехватит все исключения, и остальные `catch`-блоки никогда не сработают. К счастью, IDE обычно предупреждает о такой ошибке. Если видите такое предупреждение, просто поменяйте порядок `catch`-блоков.

Вложенные операторы try

Блоки `try` могут быть вложены друг в друга. Если внутри вложенного блока `try` возникает исключение, и у этого блока нет своего блока `catch`, то Java будет искать блок `catch` для обработки этого исключения во внешнем блоке `try`. Этот поиск продолжается вверх по уровням вложенности. Если подходящий блок `catch` так и не будет найден, то исключение будет обработано системой по умолчанию.

Оператор throw

Часть исключений может обрабатывать сама система. При помощи оператора `throw` можно «выбросить» исключение из метода. Код будет таким: `throw` экземпляр класса `Throwable`.

При использовании `throw` для генерации исключения, необходимо создать объект класса `Throwable` или его подкласса. Этот объект можно получить либо в блоке `catch`, либо создать напрямую с помощью `new`. Когда выбрасывается исключение, выполнение текущего кода немедленно прекращается, и Java ищет ближайший блок `try/catch`, способный обработать исключение данного типа. Например:

```
1 public class SampleApp
2 {
3     public static void main(String... args)
4     {
5         try
6         {
7             throw new NullPointerException("Кота не существует");
8         }
9         catch (NullPointerException e)
10        {
11            System.out.println(e.getMessage());
12        }
13    }
14 }
```

Мы создали новый объект класса **NullPointerException**. В конструктор передали описание исключения. Получить текст из него можно через метод **getMessage()**, что мы и сделали в блоке **catch**.

Оператор throws

Если метод может породить исключение, которое он сам не обрабатывает, он должен позаботиться о том, чтобы вызывающий его код мог его обработать. Для этого к объявлению метода добавляется оператор **throws**, который перечисляет типы исключений (кроме исключений **Error** и **RuntimeException** и их подклассов).

Общая форма объявления метода с оператором **throws**, например:

```
тип имя_метода(список_параметров) throws список_исключений
{
    // код внутри метода
}
```

В фрагменте **список_исключений** можно указать список исключений через запятую.

Оператор finally

Блок **finally** гарантирует выполнение определенного участка кода, независимо от того, было ли выброшено исключение в блоке **try** или нет. Это особенно важно для освобождения ресурсов, таких как закрытие файлов или сетевых соединений. Блок **finally** выполняется после блока **try** и любых соответствующих блоков **catch**, но перед тем, как программа продолжит выполнение после конструкции **try-catch-finally**. Хотя блоки **catch** и **finally** не являются обязательными по отдельности, каждый блок **try** должен быть связан хотя бы с одним из них.

ЗАДАНИЕ К ЛАБОРАТОРНОЙ РАБОТЕ

1. Создать консольный интерфейс для динамического создания объектов из лабораторной работы 2 (Наследование).

2. В интерфейсе должны быть предусмотрены команды для создания различных типов объектов и ввода всех их параметров, а также команда просмотра списка уже созданных объектов.

3. В конструкторах и сеттерах объектов предусмотреть появление исключения в случае попытки записать в объект некорректное значение, создать обработчик данных исключений с корректными сообщениями пользователю.

4. Предусмотреть наличие нескольких (не менее 3) пользовательских исключений с их использованием и обработкой.
5. Получить дополнительное индивидуальное задание у преподавателя.

Порядок выполнения работы

1. Изучить основные теоретические положения.
2. Получить задание у преподавателя, выполнить задание.
3. Оформить отчет.

Требования к отчету

1. Цель работы.
2. Результаты работы.
3. Выводы.

ПЕРЕЧЕНЬ ВОПРОСОВ К ЗАЧЕТУ

1. Абстрактные классы и интерфейсы.
2. Шаблоны в языке Java.
3. Обработка исключений в языке Java.
4. Сборка мусора. Основные понятия. Особенности настройки.
5. Коллекции языка Java. Основные интерфейсы.
6. Потоки ввода-вывода в языке Java. Работа с файлами.
7. Назначение и область применения обобщений.
8. Многопоточное программирование в языке Java. Основные понятия.

Создание процессов и потоков.

9. Управление потоками. Синхронизация.
10. Конструкторы. Внутренние и анонимные классы.
11. Рефлексия в языке Java.

КОНТРОЛЬ КАЧЕСТВА УСВОЕНИЯ ЗНАНИЙ

Диагностика результатов учебной деятельности осуществляется следующими средствами:

- устный опрос на лекции;
- письменный отчет по лабораторной работе с его устной защитой;
- устный зачет.

