Оглавление

[1.2 Первый контакт — Hello World 1](#_Toc51594436)

[1.2.4 Подробнее о виртуальной машине 1](#_Toc51594437)

[1.2.5 О производительности 2](#_Toc51594438)

[1.3 Знакомство со средой разработки 4](#_Toc51594439)

[1.3.1 Выбираем среду разработки 4](#_Toc51594440)

[1.3.2 Повторяем опыт с Hello World 4](#_Toc51594441)

[1.3.3 Про автоматическую сборку проектов 10](#_Toc51594442)

[2 Базовый синтаксис Java 13](#_Toc51594443)

[2.1 Примитивные типы 13](#_Toc51594444)

[2.1.1 О чем этот модуль 13](#_Toc51594445)

[2.1.2 Примитивные и ссылочные типы 13](#_Toc51594446)

[2.1.3 Тип boolean 16](#_Toc51594447)

[2.1.4 Целочисленные типы 18](#_Toc51594448)

[2.1.5 Тип char 23](#_Toc51594449)

[2.1.6 Вещественные типы 25](#_Toc51594450)

[2.1.7 Полезные классы - Math, BigInteger, BigDecimal 29](#_Toc51594451)

[2.2 Преобразование типов 31](#_Toc51594452)

[2.2.1 Преобразования примитивных типов 31](#_Toc51594453)

[2.2.2 Классы-обертки и связанные преобразования 36](#_Toc51594454)

[2.2.3 Преобразования в строку и обратно 39](#_Toc51594455)

[2.2.4 Полезный совет 40](#_Toc51594456)

[2.3 Массивы и строки 40](#_Toc51594457)

[2.3.1 Про выделение и освобождение памяти 41](#_Toc51594458)

[2.3.2 Массивы 43](#_Toc51594459)

[2.3.3 Varargs 46](#_Toc51594460)

[2.3.4 Сравнение массивов 47](#_Toc51594461)

[2.3.5 Печать массивов 48](#_Toc51594462)

[2.3.6 Строки 49](#_Toc51594463)

[2.4 Управляющие конструкции условные операторы и циклы 52](#_Toc51594464)

[2.4.1 Условные операторы 52](#_Toc51594465)

[2.4.2 Циклы 55](#_Toc51594466)

[2.4.3 Операторы break и continue 57](#_Toc51594467)

[2.4.4 Оператор return 58](#_Toc51594468)

[3 Объекты, классы и пакеты в Java 60](#_Toc51594469)

[3.1 Основы ООП 60](#_Toc51594470)

[3.1.1 О чем этот модуль 60](#_Toc51594471)

[3.1.2 Эволюция подходов к разработке ПО 60](#_Toc51594472)

[3.1.3 Что такое объект 63](#_Toc51594473)

[3.1.4 Инкапсуляция 64](#_Toc51594474)

[3.1.5 Наследование и полиморфизм 66](#_Toc51594475)

[3.1.6 Пример использования ООП 67](#_Toc51594476)

[3.1.7 Так что насчет Java 69](#_Toc51594477)

[3.2 Пакеты и модификаторы доступа 70](#_Toc51594478)

[3.2.1 Ключевое слово package 70](#_Toc51594479)

[3.2.2 Именование пакетов 73](#_Toc51594480)

[3.2.3 Модификаторы доступа 75](#_Toc51594481)

[3.3 Объявление класса 78](#_Toc51594482)

[3.3.1 Класс 78](#_Toc51594483)

[3.3.2 Поля класса 81](#_Toc51594484)

[3.3.3 Конструкторы 83](#_Toc51594485)

[3.3.4 Методы 86](#_Toc51594486)

[3.3.5 Immutable-классы. Вложенные классы 89](#_Toc51594487)

[3.3.6 Перечисления 93](#_Toc51594488)

[3.3.7 Аннотации 96](#_Toc51594489)

[3.4 Наследование. Класс Object 98](#_Toc51594490)

[3.4.1 Наследование, возможности переопределения 98](#_Toc51594491)

[3.4.2 Конструктор класса‑наследника 99](#_Toc51594492)

[3.4.3 Класс Object. Методы toString(), equals(), hashCode() 99](#_Toc51594493)

[3.3.4 Пример семейство геометрических фигур 99](#_Toc51594494)

[3.4.5 О правильном использовании наследования 99](#_Toc51594495)

[3.5 Абстрактные классы и интерфейсы 99](#_Toc51594496)

[3.5.1 Абстрактный класс 99](#_Toc51594497)

[3.5.2 Интерфейс 99](#_Toc51594498)

[3.5.3 Функциональный интерфейс (а также лямбда‑выражения и ссылки на методы) 99](#_Toc51594499)

[4 Обработка ошибок, исключения, отладка 99](#_Toc51594500)

[4.1 Знакомство с исключениями 99](#_Toc51594501)

[4.1.1 Способы возврата ошибки 99](#_Toc51594502)

[4.1.2 Знакомство с исключениями 99](#_Toc51594503)

[4.1.3 Класс Throwable 99](#_Toc51594504)

[4.1.4 Типы исключений 99](#_Toc51594505)

[4.1.5 Пример собственного типа исключения 99](#_Toc51594506)

[4.2 Обработка исключений. Try-catch 99](#_Toc51594507)

[4.2.1 Ловим исключение 99](#_Toc51594508)

[4.2.2 Обрабатываем исключение 99](#_Toc51594509)

[4.2.3 Рекомендации по использованию исключений 99](#_Toc51594510)

[4.3 Логирование 99](#_Toc51594511)

[4.3.1 Класс Logger 99](#_Toc51594512)

[4.3.2 Вывод сообщений в лог 99](#_Toc51594513)

[4.3.3 Тонкая настройка: Handler, Formatter 99](#_Toc51594514)

[4.3.4 Пример использования 99](#_Toc51594515)

[5 Ввод-вывод, доступ к файловой системе 99](#_Toc51594516)

[5.1 Доступ к файловой системе 99](#_Toc51594517)

[5.1.1 java.io.File работа с путями 99](#_Toc51594518)

[5.1.2 java.io.File модифицирующие операции 99](#_Toc51594519)

[5.1.3 java.nio.file.Path новый API 99](#_Toc51594520)

[5.2 Потоки байт 99](#_Toc51594521)

[5.2.2 Конкретные наследники 99](#_Toc51594522)

[5.2.3 Потоки-обертки 99](#_Toc51594523)

[5.3 Потоки символов 99](#_Toc51594524)

[5.3.1 Абстрактные Reader и Writer 99](#_Toc51594525)

[5.3.2 Конкретные наследники 99](#_Toc51594526)

[5.3.3 Классы-обертки 99](#_Toc51594527)

[5.3.4 Форматированный вывод 99](#_Toc51594528)

[5.3.5 Форматированный ввод 99](#_Toc51594529)

[5.3.6 Стандартные потоки 99](#_Toc51594530)

[5.4 Продвинутые возможности 99](#_Toc51594531)

[5.4.1 Запуск внешних процессов 99](#_Toc51594532)

[5.4.2 Классы java.nio 99](#_Toc51594533)

[5.4.3 Сериализация 99](#_Toc51594534)

[6.2 Коллекции 99](#_Toc51594535)

[6.2.1 Зачем нужны коллекции 99](#_Toc51594536)

[6.2.2 Базовый интерфейс java.util.Collection 99](#_Toc51594537)

[6.2.3 Обход коллекции, java.util.Iterator 99](#_Toc51594538)

[6.2.4 Список, java.util.List 99](#_Toc51594539)

[6.2.5 Очередь, java.util.Queue 99](#_Toc51594540)

[6.2.6 Множество, java.util.Set 99](#_Toc51594541)

[6.2.7 Отображение, java.util.Map 99](#_Toc51594542)

[6.2.8 Преобразование в массив и обратно 99](#_Toc51594543)

[6.3 Функциональные интерфейсы 99](#_Toc51594544)

[6.3.1 Зачем нужны функциональные интерфейсы? 99](#_Toc51594545)

[6.3.2 Стандартные функциональные интерфейсы java.util.function 99](#_Toc51594546)

[6.3.3 Инстанцирование функциональных интерфейсов 99](#_Toc51594547)

[6.3.4 Дополнительные возможности 99](#_Toc51594548)

[6.4.2 Пример и общая схема 99](#_Toc51594549)

[6.4.5 Терминальные операции 99](#_Toc51594550)

[6.4.6 Два примера 99](#_Toc51594551)

1.2 Первый контакт — Hello World

1.2.4 Подробнее о виртуальной машине

Поговорим немного подробнее о виртуальной машине. Виртуальная машина Java, поставляемая Oracle в составе JDK или JRE имеет собственное имя HotSpot. Увидеть это имя мы можем, например, в выводе команды: java -version. Существуют и другие реализации виртуальных машин Java, в том числе от других производителей, но они используются реже и в основном в каких-то специфических случаях. В рамках данного курса мы будем иметь дело только с HotSpot. Это наша основная рабочая лошадка. Задача виртуальной машины исполнять java байт‑код. Вопрос: откуда виртуальная машина знает, как именно ей нужно понимать байт-код и исполнять его? Для этого Oracle с каждом релизом java выпускает объемный документ под названием The Java Virtual Machine Specification. В этом документе описан формат класс-файлов, перечислены все возможные в байт-коде инструкции, и подробно описано как именно виртуальная машина должна их исполнять. Именно благодаря такой спецификации возможно существование независимых, но при этом совместимых между собой реализации виртуальной машины Java от разных производителей. А также альтернативных языков программирования для виртуальной машины. Представьте себе, что можно придумать свой собственный похожий или не очень похожий на Java язык программирования, написать для него компилятор в java байт-код. И автоматически получить все преимущества платформы Java, такие как переносимость скомпилированной программы между аппаратными платформами и операционными системами, безопасность, производительность и тому подобное. На самом деле таких языков программирования уже существует не мало. Я назову лишь несколько: Clojure, Groovy, Scala. А также на JVM были импортированы некоторые исходно самостоятельные языки программирования: JRuby (реализация языка программирования Ruby) и Jython (реализация языка программирования Pyton).

1.2.5 О производительности

Как JVM исполняет байт-код? В простейшем случае она будет идти по байт‑коду, последовательно анализировать каждую инструкцию, исполнять ее и переходить к следующей. Этот режим называется интерпретацией. Интерпретатор довольно просто написать, но, к сожалению, даже хорошо написанный интерпретатор будет работать медленно. Если мы сравним один и тот же алгоритм, записанный в машинных кодах процессора и в байт-коде для интерпретатора, то по скорости исполнения интерпретатор проиграет наивному коду в 10-20 раз. С этим ничего не поделать. Первая версия Java 1.0 поддерживала только режим интерпретации поэтому работала медленно. К счастью, довольно скоро уже в версии 1.1 в виртуальную машину Java был добавлен режим Just-in-time compilation или просто JIT. В этом режиме виртуальная машина сначала компилирует байт-код в машинный код, который затем уже непосредственно исполняется процессором. За прошедшие годы JID компилятор был значительно усовершенствован и оптимизирован и теперь он способен на такие сложные вещи как, например, адаптивная компиляция и оптимизация байт‑кода в зависимости от того сколько раз и с какими параметрами исполнился тот или иной участок кода. В результате, Java программа, запущенная в современной виртуальной машине, будет работать уже не в 10-20 раз, а всего в 1,5-2 раза медленнее, чем аналогичная нативная программа. А в некоторых случаях Java‑программы может оказаться даже быстрее. Это возможно благодаря тому, что HotSpot в своих оптимизациях использует знания о фактическом выполнении программы. В отличии от традиционных компиляторов, которые могут видеть только статические исходные тексты.

1.3 Знакомство со средой разработки

1.3.1 Выбираем среду разработки

Теперь, когда вы знаете, как на самом деле компилируются и запускаются Java‑программы, вы можете начать пользоваться интегрированной средой разработки, которая скроит все эти утомительные подробности за красивыми кнопочками своего интерфейса. Существует 3 основных среды для разработки на Java. Это IntelliJ IDEA, NetBeans, Eclipse. Вы можете выбрать любую на свой вкус. Если определится не можете, то рекомендую выбрать IntelliJ. Я много пользовался каждой из этих сред разработки и в итоге остановился именно на ней как на самой удобной.

1.3.2 Повторяем опыт с Hello World

Предлагаю снова написать и запустить простейшую программу HelloWorld. Но на этот раз пользуясь средой разработки. Я буду показывать IntelliJ IDEA. Но то же самое вы можете повторить и в любой выбранной вами среде разработки. То, что я покажу, не заменит вам полноценного туториал по среде разработки, но надеюсь, что за 5 минут я смогу показать достаточно, чтобы вы начали пользоваться этим инструментом и получили первые бонусы к своей продуктивности.

Чтобы приступить к работе нам нужно создать проект. Все перечисленные мною среды разработки оперируют понятием проекта. Проект — это набор файлов с исходниками, которые все вместе будут компилироваться плюс настройки, такие как используемые JDK, необходимые для сборки библиотеки и так далее. Мы создадим простой Java‑проект. При первом запуске IDEA вам, возможно, потребуется настроить путь к JDK. Проект назовем HelloWorld. Создадим в проекте наш главный класс. Добавим в него метод main, точку входа в программу. Сигнатура метода main довольно длинная. Чтобы не ошибиться можно попросить среду разработки немного нам помочь — psvm. Аналогично с инструкцией для печати текста в консоль — sout. Осталось дописать только текст, который мы будем выводить. IDEA имеет очень много удобных функций для ускорения написания кода. И то, что я показал, это только верхушка айсберга. Среда разработки — это не только удобный текстовый редактор. Она анализирует код, который вы пишите и в любой момент может оказать вам поддержку. В любом месте программы можно попросить среду разработки подсказать, что написать дальше. Например, посмотрим какие существуют разновидности метода для печати текста в консоль — Ctrl + Space. Еще одна очень полезная функция среды разработки — это переход к объявлению класса, метода или любого другого идентификатора, который вы видите в программе — Ctrl + ЛК. Также среда разработки с удовольствием подскажет вам какие параметры принимает данный метод — Ctrl + P. Или даже покажет вам документацию на этот метод — Ctrl + Q. Любой идентификатор в программе можно переименовать, а среда разработки позаботиться об обновлении ссылок на него во всех метах кода. Если мы переименуем класс, то среда разработки также переименует также и исходный файл — Shift + F6. Небольшое отступление про документацию. В JDK есть стандартный инструмент для генерации документации по исходному коду. Он называется Javadoc. На выходе получается html-страничка, на которой перечислены все классы, методы, поля. Описание классам и методам, которые попадают в документацию, берутся из комментариев специального вида из исходного кода. Вообще в Java есть 3 вида комментариев. Есть обычные однострочные комментарии:

// ...

Есть многострочные комментарии:

/\*

...

\*/

И есть те самые специальные комментарии, содержащие документацию. Они выглядят почти как многострочные комментарии:

/\*\*

\* ...

\*/

Классы стандартной библиотеки, а также многих других сторонних библиотек снабжены такими Javadoc комментариями. А среды разработки умеют их удобно показывать — Ctrl + Q. Для примера мы можем добавить Javadoc к нашему классу и его главному методу. В Javadoc можно использовать обычную html-разметку и некоторые специальные теги, которые подскажет среда разработки.

Хочу показать еще одну полезную фишку среды разработки. Она умеет автоматически форматировать код для соответственно принятому стилю кодирования. Когда вы имеете дела с неряшливо написанным кодом, то очень полезно сначала его по‑человечески отформатировать, а уж потом читать и тем более править. Если вам от какого-то злодея досталась программа, в которой совершенно нет отступов, то легким движением руки ее можно привести в нормальный вид Ctrl + Alt + L. (удаление отступа: Shift + Tab). Стиль кода, к которому все автоматически приводится задается в настройках. При желании его можно изменить. Но есть некоторые правила, которые придерживаются практически все Java‑программисты. Эти правила берут начало с документа Code Conventions for the Java Programing Language, который Sun Microsystems выпустил вместе с одной из ранних версий Java. Например, для отступов используется 4 пробела. А открывающая фигурная скобочка не переносится на новую строку. Также в Java есть общепринятые правила наименования. Классы имеются заглавными буквами. Если название класса состоит из нескольких слов, то они записываются слитно, и каждая начинается с заглавной буквы. Переменные и методы принято именовать со строчной буквы. Если название состоит из нескольких букв, то второе и последующее начинается с заглавной буквы. Автоформатирование кода имена не трогает. Для приведения имен к принятому стилю придется переименовать каждый элемент программы отдельно. Но с помощью среды разработки это совсем не трудно — Shift + F6. Теперь, когда программа написана, ее можно запустить. Делается это очень просто: выбирается главный класс программы и пункт меню Run — Shift + Ctrl + F10. После запуска автоматически создается конфигурация запуска. Там можно указать разные настройки. Например, передаваемые программе аргументы командной строки. В одном проекте может быть много конфигураций запуска. В среде разработки появляется возможность не просто запускать программу и наблюдать ее вывод, а еще и отлаживать ее, то есть останавливать и изучать ее внутреннее состояние во время исполнения. Например, мы можем убедиться, что в метод main действительно предаются параметры командной строки.

1.3.3 Про автоматическую сборку проектов

Среда разработки хороша для написания Java-кода и его отладки. Однако программистам приходится решать и задачи другого сорта. Я говорю о сборки финальной программы их исходников. В простейшем случае сборка программы представляет собой просто вызов компилятора. Однако в реальных программах сборка состоит зачастую из большого числа нетривиальных шагов таких как скачивание необходимых библиотек, генерации части исходников из каких-нибудь мета данных, собственно компиляция, прогон тестов, упаковка собранных классов в JAR архив, генерация документации и так далее. Такие сложные процессы обязательно нужно автоматизировать. Для этого можно было бы написать bat-ник или shell-скрипт. Однако такой скрипт будет очень сложен и более того он не сможет работать, например, на другой операционной системе. Поэтому появились и стали широко использоваться специализированные инструменты для сборки Java-программ.

На данный момент есть три популярных системы сборки Java-программ: Ant, Gradle, Maven. Все они являются кроссплатформенными благодаря тому, что написаны на Java. Подробное знакомство с этими инструментами выходит далеко за рамки нашего базового курса. Я лишь хочу сказать, что инструменты такие существуют, что вы с ними обязательно столкнетесь и что боятся их не нужно. У каждого инструмента есть характерное имя файла, которым описывается сборка Java-программ. Для Ant — build.xml, для Gradle — build.gradle, для Maven — pom.xml. Так что если вы где-нибудь, например, на GitHub, найдете интересную Java-программку, то по одному из этих файлов, лежащим рядом с исходниками вы легко определите как именно эту Java‑программу нужно собирать. Кроме того, если для сборки используется Gradle или Maven, то вы можете напрямую импортировать такую Java-программу в среду разработки, а она по файлу build.gradle или pom.xml автоматически определит директории, где лежат исходники, скачает и подключит нужные библиотеки и выставит прочие настойки. Так что вы практически моментально получите успешно компилирующийся и готовый к разработке проект.

2 Базовый синтаксис Java

2.1 Примитивные типы

2.1.7 Полезные классы - Math, BigInteger, BigDecimal

2.2 Преобразование типов

2.2.1 Преобразования примитивных типов

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9. | double doubleValue = 1d + 1f;  float = floatValue = 1f \* 1;  long = longValue = 1L – '0';  byte a = 1;  byte b = 2;  byte c = (byte) (a + b); |

|  |  |
| --- | --- |
| 1.  2.  3. | byte a = 1;  a += 3;  // a = (byte) (a + 3); |

Итак, если к байт прибавить int, получится int? Тогда как же работает (2). Ведь мы получается записываем int в byte. На самом деле эта сокращенная запись эквивалентна выражению (3), в которое включено обратное преобразование из int в byte. Теперь, зная все эти подводные камни, вы уже не должные удивляться следующему примеру.

|  |  |
| --- | --- |
| 1.  2.  3. | byte b = -1;  b >>>= 7;  // b = (byte) (b >>> 7) |

Берем byte, в котором храниться -1, то есть в двоичном представлении все 8 бит — 11111111. Делаем логический сдвиг вправо на 7 позиций. Что у нас получится? Интуиция подсказывает, что во 2-ом представлении должны остаться 00000001. Однако, как бы не так. Вспомним, что перед любой операцией над байтом происходит автоматическое расширение до int, то есть в данном случае получается 32 2‑ичных единицы. Логический сдвиг вправо на 7 позиций применяется к этому int, а потом выполняется обратное приведение к byte, то есть берутся младшие 8 бит все единицы. Окончательное значение переменной b — -1, то есть в результате сдвига значение не изменилось.

Учитывая все те ужасы, которые я вам рассказал и показал могу дать рекомендацию: не использовать типы byte и short без необходимости. А тип char использовать только для хранения символов. Локальные переменные такие как счетчики цикла и прочие аккумуляторы лучше сразу делать типа int чтобы не наступить на грабли. А задумываться об использовании менее емких типов стоит только в том случае, когда это даст существенную экономию памяти. Например, когда у вас в программе есть большой массив чисел.

2.2.2 Классы-обертки и связанные преобразования

|  |  |
| --- | --- |
| boolean  byte  short  int  long  char  float  double | Boolean  Byte  Short  Integer  Long  Character  Float  Double |

Мы будем плавно закругляться с примитивными типами и переходить к ссылочным. Но есть еще несколько моментов различных преобразований на грани между ними. Первый тип преобразования связан с так называемыми классами обертками. Для каждого примитивного типа в стандартной библиотеке есть брат близнец — ссылочный тип. Класс с таким же названием, но начинающийся с заглавной буквы. Есть только два небольших отличия в написании: Integer вместо int и Character вместо char. Эти классы называют классами обертками, потому что внутри они содержат единственное поле value соответствующего примитивного типа.

Мы можем взять примитивный int и завернуть его в объект Integer. Если нарисовать что при этом происходит в памяти, то получится следующая картинка. Примитивный int — это просто 4 байта памяти, в которых лежит значение. Ссылочный Integer — это ссылка на объект Integer, лежащий где-то в памяти, а внутри него лежит примитивное значение int.