Контроль качества усвоения знаний проводится в соответствии с Положением о рейтинговой системе оценки знаний и компетенций студентов (приказ ректора университета от 06.06.2014 № 294 (в редакции, утвержденной приказом ректора университета от 17.11.2014 № 605) в форме текущего контроля и промежуточной аттестации.

Результат текущего контроля за семестр оценивается отметкой в баллах по десятибалльной шкале и выводится, исходя из отметок, выставленных в ходе проведения мероприятий текущего контроля в течение семестра по следующей формуле:

$$T = \frac{(LP_1 + \dots + LP_8)}{8},$$

где $LP_1, \dots, LP_8$ – отметки, выставленные по результатам устных защит отчетов по лабораторным работам.

Результат текущего контроля рассчитывается как округленное среднее значение.

Промежуточная аттестация проводится в форме **зачета**.

Итоговая отметка о зачете формируется по формуле

$$З = k \cdot T,$$

где k – весовой коэффициент текущего контроля, $k = 1$;

T – результат текущего контроля за семестр.

Если итоговая отметка $З < 4$ баллов, то проводится устный зачет отдельно по представленным в программе вопросам.

Перевод отметки по зачету осуществляется по следующим правилам: отметка «зачтено» выставляется студентам, получившим от 4 до 10 баллов, отметка «не зачтено» выставляется студентам, получившим от 1 до 3 баллов.

Литература

1. Эккель, Б. Философия Java = Thinking in Java / Б. Эккель ; пер. с англ. Е. Матвеева. – 4-е пол. изд. – СПб. : Питер, 2021. – 1165 с. : ил.
2. Курбатова, И. В. Основы программирования на языке Java : учеб. пособие / И. В. Курбатова, А. В. Печкуров. – СПб. : Лань, 2024. – 348 с. – Режим доступа: <https://e.lanbook.com/book/385925>
3. Пономарчук, Ю. В. Программирование на языке Java : учеб. пособие / Ю. В. Пономарчук, И. В. Кузнецов. – Хабаровск : ДГУПС, 2021. – 103 с. – Режим доступа: <https://e.lanbook.com/book/259451>
4. Николаев, Е. И. Объектно-ориентированное программирование : учеб. пособие / Е. И. Николаев. – Ставрополь: СКФУ, 2015. – 225 с. – Режим доступа: http://biblioclub.ru/index.php?page=book_red&id=458133
5. Курняван, Б. Создание web-приложений на языке Java с помощью сервлетов, JSP и EJB / Б. Курняван ; пер. О. Труфанова, науч. ред. А. Головки. – М. : Лори, 2013. – 880 с.
6. Строганкова, Н. В. Шаблоны программных платформ языка Java : практикум / Н. В. Строганкова, К. В. Касьяненко, А. В. Хозяинов. – М. : МИРЭА, 2021. – 83 с. – Режим доступа: <https://e.lanbook.com/book/182466>
7. Сурто, С. Г. Кросс-платформенное программирование : метод. указания к лабораторным работам для студентов специальности 1-40 02 01 «Вычислительные машины, системы и сети» / С. Г. Сурто. – Новополюк : ПГУ, 2022. – 195 с. – URL: <https://elib.psu.by/handle/123456789/31956>

Учреждение образования
«Полоцкий государственный университет имени Евфросинии Полоцкой»

УТВЕРЖДАЮ

Ректор учреждения образования
«Полоцкий государственный
университет имени
Евфросинии Полоцкой»

Ю.Я. Романовский
«28» 06 2024 г.
Регистрационный № УД - 78/24 /уч.

МОДУЛЬ «РАЗРАБОТКА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ»

КРОСС-ПЛАТФОРМЕННОЕ ПРОГРАММИРОВАНИЕ

Учебная программа учреждения образования
по учебной дисциплине для специальности
6-05-0611-05 «Компьютерная инженерия»
с профилизацией «Вычислительные машины, системы и сети»

2024 г.

Учебная программа составлена на основе образовательного стандарта по специальности высшего образования ОСВО 6-05-0611-05-2023 и учебного плана по специальности 6-05-0611-05 «Компьютерная инженерия» с профилизацией «Вычислительные машины, системы и сети». Регистрационный номер 18-23 / уч. ФИТ от 04.04.2023 г. для дневной формы получения высшего образования

СОСТАВИТЕЛЬ:

Сергей Геннадьевич Сурто, старший преподаватель кафедры вычислительных систем и сетей учреждения образования «Полоцкий государственный университет имени Евфросинии Полоцкой»

РЕКОМЕНДОВАНА К УТВЕРЖДЕНИЮ:

Кафедрой вычислительных систем и сетей учреждения образования «Полоцкий государственный университет имени Евфросинии Полоцкой» (протокол № 9 от 24.06 2024)

Методической комиссией факультета информационных технологий учреждения образования «Полоцкий государственный университет имени Евфросинии Полоцкой» (протокол № 6 от 26.06 2024)

СОДЕРЖАНИЕ УЧЕБНОГО МАТЕРИАЛА

Раздел 1

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ JAVA

Тема 1.1

Кроссплатформенное программирование, общие понятия

Общие понятия о кроссплатформенности, ее важности в современной индустрии разработки программного обеспечения. Популярные кроссплатформенные технологии. Технология Java. Виртуальная машина Java.