Классы обертки полезны в нескольких случаях. Например, для того чтобы хранить числа и символы в коллекциях. Стандартные коллекции в Java не поддерживают примитивные типы, поэтому приходиться превращать их в объекты. Во-вторых, классы обертки могут пригодиться, когда вам нужно выразить в программе факт отсутствия значения. Для этого вместо переменной примитивного типа вы заводите переменную соответствующего ссылочного типа и храните в ней либо null, либо ссылку на значение.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6. | int primitive = 0;  Integer reference =  Integer.valueOf(primitive);  int backToPrimitive = reference.intValue(); |

Для упаковки примитивов в объекты в каждом классе обертки есть метод valueOf(). Такую упаковку в мире Java принято называть boxing. А получить из объекта обертки обратно в примитивное значение можно при помощи метода intValue() или другого аналогичного метода, имя тут зависит от конкретного типа. Это называется unboxing.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6. | Integer a = 1;  int b = a;  Integer c = 10;  Integer d = 10;  Integer e = c + d; |

Также в Java поддерживается неявное преобразование. То есть можно использовать объекты обертки там, где ожидаются примитивные значения и наоборот. Вызовы методов преобразования компилятор добавит сам. Как видите над обертками можно даже выполнять операции (6). На вид это не отличается от операции над примитивами, однако на самом деле там происходит довольно сложная работа. Сначала из оберток извлекаются примитивные значения, над ними выполняются операции. Результат опять упаковывается в новую обертку. Это на порядок медленнее, чем аналогичная операция с переменными примитивных типов. Поэтому к использованию классов оберток надо прибегать только при необходимости.

2.2.3 Преобразования в строку и обратно

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | long fromString = Long.parseLong("12345");  String fromLong = Long.toString(12345);  String concatenation = "area" + 51; |

И 2-ой тип преобразований между примитивными и ссылочными типами — это конвертация примитивов в строку и обратно. Для этих преобразований в каждом классе есть пара статических методов: parseИмяТипа и toString. Причем для чисел есть варианты, принимающий второй параметр основания системы исчисления. Поддерживаются основания до 36 включительно. Конвертация строки в примитив возможно только явно при помощи метода parse, а вот примитив в строку может превратиться автоматически, если мы конкатенируем его со строкой.

2.2.4 Полезный совет

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12. | short maxShortValue = Short.MAX\_VALUE;  int bitCount = Integer.bitCount(123);  boolean isLetter = Character.isLetter('a');  float floatInfinity =  Float.POSITIVE\_INFINITY;  double doubleNaN = Double.NaN;  boolean isNaN = Double.isNaN(doubleNaN); |

В заключение разговора о классах‑обертках порекомендую вам на досуге заглянуть в их Javadoc. Или сразу в исходники. Со средой разработки это совсем не сложно. Там вы найдете много полезных методов и констант, которые могут когда‑нибудь пригодиться. Например, минимальное или максимальное значение, представимое каждым типом. Для целочисленных типов различные битовые манипуляции. Для символов — определение букв и цифр (5), преобразование регистров. Для вещественных типов — вещи, связанные с бесконечностями и NaN. Например, метод isNaN, который нужен потому, что обычном оператором сравнения равно‑равно NaN не поймать

2.3 Массивы и строки

2.3.1 Про выделение и освобождение памяти

Все, о примитивах мы с вами поговорили достаточно. Теперь окончательно и бесповоротно перехожим к изучению ссылочных типов. В данном модуле мы с вами рассмотрим массивы и строки. Это типы, без которых не обходится практически не одна программа. Но для начала немного теории о любых ссылочных типах и управлению памятью.

|  |  |
| --- | --- |
| 1.  2.  3. | BigInteger number;  Number = new BigInteger("12345"); |

Объявление переменной и выделение памяти под объект — это два независимых действия. Если просто объявить в программе переменную, например типа BigInteger, то никакой объект в этот момент создан еще не будет. Мы просто объявляем ячейку памяти, которая может содержать в будущем ссылку на объект. Объект создается оператором new. Помимо выделения мета в памяти оператор new вызывает конструктор. Это специальный метод, инициализирующий внутреннее состояние объекта в соответствии с переданными параметрами. Например, у класса BigInteger есть конструктор, принимающий строку. И он порождает объект, представляющий соответствующее число.

Оператора delite или другого, явно высвобождающего память, в Java нет. Помните, когда я говорил об истории языка Java то упоминал, что одной из целью создания Java было помочь программисту написать корректные безопасные программы. Известно, что явное управление памятью, как, например, в C и C++, это причина большого количества ошибок. Поэтому в Java используется автоматическое управление памятью. Программист выделяет память, а за ее освобождение отвечает виртуальная машина. При этом JVM следит за тем какие объекты в памяти доступны в вашей программе, то есть на какие объекты в памяти есть прямые или косвенные ссылки из ваших переменных. Когда ваша программа больше не ссылается на объект, то он может быть удален виртуальной машиной, память освобождена и переиспользована под новые объекты. Для современных сборщиков мусора проблем даже не представляют даже циклические ссылки, то есть ситуации, когда есть набор объектов, ссылающихся друг на друга. Если эти циклы не достижимы из вашей программы, то все такие объекты будут благополучно удалены. Несмотря на автоматическую и очень умную сборку мусора утечки памяти в Java‑программах все‑таки бывают. Но связаны они только с тем, что по недосмотру программиста программа продолжает держать ссылки на объекты, которые ей больше не нужны.

2.3.2 Массивы

|  |  |
| --- | --- |
| 1.  2.  3. | int[] numbers;  String[] args;  boolean bits[]; |

Теперь, после теоретического отступления начнем знакомства с массивами. Объявление массива отличается от объявления простой переменной наличием пары скобок. Компилятор позволяет ставить скобки как после имени типа, так и после имени переменной. Но рекомендуется ставить их у имени типа, чтобы четко отделить тип от имени переменной.

|  |  |
| --- | --- |
| 1.  2.  3. | int[] numbers = new int[100];  String[] args = new String[1];  boolean bits[] = new boolean[0]; |

Объявление переменной и выделение памяти под массив — это отдельные действия. Массив создается оператором new. Размер массива фиксируется в момент создания и в последствии не может меняться. При создании массива все элементы автоматически инициализируются значениями по умолчанию. Для чисел и символов — это нули, для булевских значений — это false. Для ссылок — это null. Явно инициализировать свежесозданный массив с нулями не требуется.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | int[] numbers = new int[] {1, 2, 3, 4, 5};  boolean bits[] = new boolean[] {true, false};  // this works only in variable declaration  char[] digits = {  '0', '1', '2', '3', '4',  '5', '6', '7', '8', '9'}; |

Если вам нужен массив с какими‑то не нулевыми значениями, можно перечислить значения всех элементов при создании. Тогда количество элементов в квадратных скобках указывать не нужно. Размер массива будет определяться тем, что вы напишите в фигурных скобках. В некоторых случаях new, имя типа и квадратные скобки можно опустить, оставив только фигурные скобки с содержимым массива. Это работает только когда переменная объявляется и инициализируется в одной строчке. В остальных случаях надо использовать полную запись с оператором new.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9. | int[] numbers = {1, 2, 3, 4, 5};  int arrayLength = numbers.length;  int firstNumber = number[0];  int lastNumber = number[arrayLength – 1];  int indexOutOfBounds = numbers[5]; |

Длина созданного массива доступна через поле length(). Обращение к элементу массива выполняется при помощи квадратных скобок и индекса элемента. Индексация элемента всегда начинается с 0. Таким образом индексы массива длины n — это числа от 0 до n – 1. Java проверяет, что вы не выходите за допустимые границы индексов. При попытки обратиться к несуществующему элементу массива будет сгенерировано исключение, а программа аварийное завершится.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | int[][] matrix1 = new int[2][2];  int[][] matrix2 = {{1, 2}, {3, 4}}  int[] firstRow = matrix2[0];  int someElement = matrix2[1][1]; |

В Java поддерживаются 2-х и более ‑мерные массивы. Но многомерность здесь не совсем честная. Она эмулируется через вложенные одномерные массивы. Например, двумерный массив — это массив массивов, если нарисовать раскладку всего этого дела по памяти, то получится следующая картинка … Это может быть не очень экономно по памяти, да и обращение глубоко вложенным элементам будут по дороже, чем конструкции, когда многомерный массив хранится в памяти монолитным куском.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | int[][] triangle = {  {1, 2, 3, 4, 5},  {6, 7, 8, 9},  {10, 11, 12},  {13, 14},  {15}};  int secondRowLength = triangle[1].length; |

Зато позволяет иметь ступенчатые массивы, когда строчки отличаются по длине. Или даже некоторые строчи могут отсутствовать.

2.3.3 Varargs

|  |  |
| --- | --- |
| 1.  2.  3. | static int maxArray(int[] numbers) { ... }  static int maxVarargs(int... numbers) { ... } |

Есть один случай, когда для объявления массива вместо квадратных скобок используется многоточие. Это объявление метода, принимающего переменное количество параметров. Давайте посмотрим в чем разница. Если у нас есть метод, вычисляющий максимум элементов массива, то чтобы вызвать его и посчитать максимум из трех элементов, мы должны написать что‑то такое: mA(new int[]{1,2,3}). Довольно неудобно и неуклюже выглядит. Если же метод объявлен с многоточием, то мы можем написать гораздо компактнее: mV(1,2,3). Компилятор сам упакует параметры в массив и передаст в метод. Обычный массив также сюда можно передать. Внутри метода работа с таким массивом ничем не будет отличаться от работы с обычным массивом. Многоточие влияет только на то, как метод можно вызывать.

2.3.4 Сравнение массивов

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10. | int[] a = {1, 2, 3};  int[] b = {4, 5, 6}  boolean equals1 = a == b;  boolean equals2 = a.equals(b);  boolean equals3 = Arrays.equals(a, b);  boolean equals4 = Arrays.deepEqeals(a, b); |

Часто возникающая практическая задача — сравнение 2-х массивов. Общее правило в Java заключается в том, что оператор равно‑равно сравнивает ссылки, то есть ссылаются ли две переменные на один и тот же объект. Это подходит не всегда. Для сравнения объектов по содержимому в Java обычно используется метод equals(). К сожалению, именно для массивов он не работает. Точнее работает, но так же сравнивает ссылки. Для сравнения содержимого есть метод equals в классе Arrays. Не забудьте только импортировать этот класс в начале вашей программы: import java.util.Arrays. Метод Arrays.equals попарно сравнивает элементы 2‑х массивов, но только на первом уровне вложенности. То есть он отлично работает для одномерных массивов, но для 2‑х и более ‑мерных массивов он будет ошибаться. Решение для многомерных массивов — метод deepEqeals(), глубокое сравнение в том же классе Arrays.

2.3.5 Печать массивов

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | int[] a = {1, 2, 3};  System.out.println(a);  System.out.println(Arrays.toString(a));  System.out.println(Arrays.deepToString(a)); |

Другая практическая задача — распечатать содержимое массива. Если просто передать массив в метод println(), то мы увидим непонятную абракадабру, что‑то в таком духе: [I@...] (16-ричные цифры). Как же получить содержимое элемента массива? В этом поможет тот же класс Arrays и содержащиеся в нем методы toString() и deepToString(). Разница между ними такая же, как между equals() и deepEqeals(). Вообще класс Arrays очень полезный. В нем есть методы для сортировки, 2-ичного поиска, копирование всего или части массива. Прежде чем писать свой велосипед по работе с массивами стоит проверить нет ли подходящего метода в этом классе.

2.3.6 Строки

|  |  |
| --- | --- |
| 1.  2.  3. | String hello = "Hello";  String specialChars = "\r\n\t\"\\";  String empty = ""; |

Перейдем к обсуждению строк в Java. Со строками мы уже несколько раз встречались, но сейчас поговорим более обстоятельно. Строка — это ссылочный тип, хранящий последовательность символов, произвольной длины.

|  |  |
| --- | --- |
| 1.  2.  3. | char[] charArray = {'a', 'b', 'c'};  String string = new String(charArray);  char[] charFromString = string.toCharArray(); |

Строки представлены классом String. String — это не то же самое, что массив символов, но их можно легко конвертировать друг в друга. В некоторых языках строки должны заканчиваться нулевым символом. В Java это не так.

|  |  |
| --- | --- |
| 1. | String zeros = "\u0000\u0000"; |

Длина строки хранится в отдельном поле, поэтому внутри строки можно свободно использовать нулевые символы. Они будут ее частью.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9. | String s = "stringIsImmutable";  int length = s.length();  char firstChar = s.charAt(0);  boolean endsWithTable = s.endsWith("table");  boolean containsIs = s.contains("Is"); |

Фундаментальное свойство строк в Java — это их неизменяемость. Когда объект String создан, мы не можем в нем уже поменять ни один символ. Содержимое объекта останется таким навсегда. Мы можем только читать содержимое строки, узнавать длину, получать символы по индексам, проверять содержит ли строка определенный суффикс, префикс или подстроку.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | String s = "stringIsImmutable";  String substring = s.substring(0, 6);  String afterReplace = s.replace("Imm", "M");  String allCapitals = s.toUpperCase(); |

Любое изменение строки связано с созданием нового объекта. Поэтому методы взятия подстроки, поиск и замена, преобразование регистра — все они возвращают новую строку результата.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | String hello = "Hello ";  String world = "world!";  String helloWorld = hello + world;  StringBuilder sb = new StringBuilder();  sb.append(hello);  sb.append(world);  String helloWorld = sb.toString(); |

Так же и конкатенация. При конкатенации создается новая строка, состоящая из символов исходных строк. На самом деле, если декомпилировать класс с таким кодом, то мы увидим, что компилятор заменил нашу конкатенацию на фрагмент кода с использованием StringBuilder. То есть оператор конкатенации — это просто синтаксический сахар, более простая форма записи для использования StringBuilder.

StringBuilder — это объект очень похожий на строку, но допускающий изменения содержимого. Когда у вас в программе есть строка, которую нужно постоянно мутировать: добавлять, удалять, заменять символы или фрагменты, то может быть более эффективно и удобно использовать вместо класса String — класс StringBuilder. Если речь идет просто о замене отдельных символов без изменения длины строки, то вам прекрасно подойдет обычный массив символов.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6. | boolean referenceEquals = s1 == s2;  boolean contentEquals = s1.equals(s2);  boolean concatEqualsIgnoreCase = s1.equalsIgnoreCase(s2) |

Популярная ошибка при использовании строк — это сравнение строк при помощи оператора равно‑равно. Надо запомнить, что в Java он всегда сравнивает ссылки, а не содержимое. Для сравнения содержимого есть метод equals(). И для строк, в отличии от массивов он работает. А так же есть специальный метод для сравнения без учета регистра — equalsIgnoreCase().

2.4 Управляющие конструкции условные операторы и циклы

2.4.1 Условные операторы

Теперь мы хорошо умеем объявлять переменные разных типов и записывать простые выражения такие как арифметические и логические операции и вызовы методов. Но на одних переменных и арифметике далеко не уедешь. В настоящих программах нужны управляющие конструкторы такие как условный оператор и циклы. О них то мы и поговорим. Я рассчитываю, что вы в свой жизни уже программировали на каком‑то языке, поэтому знаете, что такое условный оператор и цикл. Мы с вами быстренько пробежимся по синтаксису данных операторов в Java без каких‑либо философских подробных рассуждений.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | if (weatherIsGood) {  walkInThePark();  } else {  learnJavaOnStepic();  } |

Условный оператор if с опциональной частью else. Условие, по которому осуществляется ветвление, должно быть заключено в круглые скобки. Условие может быть только выражение типа boolean, то есть такая популярная в других языках ошибка, как использование в условии присваивания вместо сравнения будет сразу замечена компилятором. Если под if или else находится всего один оператор, то компилятор разрешает не ставить фигурные скобки. Но рекомендуется использовать скобки даже в этом случае для улучшения читабельности и избежание ошибок в будущем, когда захочется дописать куда‑нибудь вторую строчку.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9. | if (weatherIsGood) {  System.out.println("Weather is good");  } else {  System.out.println("Weather is bad");  }  // same effect, but much shorter  System.out.println("Weather is " +  (weatherIsGood ? "good" : "bad")); |

Иногда удобно вместо полновесного if использовать сокращенную запись — тернарный условный оператор. Он предназначен для включения внутрь других выражений. Результатом вычисления тернарного оператора является значение либо первого, либо второго подвыражений в зависимости от условия. Чтобы компилятор сообразил какой тип значения возвращает тернарный оператор, оба его подвыражения должны либо иметь один и тот же тип, либо приводится к общему типу.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13. | switch (digit) {  case 0:  text = "zero";  break;  case 1:  text = "one";  break;  // ...  default:  text = "???";  } |

Иногда ветвление нужно делать по более богатому типу, чем boolean. Для этого есть оператор switch. Он поддерживает ветвление по значениям типа char, String, E-NUM (перечисление) и по целочисленным типам, кроме long. Выбирается та ветка case, которая соответствует значению, переданному в условии. Если совпадение не найдено, то выполняется ветка default. Если не написать break, то исполнение перейдет на следующий кейс и так далее либо до первого breake, либо до конца оператора switch.

2.4.2 Циклы

|  |  |
| --- | --- |
| 1.  2.  3. | while (haveTime() && haveMoney()) {  goShopping();  } |

В Java есть несколько видов циклов. Первым рассмотрим цикл с предусловием. Он имеет следующий синтаксис: while, дальше в круглых скобках булевское условие и далее в фигурных скобках повторяющееся действие. Фигурные скобки в случае единственного оператора внутри цикла можно опустить, но рекомендуется этого не делать из тех же соображений, что и с условным оператором.

|  |  |
| --- | --- |
| 1.  2.  3. | do {  goShopping();  } while (haveTime() && haveMoney()) |

Несколько менее популярен цикл с пост‑условием do-while. В нем 1‑ая итерация совершается безусловно. Условие проверяется только после нее. Еще одна особенность данного цикла — после него обязательно точка запятой.

|  |  |
| --- | --- |
| 1.  2.  3. | for (int i = 0; i < args.length; i++) {  System.out.println(args[i]);  } |

А вот цикл for используется очень часто. В его заголовке есть 3 части. Первая — инициализация. Здесь, например, можно объявить переменную счетчик и присвоить ей начальное значение. Вторая часть — это условие начала итерации. Она проверяется так же и перед первой итерацией. И третья часть — это выражение, выполняемое после каждой итерации. Каждая из 3-х частей опциональна. Обязательные только разделяющие их точки запятой. В пределе, опустив все 3 части, мы получим бесконечный цикл.

|  |  |
| --- | --- |
| 1.  2.  3. | for (String arg : args) {  System.out.println(arg);  } |

Самый характерный случай использования цикла for — это обход всех элементов массива или коллекции. Поэтому в Java появился упрощенный синтаксис именно для этого случая. Здесь не нужно думать об индексах и соответственно меньше возможностей напортачить.

2.4.3 Операторы break и continue

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | boolean found = false;  for (String element : haystack) {  if (needle.equals(element)) {  found = true;  break;  }  } |

Рука об руку с циклами идут операторы break и continue. Оператор break прерывает цикл и передает управление на следующую за циклом строку. Он применим ко всем видам циклов.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | int count = 0;  for (String element : haystack) {  if (!needle.equals(element)) {  continue;  }  count++;  } |

Оператор continue досрочно завершает текущую итерацию и начинает следующую. Разумеется, перед новой итерацией проверяется условие цикла. Это так же применимо ко всем видам циклов.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | boolean found = false;  outer:  for (int[] row : matrix) {  for (int x : row) {  if (x > 100) {  found = true;  break outer;  }  }  } |

Когда у вас в программе несколько вложенных циклов, то операторы break и continue по умолчанию действуют на самый глубоко вложенный цикл. С использованием меток можно указать оператором конкретный цикл, на который они будут действовать. Например, самый внешний. Метки в Java существуют только для операторов break и continue. Никакого goto, к счастью, нет. Поэтому любители спагетти‑кода просим поискать другой язык программирования.

2.4.4 Оператор return

Последний оператор, который мы рассмотрим в этом модуле, — это оператор return, возврат значения из метода. Мы пока создавали только метод main(), объявленный как void, то есть ничего не возвращающий. Поэтому оператор return нам пока не был нужен.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | private static String getGreeting(String  name) {  if (name == null) {  return "Hello anonymous!";  } else {  return "Hello " + name + "!";  }  } |

Но со временем, с ростом размера программы мы будем выносить какую‑то повторно используемую или просто слишком многословную логику в отдельные именованные методы, объявляя им входные параметры и тип результата. Если метод должен что‑то возвращать, то компилятор проверит, что значение возвращается при любых условиях. Например, если в одной из веток условного оператора забыть return, то программа не скомпилируется.

Оператор return без аргумента можно использовать в void методе. Это будет означать немедленный выход из метода.

3 Объекты, классы и пакеты в Java

3.1 Основы ООП

3.1.1 О чем этот модуль

Java — это объектно‑ориентированный язык программирования. Он изначально спроектирован и заточен под написание программ в объектно‑ориентированном стиле. И хотя на Java можно, как мы с вами до сих пор и делали, писать программы в обычном процедурном стиле, где все методы статические, а объектов нет или почти нет, на самом деле Java способна на большее. В этом модуле мы с вами будем учиться правильно готовить Java, писать программы в объектно‑ориентированном стиле, раскрывая потенциал языка и получая от него максимум отдачи.

3.1.2 Эволюция подходов к разработке ПО

Для начала обсудим, что из себя вообще представляет объектно‑ориентированное программирование или сокращенно ООП, и чем оно выгодно отличается от процедурного? Мы не будем сильно углубляться в теорию, потому что здесь легко закопаться. Про ООП умными людьми написаны целые фолианты. Например, фундаментальный труд Бертрана Майера «Object‑Oriented Software Construction» на 1200 станиц. Я постараюсь давать простые определения и, по возможности, сводить все к практике.

Движущей силой эволюцией подхода к разработке программного обеспечения всегда являлось стремление писать более сложные программы меньшими усилиями. Когда‑то давно программы писались в машинных кодах. Это было страшно сложно и неудобно. Поэтому со временем появился язык ассемблера, более‑менее человеко-читаемая альтернатива машинным кодам.