Тема 1.2

Основы программирования на языке Java

Примитивные типы данных. Преобразования базовых типов данных. Операции языка Java. Условные конструкции. Циклы в языке Java.

Раздел 2

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ В ЯЗЫКЕ JAVA

Тема 2.1

Объявление класса и атрибутов класса

Основные принципы объектно-ориентированного программирования. Понятие класса. Синтаксис объявления класса в языке Java. Модификаторы класса. Использование классов в основной программе. Члены класса. Модификаторы полей класса. Примеры использования. Объявление атрибутов класса.

Тема 2.2

Методы класса. Блоки. Конструкторы. Внутренние и анонимные классы

Объявление методов класса. Модификаторы методов. Переопределение методов. Блоки инициализации, их назначение и виды. Типы конструкторов. Особенности объявления внутренних классов. Назначение анонимных классов, объявление, примеры использования.

Тема 2.3

Абстрактные классы и интерфейсы

Реализация концепции абстракции в языках программирования. Место абстракций в архитектуре программных продуктов. Объявление абстрактного класса и его методов. Особенности модификаторов абстрактного класса. Последовательность инициализации блоков класса. Объявление интерфейса. Примеры использования интерфейсов. Отличия интерфейсов и абстрактных классов.

Раздел 3

СПЕЦИАЛЬНЫЕ ВОЗМОЖНОСТИ ЯЗЫКА JAVA

Тема 3.1

Шаблоны в языке Java

Концепция безопасного кода. Параметризация типов данных. Универсальные (generic) методы, универсальные (generic) интерфейсы. Маски (wildcards). Формальные параметры. Методы с шаблонами. Ограниченный неизвестный параметр. Соглашение по именам параметров.

Тема 3.2

Обработка исключений в языке Java

Организация обработки ошибок в языке Java. Виды исключений. Ключевые идентификаторы. Иерархия исключений. Дерево наследования классов обработки исключений. Конструкция try ... catch. Подклассы Exception.

Тема 3.3

Базовые объекты языка Java. Классы-обертки. Строки. Стандартные классы и утилиты

Основные Java пакеты. Классы-обертки. Работа со строками. Обзор утилитных классов.

Тема 3.4

Сборка мусора. Основные понятия. Особенности настройки и реализации в языке Java

Управление памятью в языке Java, принципы автоматического освобождения памяти. Понятие «сборка мусора». Время жизни объектов. Удаление объектов из памяти. Настройка и параметры сборки мусора в языке Java.

Тема 3.5

Коллекции языка Java, основные интерфейсы

Базовые интерфейсы. Интерфейс Collection. Итераторы. Интерфейс списка, множества, очереди.

Тема 3.6

Основные реализации коллекций языка Java. Синхронизированные коллекции

Реализации интерфейса List. Реализации интерфейса Set. Реализации интерфейса Queue. Реализации интерфейса Map. Устаревшие коллекции. Синхронизированные коллекции.

Тема 3.7

Потоки ввода-вывода в языке Java. Работа с файлами

Потоки ввода-вывода. Чтение и запись файлов. Буферизуемые потоки. Сериализация объектов. Работа с файловой системой.

Тема 3.8

Многопоточное программирование в языке Java, основные понятия. Создание процессов и потоков

Основные понятия. Процесс. Поток. Организация потоков. Создание нового потока. Создание многопоточных приложений.

Тема 3.9

Управление потоками при многопоточном программировании в языке Java. Синхронизация

Способы синхронизации потоков в языке Java. Управление потоками: приостановка, возобновление выполнения, время жизни потоков. Синхронизация и блокировки. Мониторы и условия. Пулы потоков и очередь действий.

Тема 3.10

Построение кроссплатформенных графических интерфейсов. Библиотека Swing

Описание возможностей библиотеки Swing при создании графических интерфейсов. Окно JFrame. Панель содержимого. Менеджеры размещения компонентов. Ручное размещение элементов.

Тема 3.11

Введение в лямбда-выражения

Понятие лямбда-выражений, области применения, примеры использования.