Далее появились языки высокого уровня и структурное программирование, где программист вместо регистров процессора использовал переменные, вместо машинных инструкций — удобные арифметические логические операторы и выражения, составленные из них, вместо спагетти из goto — структурированные и условные операторы, циклы и вызовы подпрограмм.

Следующим этапом эволюции стало появление объектно-ориентированного программирования. ООП — это не единственное возможное направление эволюции. Например, много интересного сейчас происходит в области функционального программирования. И даже в Java в 8 версии появились некоторые элементы функционального стиля.

ООП также не является серебряной пулей, то есть его использование не гарантирует автоматически успеха разработки какой‑то программной системы. Однако, тем не менее, ООП является широко распространенным и популярным подходом к разработке программного обеспечения. Он применим к широкому кругу задач и при правильном использовании действительно помогает писать более простые, понятные и поддерживаемые программы.

ООП — это парадигма программирования, в которой программа состоит из объектов, взаимодействующих друг с другом посредствам обмена сообщениями. При этом объект, согласно Гради Бучу, признанному мировому авторитету и эксперту по ООП, это мыслимая или реальная сущность, обладающая характерным поведением и отличительными характеристиками и являющаяся важной в предметной области.

Данное определение очень абстрактно и попахивает философией. Поэтому попробуем наполнить его практическим смыслом и описать более конкретно, что же такое объект.

3.1.3 Что такое объект

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | Client client = new Client();  Contract contract = new Contract();  Order order = new Order(); |

Каждый объект относится к какому‑либо типу. Или еще говорят, что объект является экземпляром класса. Классы часто соответствуют сущностям предметной области, в контексте которой работает наша программа. Например, практически в каждой корпоративной системе мы найдем такие классы как клиент, контракт, заказ. Также классы могут играть вспомогательную роль и не соответствовать сущностям реального мира. Например, класс StringBuilder помогает собирать строки, класс BigInteger реализует длинную арифметику. Заводя в программе классы, мы на самом деле описываем предметную область и в каком‑то смысле формируем тот язык, на котором наша программа будет написана.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9. | Client client = new Client();  client.setName("Vasily Poupkin");  client.setBirthDate("1990-06-13");  client.setAddress("Uryupinsk");  Order[] orders = client.getOrders(); |

Класс — это общее описание, схема того, как устроена сущность. Например, у клиента должно быть имя, дата рождения, домашний адрес. А объект — это конкретный экземпляр класса с конкретным состоянием. Например, клиент по имени Василий Пупкин, 1990 года рождения, проживающего в Урюпинске. В программе может быть много клиентов со своими именами и адресами, но все они будут устроены одинаково, так как описано в классе. Объект может принимать сообщения. В большинстве языков, в том числе и в Java, отправка сообщение объекту реализована как вызов метода этого объекта. Набор поддерживаемых сообщений описан в классе и соответствует тому, что нам требуется от данной сущности. Например, клиент может переехать, и нам придется обновить его адрес. Значит в классе Client будет метод объявления адреса. Или мы захотим получить по клиенту список его активных заказов. Такой метод в классе тоже можно завести.

3.1.4 Инкапсуляция

Объект — это умные данные. То есть это не просто запись с набором полей, которые программа напрямую читает и пишет. Объект сам контролирует свое внутреннее состояние и обеспечивает его корректность. Взаимодействие между объектами в программе строится на основе тех методов, того внешнего контракта, который описан в классе. Методы обычно, как и классы, выделяются на основе анализа предметной области и соответствуют тому, что с объектом можно делать в реальном мире.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | Order order = getOrder();  order.paymantReceived();  order.sendToDelivery();  order.cancel(); |

Например, заказ можно оплатить, можно передать в доставку, можно отменить. Вызывающей стороне все равно как реализован тот или иной вызываемый ее метод объекта. Вызвав метод, она понятным для объекта способом выразила свое намерение, а дальше все в руках объекта. Он сам решает, что и как нужно сделать, чтобы удовлетворить потребность вызывающей стороны. Например, передача заказа в доставку может на самом деле означать следующее: это печать сопроводительной документации, отправка уведомления в службу доставки, обновление данных в системе складского учета, отправка SMS клиенту и так далее. Все эти подробности от вызывающей стороны скрыты. Такое сокрытие реализации внутри класса в мире ООП принято называть инкапсуляцией.

3.1.5 Наследование и полиморфизм

В ООП предусмотрена возможность создавать новые классы, наследуясь от существующих. Унаследованный класс автоматически получает все признаки базового класса и по желанию может к ним что‑то добавить или переопределить. В частности, можно переопределить обработку метода, объявленного в базовом классе.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | Client client = new VIPClient();  client.getDeliveryPrice();  // actually VIPClient.getDeliveryPrice()  // is executed |

Например, мы можем создать класс VIPClient, унаследованного от Client и переопределить в нем метод, вычисляющий стоимость доставки. ВИП‑клиентам доступна бесплатная доставка товара беспилотным квадрокоптером в день заказа. При этом код, подсчитывающий стоимость заказа с учетом доставки, будет просто вызывать этот метод у клиента и получать правильный результат без единого if и даже не зная, что у класса Client есть подклассы с разными алгоритмами вычисления.

Таким образом конкретный код, выполняемый при вызове метода объекта, определяется классом объекта, находящегося по ссылке в момент исполнения программы. Это свойство в ООП называется полиморфизмом.

3.1.6 Пример использования ООП

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15. | double[][] m1 = new double[3][3];  double[][] m2 = new double[3][3];  // multiply  int m = m1.length;  int n = m2[0].length;  int o = m2.length;  double[][] res = new double[m][n];  for (int i = 0; i < m; i++) {  for (int j = 0; j < n; j++) {  for (int k = 0; k < o; k++) {  res[i][j] += m1[i][k] \* m2[k][j];  }  }  } |

Хочу продемонстрировать особенности ООП на примере из другой, более простой предметной области. Предположим, что мы пишем программу, реализующую работу с матрицами. Математическими матрицами, двумерными наборами чисел. До знакомства с ООП мы вероятно представляли бы матрицы в программе в виде двумерных массивов чисел. Доступ к элементу осуществлялся при помощи квадратных скобок. Это бы неплохо работало до тех пор, пока мы с целью экономии памяти не решили бы хранить диагональные матрицы более компактным способом в виде одномерного массива. В этот момент весь ранее написанный код придется перелопатить и отдельно поддержать там особый случай диагональной матицы. Затем появилась бы оптимизация под ленточные матрицы и все пришлось бы опять переписывать.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17. | Matrix m1 = new FullMatrix(new double[3][3]);  Matrix m2 = new DiagonalMatrix(  new double[3]);  // multiply  int m = m1.getHeight();  int n = m2.getWedth();  int o = m2.getHeight();  double[][] res = new double[m][n];  for (int i = 0; i < m; i++) {  for (int j = 0; j < n; j++) {  for (int k = 0; k < o; k++) {  res[i][j] += m1.get(i,k) \*  m2.get(k,j);  }  }  }  Matrix m3 = new FullMatrix(res); |

Посмотрим, как все эти проблемы можно было бы решить в объектно‑ориентированном стиле. Надо завести класс Matrix, который скроит от остальной программы, то есть инкапсулирует особенности хранения и выполнения различных операций с матрицей. У класса Matrix будет несколько подклассов: DiagonalMatrix (диагональная), ленточная матрица, верхнетреугольная или нижнетреугольная матрица и так далее. Все они представляют одинаковый набор методов, но с разной реализацией, знающих про особенности хранения данного типа матрицы. Код, работающий с матрицами, пользуется предоставляемыми методами и, благодаря полиморфизму, может ничего не знать о том, что вообще-то матрицы бывают разными.

3.1.7 Так что насчет Java

Все сказанное про ООП естественным образом ложится на Java. В Java можно описывать классы и создавать их экземпляры — объекты. Благодаря модификаторам доступа все внутреннее состояние и прочие детали реализации можно скрыть от посторонних глаз, таким образом обеспечить инкапсуляцию. Для классов поддерживается наследование и переопределение методов. И это дает нам полиморфизм. Далее мы будем рассматривать конкретные ключевые слова и синтаксические конструкции, при помощи которых в Java описываются классы, их поля и методы.

3.2 Пакеты и модификаторы доступа

3.2.1 Ключевое слово package

Итак, мы собираемся писать программы в объектно-ориентированном стиле. Если следовать духу ООП, то наша программа будет разбита на маленькие классы, каждый из которых слабо связан с остальными и сильно специализирован, то есть решает одну задачу, а не делает все сразу.

Когда количество классов в нашей программе переваливает за десяток, то возникает потребность как-то наши классы упорядочивать и структурировать. Для этого в Java есть пакеты. О них то мы сейчас и поговорим.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | package org.stepic.java;  public class HelloWorld {  // ...  } |

Принадлежность класса пакету обозначается директивой package в самом начале файла с исходным кодом класса. Обычно эта директива идет первой же строчкой. Перед ней могут быть только комментарии. 1 — имя пакета. С учетом того, что класс HelloWorld находится в этом пакет, полное имя класса будет org.stepic.java.HelloWorld. Когда вы запускаете виртуальную машину командой java, то ей всегда нужно передавать полное имя класса. Если в файле несколько классов, то директива package действует на все. В отсутствии директивы package класс принадлежит пакету по умолчанию. А полное имя класса совпадет с его коротким именем HelloWorld. В Java принято раскладывать исходники в соответствии с их пакетами. Исходный код такого класса должен находится в файле org/stepic/java/HelloWorld.java. То же справедливо и для скомпилированного байт‑кода. Класс файлы будут разложены компилятором по директориям в соответствии с их пакетами org/stepic/java/HelloWorld.class. Это важно для виртуальной машины, которая, когда и потребуется загрузить класс с таким полным именем org.stepic.java.HelloWorld, будет искать его строго в файле с именем org/stepic/java/HelloWorld.class и нигде больше. Пакеты в Java напоминают найспесы в C++. Пакет задает область видимости класса. Другие классы этого же пакета могут обращаться к классу HelloWorld по его короткому имени.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10. | package org.stepic.java.other;  import org.stepic.java.HelloWorld;  import java.util.\*;  import static java.lang.Math.sqrt;  import static java.lang.System.out;  public class OtherClass {  // ...  } |

Классы из других пакетов должны ссылаться на класс HelloWorld по его полному имени, либо использовать директиву import. Импортировать можно либо конкретные классы, либо все содержимое пакета сразу.

Кстати, такие классы как Integer и прочие обертки String, System, с которыми мы уже имели дело, на самом деле находится в стандартном пакете java.lang. Однако, импортировать нам их не приходилось. Это потому, что в любой Java‑программе подразумевается неявный импорт всех классов из пакета java.lang. Классы этого пакета всегда доступны по их коротким именам.

Интересная особенность. Классы из пакета по умолчанию невозможно использовать из других пакетов даже при наличии явного импорта. Есть еще одна форма директивы импорта — import static, которая позволяет импортировать статические поля и методы. То есть в тексте программы можно будет обращаться к ним не указывая имя класса. Например, sqrt(4), out.println("foo"). Также есть вариант со \*, который одним махом импортирует сразу все статические поля и методы из класса. Иногда удобно, но злоупотреблять этим не стоит.

Подчеркну, что директива import служит только одной цели. Она позволяет в исходном коде вашей программы ссылаться на импортированные классы по их коротким именам. Никакой подстановки тело импортированных классов в ваш файл не происходит, то есть это не аналог директивы инкуд из C. Когда ваша программа исполняется в виртуальной машине, то все импортированные классы также должны присутствовать в classpath.

3.2.2 Именование пакетов

Классы стандартной библиотеки в Java находятся в пакетах с префиксами java и javax. Здесь приведены пакты, классы из которых используются чаще всего: java.lang, java.io, java.nio, java.math, java.time, java.util, java.util.regex, javax.xml.

Копаясь в jarax, входящий в состав JRE, вы можете также наткнуться на пакеты com.oracle, com.sun, sun и массу классов в них. Эти классы являются спецификой реализации оракловой Java и не предназначены для использования прикладными программистами. Во-первых, они доступны только в реализации Java от Oracle. Соответственно программы, использующая их, не будет переносима. Во‑вторых, никаких гарантий, что эти классы не будут удалены, переименованы или другим несовместимым образом изменены в следующей версии Java.

Для пользовательский пакетов в Java существуют следующие соглашения. Пакеты принято называть в соответствии с доменными именами компании, продукта или проекта, в рамках которой этот код написан. Например, для моего курса на stepic.org, согласно этой рекомендации, имя пакета должно начинаться с org.stepic. Код, который пишут в компании Google, будет в пакетах, начинающихся на com.google. Многие проекты, исходники которых хостятся на сорт форч имеют имя пакета, начинающийся на net.sf. Внутри уже можно заводить произвольную структуру подпакетов. Примеры: org.stepic.java, com.google.common, org.apache.maven, com.intellij.idea, net.sf.json, io.netty.

Смысл правила про использование доменных имен при именовании пакетов состоит в том, чтобы снизить вероятность коллизии имен классов, написанных разными людьми. Благодаря этому можно, например, найти в интернете найти какую-то Java‑библиотеку, скачать джарник, подключить своей программе и не беспокоиться о том, что классы из этого джара будут конфликтовать с классами вашей программы или классами других подключенных библиотек.

Помимо задания области видимости и предотвращения коллизии имен пакеты могут служить для группировки связанных классов внутри программы. Например, все классы, относящиеся к ядру программы, могут быть собраны в пакете core, разнообразные утилиты класса в пакете util, подключенные модули в пакете plugins.

3.2.3 Модификаторы доступа

И еще одна функция пакетов — это инкапсуляция на более высоком уровне, чем классы. При помощи модификаторов доступа можно сделать класс, метод или поле доступным только в пределах пакета. То есть можно скрыть какие-то детали реализации внутри пакета так, что они будут недоступны классам из других пакетов. Это особенно важно, когда вы, например, пишите библиотеку, которую будут использовать другие программисты. В этом случае вам нужно выставить наружу только строго определенное подмножество классов и методов, а все остальное — детали реализации вашей библиотеки — максимально скрыть.

Казалось бы, кому могут понадобиться ваши внутренние классы? Также думали и разработчики Sun Microsystems. И вот теперь мы имеем огромное количество программ, зависящих от классов с пакетами sun.misc или com.sun, которые никогда не были предназначены для широкого использования. Так что не надо недооценивать сообразительность и длину рук программистов, которые будут использовать написанный вами код.

Против длинных рук других программистов помогут специальные ключевые слова — модификаторы, устанавливающие правила доступа к вашим классам, их полям и методам.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | public class ModifiersDemo {  public static void visibleEverywhere() {}  protected static void inSubclasses() {}  static void inPackage() {}  private void inClass() {}  } |

В Java есть 4 типа доступа: public — разрешен доступ отовсюду без ограничений, protected — доступ разрешен только для классов наследников и для классов текущего пакета, отсутствие модификатора — доступ только в пределах пакета и privat — доступ только в пределах класса. Модификаторы protected и private не применимы к классам верхнего уровня. Их можно применять к вложенным классам. О вложенных классах мы поговорим отдельно. Также в некоторых контекстах отсутствие модификатора означает не пакетную видимость, а что-то другое. Но об мы тоже будем говорить потом, когда придет время.

Стоит хорошенько думать о том, какие элементы вашей программы вы выставляете в общее пользование, а какие нет. Если сомневаетесь, то лучше поставить более строгий модификатор доступа. Если в один прекрасный момент действительно появится реальная необходимость переиспользовать ваш код в другом месте, то поставить другой модификатор доступа будет нетрудно. А вот если вы изначально объявили ваш класс как публичный, а потом передумали, то вам, возможно, придется перелопатить немаленькое количество кода, где ваш класс таки могли успеть заиспользовать. И вообще далеко не всегда вам доступен весь код, где могут использоваться ваши классы. Например, если вы выкладываете библиотеку в интернет, то вы уже ничего не контролируете. Фактически вы подписываетесь поддерживать обратную совместимость, то есть не удалять, не переименовывать и не менять другим несовместимым образом ничего из публичных классов, методов и полей. Пользователи очень не любят, когда новая версия библиотеки ломает им их код.

3.3 Объявление класса

3.3.1 Класс

Разберем теперь подробно объявление класса. Какие ключевые слова могут использоваться в объявлении класса, что они означают и, вообще, чем может быть класс наполнен.

Для данного урока я подобрал примеры из стандартной библиотеки Java реализации от Oracle. Так что мы немного даже заглянем в исходники стандартной библиотеки. Но должен предупредить, что код приведен со значительными сокращениями, чтобы не отвлекаться на вещи, которые мы, например, еще не знаем, и сосредоточиться на главном.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | package java.lang;  public find class Integer {  // ...  } |

Итак, приступим. Класс объявляется ключевым словом class с именем класса и модификатором доступа. В случае класса верхнего уровня может использоваться модификатор public или модификатора может не быть, что означает видимость класса в пределах пакета. Модификаторы private и protected здесь лишены смысла.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | package java.lang;  public find class Integer {  // ...  }  class Foo {} |

Напомню, что в одном исходном файле друг за другом может быть объявлено несколько классов, но public может быть только один из них и его имя обязано совпадать с именем файла. Ключевое слово final в объявлении класса означает, что от данного класса нельзя наследоваться, то есть он является финальным в иерархии наследования. Подробнее про наследование мы поговорим в следующем уроке.

|  |  |  |
| --- | --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14. | package java.lang;  /\*\*  \* The {@code Integer} class wraps a value of  the primitive type  \* {@code int} in an object. An object of  type {@code Integer}  \* contains a single field whose types is  {@code int}  \*/  public find class Integer {  // ...  } | package java.lang;  public find class Integer {  // ...  }  class Foo {} |

Любой элемент программы (класс, метод или поле) может сопровождаться Javadoc‑комментарием. Классы стандартной библиотеки снабжены хорошими подробными Javadoc, поэтому часто можно решить какой‑то вопрос по использованию класса в стандартной библиотеке просто прочитав их Javadoc. Не обязательно сразу лезть на stackoverflow.

3.3.2 Поля класса

|  |  |  |
| --- | --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | package java.lang;  public final class Integer {  private final int value;  // ...  } | package java.lang;  public find class Integer {  // ...  }  class Foo {} |

Состояние экземпляра класса храниться в переменных членах класса, которые в Java называются полями. Объявление поля состоит из модификатора доступа, типа поля и его имени. Тут же можно присвоить полю начальное значение, которое поле будет иметь во всех свежесозданных экземплярах класса. Если значение не указано, то поле будет иметь значение по умолчанию — 0, null или false в зависимости от типа поля.

У каждого экземпляра будет свое собственное поле, значение которого можно менять не зависимо от других экземпляров. В данном примере поле объявлено как private. Это типичная практика в Java. В большинстве случаев состояние экземпляра класса, то есть значение его полей, является деталей реализации и должно быть скрыто. Например, у BigInteger внутри находится пол дюжины хитрых полей, но знать об этом остальному коду совершенно не нужно. Классы должны взаимодействовать друг с другом при помощи осмысленных методов, а не обещаться напрямую к состоянию друг друга.

|  |  |  |
| --- | --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12. | package java.lang;  public final class Integer {  private final int value;  public Integer(int value) {  this.value = value;  }  // ...  } | package java.lang;  public find class Integer {  // ...  }  class Foo {} |

Модификатор final означает, что значение поля можно присвоить только один раз. После чего менять его будет нельзя. Явное присвоение можно сделать прямо в объявлении поля или в конструкторе. Компилятор проверит, что к моменту выхода из конструктора значение поля будет обязательно присвоено. Если это не так, то будет выдана ошибка компиляции. Кстати, модификатор final можно применять и к локальным переменным, и к параметрам методов. Смысл его от этого не меняется. Значение переменной может быть присвоено только однажды.

3.3.3 Конструкторы

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12. | package java.lang;  public final class Integer {  private final int value;  public Integer(int value) {  this.value = value;  }  // ...  } |

Собственно, переходим к конструкторам. Конструктор— это специальный метод, вызываемый при создании экземпляра класса, то есть когда кто‑то где‑то в программе применяет оператор new. Задача конструктора — инициализировать состояние объекта и подготовить его к использованию. Объявление конструктора состоит из модификатора доступа и имени класса. Конструктор может принимать параметры, как и любой другой метод. Если параметр конструктора имеет такое же имя, как поле класса, то для доступа к полю класса используется префикс this. this — это ссылка на текущий экземпляр в контексте которого используется конструктор или любой другой метод. Обычно явно писать this не нужно, но в случае конфликта имен поля и локальной переменной или параметра приходится это делать.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12. | package java.lang;  public final class Math {  /\*\*  \* Don't let anyone instantiate this  clas.  \*/  private Math() {}  // ...  } |

Когда в классе не объявлен ни один конструктор, то неявно создается конструктор по умолчанию без параметров. То есть даже если в классе нет конструктора, его экземпляр все равно можно создать при помощи new. Если нужно запретить создание экземпляров класса, то нужно сделать конструктор приватным.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13. | package java.math;  public class BigInteger {  public BigInteger(String val) {  this(val, 10);  }  public BigInteger(String val,  int radix) {  // ...  }  } |

В классе может быть несколько перегруженных версий конструктора с разными наборами параметров. При этом из одного конструктора можно вызывать другой. В некоторых языках поддерживается задание значений по умолчанию для параметров, то есть можно было бы, например, написать прямо в объявлении метода, что int radix = 10 и это бы означало, что во всех методах вызовы конструктора, если 2-ой параметр не указан, то он подразумевается равным 10. В Java такой механизм не поддерживается, но при помощи перегрузки конструкторов и вызова одного конструктора из другого его можно успешно эмулировать.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14. | package java.io;  public class FileInpetStream {  protected void finalize() {  // cleanup  }  public void close() {  // cleanup  }  // ...  } |

Следом за конструктами логично было бы обсудить деструкторы, но деструкторов в Java нет. Строго говоря JVM поддерживает некоторый специальный метод finalize(), который будет вызван сборщиком мусора, когда на объект больше нет ссылок.

Однако, после многих лет наступании на грабли сообщество Java‑программистов пришло к консенсусу, что полагаться на эту возможность не следует. В первую очередь потому, что метод finalize() будет вызван в непредсказуемый момент времени и из непредсказуемого потока. Если объекту все‑таки требуется какая‑то подчистка, освобождение ресурсов, то для этого заводят обычный метод, чаще всего с именем close(), и ответственность за вызов этого метода возлагается на программиста. Впрочем, как мы узнаем в следующем модуле в язык встроена удобная синтаксическая конструкция, обеспечивающая неявный вызов метода close(), при выходе объекта из области видимости.

3.3.4 Методы

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12. | package java.lang;  public final class Integer {  private final int value;  public int intValue() {  return value;  }  // ...  } |

От конструкторов мы уже плавно перешли к обсуждению методов. Объявление метода состоит из модификатора доступа, типа возвращаемого значения или void, имени и параметров. Метод исполняется в контексте конкретного экземпляра класса, поэтому он может обращаться к полям текущего объекта. Метод может иметь модификатор final. Это означает, что данный метод не может быть переопределен в классах наследниках. Впрочем, в данном примере класс уже объявлен как final, поэтому расставлять final еще на всех методах избыточно. Все равно от этого класса не унаследоваться.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15. | package java.lang;  public final class String {  public int indexOf(int ch) {  return indexOf(ch, 0);  }  public int indexOf(int ch,  int fromIndex) {  // ...  }  // ...  } |

В классе может быть несколько перегруженных методов с одинаковыми именами, но разными наборами параметров. За счет этого можно, например, эмулировать значение по умолчанию для параметров.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15. | package java.lang;  public final class Integer {  public static final int MIN\_VALUE =  0x80000000;  public static int rotateRight(int i,  int distance) {  return (i >>> distance) |  (i << -distance);  }  // ...  } |

Поля и методы могут быть статическими, то есть объявленными с модификатором static. Как минимум один статический метод присутствует в любой программе. Это метод main(). Статические поля и методы существуют независимо от экземпляров класса и могут вызываться просто по имени класса. То есть, например, Integer.rоtateRight(5, 7). Статический метод исполняется в контексте класса, а не конкретного экземпляра, поэтому он не имеет доступа к this и нестатическим полям и методам.

В комбинации с final модификатор static используется для объявления констант. Ключевое слово const, известное по другим языкам, в Java не поддерживается. Имена констант принято записывать в верхнем регистре (MIN\_VALUE), разделяя слова подчеркиванием.

3.3.5 Immutable-классы. Вложенные классы

Делая поля класса приватными и не предоставляя методов для изменения значения полей, мы тем самым можем добиться того, что экземпляры нашего класса после создания уже не могут быть никак модифицированы. В качестве примера из того, с чем мы уже знакомы, можно привести все классы обертки для примитивов, а также классы String, BigInteger и BigDecimal. Любое изменение состояния объекта возможно только при создании нового экземпляра. С одной стороны это может быть не очень экономно, но с другой стороны такие классы обладают очень ценным свойством. Передавая экземпляры immutable‑класса в какой-нибудь метод или иным образом, вынося объект на всеобщее обозрение, мы можем быть уверены, что никакой посторонний код случайно или намерено не изменит нам состояние нашего объекта.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14. | package java.math;  public class BigInteger {  public statac final BigInteger ONE =  valueOf(1);  public static BigInteger valueOf(  long val) {  // ...  }  // ...  } |

Например, в классе BigInteger есть часто используемая константа ONE. Это BigInteger, представляющий собой единицу. А теперь на секунду представим, что в классе BigInteger есть метод setValue(), заменяющий значение внутри объекта. И какой-нибудь злодей взял бы и написал бы у себя где-нибудь в коде:

|  |  |
| --- | --- |
| 1. | BigInteger.ONE.setValue(100500) |

Думаю, вы можете представить себе последствия. Это почти также коварно, как в C написать дифайнтруфолс. К счастью, метода setValue() или другого аналогичного ему в классе BigInteger нет.

Вернемся к теме объявления классов. На самом деле классы можно объявлять не только на верхнем уровне программы, но и внутри других классов. Следующий пример из стандартной библиотеки будет содержать некоторые пока незнакомые нам слова и символы, которые мы обсудим несколько позже. Но пример этот я покажу уже сейчас, потому что он хорошо очень иллюстрирует полезность вложенных классов.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17. | package java.util;  public class ArrayList<E> {  Object[] elementData;  public Iterator<E> iterator() {  return new Itr();  }  private class Itr implemens Iterator<E> {  int cursor;  // ...  }  // ...  } |

ArrayList — это коллекция-контейнер для однотипных элементов. Но в отличие от массива он может динамически изменять свой размер. Когда требуется обойти и обработать все элементы коллекции, используется итератор — объект, по очереди возвращающий каждый элемент коллекции. Итератор должен помнить коллекцию, которую он обходит, чтобы получить его элементы, а также хранить текущую позицию обхода. Этого легко добиться, поместив определение класса итератора (11) внутрь класса ArrayList. Каждый экземпляр итератора будет иметь собственное состояние (12), хранящиеся в его полях, а также неявную ссылку на экземпляр внешнего класса. То есть из кода итератора можно напрямую обращаться к полям экземпляра ArrayList, в том числе к приватным.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16. | package java.util;  public class Collecions {  public static final List EMPTY\_LIST =  new EmplyList<>();  public static final <T> List<T>  emplyList() {  return (List<T>) EMPTY\_LIST;  }  private static class EmplyList<E> {  // ...  }  } |

Если вложенный класс снабжен модификатором static, то неявная связь с внешним классом теряется. И экземпляры вложенного класса будут жить своей собственной независимой жизнью. При этом размещение одного класса внутри другого мотивируется либо желанием скрыть вложенный класс, сделав его приватным, либо тесной логической связью внешнего и вложенного классов, требующий доступа к приватным полям друг друга.

3.3.6 Перечисления

Кроме классов в Java бывают еще интерфейсы, но о них в отдельном уроке, а также перечисление и аннотации. О них поговорим прямо сейчас.

Чтобы подвести разговор к перечислениям, давайте рассмотрим следующий выдуманный пример. Любые совпадения с реальным кодом совершенно случайны.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10. | public class BadExample {  public static final int MONDAY = 1;  public static final int TUESDAY = 2;  public static final int WEDNESDAY = 3;  public static final int THURSDAY = 4;  public static final int FRIDAY = 5;  public static final int SATURDAY = 6;  public static final int SUNDAY = 7;  } |

Предположим, нам в программе надо работать с календарем и, в частности, с днями недели. День недели можно было бы хранить в памяти в виде целого числа его порядкового номера от 1 до 7 или от 0 до 6, кому как больше нравится. Это будет работать, но по типу переменной будет совершенно не очевидно, что в ней можно храниться не любое число, а только одно из перечисленных. Будет легко ошибиться и компилятор не сможет помочь нам следить за корректностью программы int dayOfWeek = ???.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12. | package java.time;  public enum DayOfWeek {  MONDAY,  TUESDAY,  WEDNESDAY,  THURSDAY,  FRIDAY,  SATURDAY,  SUNDAY  } |

Вот как раз для таких случаев в Java и существует перечисления. Перечисления объявляются ключевым словом enum. В фигурных скобках через запятую указывается набор допустимых значений перечисления. Перечисления — это полноценный ссылочный тип. Его можно воспринимать как класс с фиксированным количеством экземпляров. Перечисленные в объявлении enum значения эквиваленты public static final полям класса. Более того в перечислениях можно объявлять поля и методы. Достаточно поставить точку запятой после SUNDAY, а дальше объявлять поля и методы как в обычном классе. Можно также объявить конструктор, в том числе принимающем параметры. В таком случае параметры нужно будет передавать и в объявлении элементов перечисления: FRIDAY(args).

|  |  |
| --- | --- |
| 1.  2.  3.  4. | for (DayOfWeek fay:DayOfWeek.values()) {  System.out.println(day.ordinal() + " " +  day.name());  } |

В любом перечислении автоматически доступны методы name(), ordinal(), а также статический метод values(). name() возвращает строку — имя элемента перечисления как в исходном коде. ordinal() возвращает число — порядковый номер элемента, начиная с нуля в соответствии с объявлении в исходном коде. values() возвращает массив возможных значений перечислений в том же порядке. Зная это легко догадаться, что этот фрагмент кода выводит все элементы перечисления с их порядковым номерами и именами.

3.3.7 Аннотации

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15. | package java.lang;  public final class Character {  @Deprecated  public static boolean isJavaLetter(  char c) {  //...  }  @SuppressWarning("unchecked")  public static final Class<Character>  TYPE = (Class<Character>)  Class.getPrimitiveClass("char");  } |

И буквально два слова об аннотации. В коде стандартной библиотеке в изобилии встречаются пометки @Deprecated и @SuppressWanings(), а также некоторые другие на классах и методах.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13. | package java.lang;  import java.lang.annotation.\*;  import static  java.lang.annotation.ElementType.\*;  @Target({TYPE, FIELD, METHOD, PARAMETR,  CONSTRUCTOR, LOCAL\_VARIABLE})  @Retention(RetentionPolicy.SOURCE)  public @interface SupprsessWarnings {  Strng[] value();  } |

Это не ключевые слова языка, а специальные аннотации, объявленные также в стандартной библиотеке при помощи ключевого слова @interface. Забавно, что характеристики, объявляемые аннотацией, также создается при помощи аннотации.

Программист может заводить свои собственные аннотации, расставлять их на элементах своей программы и впоследствии как-то использовать. Например, на основе аннотации можно генерировать код или делать что‑нибудь интересное во время исполнения программы. Но это уже для продвинутого курса. Сейчас я расскажу только про смысл 2-х упомянутых аннотаций, которые вы можете использовать и в ваших собственных программах.

Аннотация @Deprecated означает, что класс или метод устарел и не рекомендуется к использованию. Обычно рядом в Javadoc описывается, что именно следует использовать вместо него. На использование Deprecated элементов в вашей программе компилятор будет вызывать предупреждение.

Аннотация @SuppressWarnings() временно отключает предупреждения компилятора на подозрительные места в коде в рамках всего класса, метода или объявления поля в зависимости от того, где аннотация размещена. Какие именно предупреждения отключить указывается в параметре аннотации.

Разумеется, правильный способ борьбы с предупреждениями компилятора — это исправлять проблемы в коде. Но в редких случаях бывает так, что код, компилирующийся без warning, не написать. Тогда можно поставить аннотацию @SuppresWarning() и подробно откомментировать почему, иначе не написать, и почему, несмотря на предупреждения, код будет работать корректно.

3.4 Наследование. Класс Object

3.4.1 Наследование, возможности переопределения

Одна из фундаментальных концепций объектно-ориентированного программирования — это наследование. Речь идет о тем, что классы можно создавать не с нуля, а наследуя или расширяя другие существующие классы.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12. | package java.lang;  public final class BigDecimal  extends Number {  public int intValue() {  // ...  }  // no shortValue() method,  // it's inherited from Number  } |

Наследование в Java обозначается ключевым словом extends. Наследоваться можно только от одного класса. Унаследованный класс автоматически получает все поля и методы базового класса. Пользоваться ими с учетом модификатора доступа может как он сам, так и другие классы программы. Причем им совершенно не важно объявлено ли поле или метод непосредственно в классе или в одном из его родительских классов. А иерархия наследования может быть произвольной глубины.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12. | package java.lang;  public final class BigDecimal  extends Number {  public int intValue() {  // ...  }  // no shortValue() method,  // it's inherited from Number  } |

Сравните. Метод intValue определен в классе BigDecimal. Метод shortValue() объявлен в его родительском классе Number. Имея на руках экземпляр bd, мы можем вызывать на нем как intValue(), так и shortValue(): bd.intValue() и bd.shortValue(), не задумываясь о том в каком уровне иерархии наследования объявлены эти методы. Работа с ними выглядит абсолютно одинаково.

В классе наследнике можно добавлять поля и методы тем самым расширяя возможности базового класса. Кроме того, методы можно переопределять, то есть в классе наследнике можно завести метод с таким же именем и параметрами как в базовом классе. При этом тип возвращаемого значения должен либо совпадать, либо быть подклассом типа возвращаемого базового метода. А также модификатор доступа должен быть либо тем же, либо более открытым. Если эти условия выполнены, то метод класса наследника заменит собой одноименный метод из базового класса.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14. | package java.lang;  public final class StringBuilder  extends AbstractStringBuilder {  @Override  public StringBuilder append(String str) {  // ...  }  // base method in AbstractStringBuilder:  // AbstractStringBuilder  // append(String str)  } |

В следующем примере класс StringBuilder наследуется от некого AbstractStringBuilder и переопределяет его метод append(), принимающий строку. При этом возвращаемый тип отличается, но переопределение является корректным, так как возвращаемый StringBuilder является подклассом возвращаемого базового метода AbstractStringBuilder. А вот если бы мы здесь стали возвращать какой‑нибудь int или String, то код бы не скомпилировался. Хорошая практика ставить на переопределенных методах аннотацию @Override. Она явно декларирует ваше намерение переопределить одноименный метод базового класса.

Это может показаться избыточным. Ведь переопределение происходит не зависимо от наличия аннотации. Однако, это полезно для дополнительной самопроверки. Например, можно ошибиться в имени метода или в его параметрах и таким образом создать самостоятельный метод, не перекрывающий метод суперкласса. Или объявление суперкласса может измениться без вашего ведома и сигнатуры методов разойдутся. При наличии аннотации @Override компилятор сразу укажет вам на эту проблему.

3.4.2 Конструктор класса‑наследника

Создание экземпляра класса наследника всегда включает в себя инициализацию базового класса, то есть вызов его конструктора. Когда у базового класса есть конструктор без параметров, то компилятор автоматически поставит его вызов в начало тело конструктора класса наследника. Программисту можно явно это не писать.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17. | package java.lang;  public final class StringBuilder  extends AbstractStringBuilder {  public StringBuilder() {  super(16);  }  @Override  public StringBuilder append(String str) {  super.append(str);  return this;  }  // ...  } |

Однако в тех случаях, когда нужен вызов конструктора с параметрами, его придется сделать явно, так как компилятор не в состоянии сам сообразить, что именно нужно передавать в качестве параметра. Вызов конструктора суперкласса делается при помощи ключевого слова super() и он обязан идти перед любым другим кодом, размещенным в конструкторе. Есть так же возможность вызвать реализацию метода из базового класса несмотря на наличие переопределенной версии методов в текущем классе. Для этого тоже используется ключевое слово super(). Если бы в данном примере мы его убрали, то случилась бы бесконечная рекурсия, поскольку метод append() вызывал бы сам себя.

Использование ключевого слова super() разрешено только в теле класса наследника. Его внешние пользователи уже не могут управлять тем, какую версию метода вызывать: базовую из родительского класса или переопределенную. Всегда будет вызываться переопределенная версия из класса наследника.

3.4.3 Класс Object. Методы toString(), equals(), hashCode()

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | package java.lang;  public final class String  /\*extends Object\*/ {  // ...  } |

Когда в объявлении класса ничего не сказано про наследование, то есть отсутствует ключевое слово extends, то подразумевается, что класс расширяет java.lang.Object. Object — это корень всей иерархии наследования, родитель всех остальных ссылочных типов, в том числе массивов, перечислений и аннотаций.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18. | package java.lang;  public class Object {  pubic String toString() {  return getClass().getName() + "@" +  Integer.toHexString(  hashCode());  }  public boolean equals(Object obj) {  return this == obj;  }  public native int hashCode();  // ...  } |

Из этого следует, что абсолютно все классы в Java содержат методы, объявленные в классе Object. Там их больше десятка, но основные, самые часто используемые, — это toString(), equals() и hashCode().

Метод toString() преобразует объект в строку. Он вызовется, например, когда мы выводим объект в консоль при помощи System.out.println() или когда конкатенируем строку с объектом. Разумеется, его можно вызвать и явно, используя в своих целях. Реализация метода toString() в классе Object() формирует строку вида имяКласса@16‑ричныйХешКод (java.lang.Object@3fe45), но многие подклассы переопределяют его для возврата чего‑то более осмысленного.

Метод equals() предназначен для сравнения объектов по содержимому. В дополнении к сравнению по ссылке, реализуемому оператором равно‑равно. Реализация по умолчанию также сравнение ссылок. Просто потому, что на данном уровне абстрактного непонятного объекта ничего другого предложить еще нельзя. Но эту реализацию можно и нужно переопределять в подклассах, если для них будет востребовано сравнение по содержимому. Реализация переопределена у многих стандартных классов. Досадным исключением являются только массивы.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18. | package java.lang;  public final class String {  @Override  public boolean equals(Object anObject) {  if (this == anObject) {  return true;  }  if (anObject instanceof String) {  String other = (String)anObject;  // ...  }  return false;  }  // ...  } |

Метод equals() принимает параметр типа Object. Это нужно для того, чтобы можно было сравнивать объекты, не зная заранее их типа. Поэтому первое, что надо сделать в пользовательской реализации equals(), — это проверить правда ли переданный объект является экземпляром правильного класса. Если нет — вернуть false. Хотя иногда в целях оптимизации первым делают проверку ссылок, чтобы быстро вернуть true, когда объект сравнивается сам с собой. Проверка типа обычно делается при помощи оператора instanceof, который проверяет является ли объект, находящийся по ссылке, экземпляром заданного класса. При этом подклассы тоже считаются. То есть, например, проверка экземпляра BigDecimal на instanceof Number вернет true. По соглашению, если в equals() передадут null, он не должен падать, а должен мирно вернуть false. Оператор instanceof справится и с этим. Проверка null на instanceof чего угодно вернет false. Дальше приводим объект к нужному типу и сравниваем уже детально по полям.

И третий метод класса Object, о котором я скажу несколько слов, — это hashCode(). Это метод для вычисления хеш‑кода объекта. Хеш‑код представляется целым числом типа int. По большому счету он нужен только для того, чтобы объекты можно было хранить в хеш‑таблицах. Хеш‑таблицы мы будем с вами обсуждать в модуле, посвященным коллекциям.

Существует требование на согласованность реализации методов equals() и hashCode(). А именно, если два объекта равны с точки зрения метода equals(), то у них обязаны быть одинаковый хеш‑код. Если для каких‑то объектов это правило нарушается, то вы не сможете хранить эти объекты в хеш‑таблице. Точнее сможете, но будете постоянно получать неправильные результаты. Поэтому рекомендуется либо переопределять одновременно и equals(), и hashCode(), либо не переопределять ни один из них, чтобы случайно не внести рассогласованность в их реализации.

Кстати, модификатор native означает, что реализация данного метода находится в нативном C‑ном коде, то есть чтобы увидеть ее нужно выкачивать исходники HotSpot.

Часто можно услышать утверждение, что хеш‑код в Java возвращает адрес объекта. Именно такая реализация хэш‑кода описана в Javadoc к методу hashCode(). Однако полагаться на это нельзя — это всего лишь пример, а не требование. Поэтому на самом деле то, что возвращает hashCode() зависит от конкретной виртуальной машины, а так же от параметров с которыми она запущена. Хеш‑коды, вычисляемые на основе адреса объекта, плохи тем, что они будут похожи у последовательно созданных объектов, расположенных друг за другом в памяти. HotSpot по умолчанию вычисляет хеш‑код как псевдослучайные числа, генерируемые и сохраняемые в объекте при первом вызове метода hashCode().

3.3.4 Пример семейство геометрических фигур

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19. | package org.stepic.java.shapes;  public class Shape {  private final Color color;  public Shape(Color color) {  this.color = color;  }  public Color getColor() {  return color;  }  public double getArea() {  return Double.NaN;  }  } |

Давайте рассмотрим пример программы, в которой используется наследование классов, чтобы почувствовать все услышанное на практике. Сначала опишем предметную область. Наша программа будет работать с геометрическими фигурами, поэтому нам понадобятся классы для представления фигур. Базовым для всех фигур будет класс Shape. Здесь мы объявим общее для всех фигур свойства, такие как цвет и возможность вычислить площадь.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10. | package org.stepic.java.shapes;  public enum Color {  BLACK,  WHITE,  RED,  GREEN,  BLUE  } |

Цвет в данном примере мы сделаем просто перечислением. Поскольку конкретный тип фигуры и ее координаты на данном уровне наследования неизвестны, то метод вычисления площади будет просто заглушкой, возвращающий NaN. Менять цвет фигуры после создания фигуры не предполагается, поэтому поле Color объявлено финальным. Опишем три конкретных типа фигур.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28.  29.  30.  31.  32.  33.  34.  35.  36. | package org.stepic.java.shapes;  public class Circle extends Shape {  private final Point center;  private final double radius;  public Circle(Point center,  double radius, Color color) {  super(color);  this.center = center;  this.radius = radius;  }  public Point getCenter() {  return center;  }  public double getRadius() {  return radius;  }  @Override  public double getArea() {  return radius \* radius \* Math.PI;  }  @Override  public String toString () {  return "Circle{" +  "center=" + corner +  ", radius=" radius +  ", color=" + getColor() +  '}';  }  } |

Circle — это круг. Он задается своим центром и радиусом. Центр задается точкой.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25. | package org.stepic.java.shapes;  public class Point {  private final double x;  private final double y;  public Point(double x, double y) {  this.x = x;  this.y = y;  }  public double getX() {  return x;  }  public double getY() {  return y;  }  @Override  public String toString() {  return "{" + x + ", " + y + ')';  }  } |

Для этого я завел еще один небольшой класс с двумя координатами x и y.

У класса Shape есть только один конструктор и он принимает в качестве параметра цвет, поэтому в конструкторе класса Circle нам нужно явно вызвать этот конструктор, передав туда значение цвета. Переопределим здесь метод вычисления площади и метод toString(), чтобы можно было вывести объект на печать и увидеть его содержимое, а не имя класса с ничего не говорящим хеш‑кодом.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28.  29.  30.  31.  32.  33.  34.  35.  36. | package org.stepic.java.shapes;  public class Square extends Shape {  private final Point corner;  private final double size;  public Square(Point corner,  double size, Color color) {  super(color);  this.corner = corner;  this.size = size;  }  public Point getCorner() {  return corner;  }  public double getSize() {  return size;  }  @Override  public double getArea() {  return size \* size;  }  @Override  public String toString () {  return "Square{" +  "corner=" + corner +  ", size=" size +  ", color=" + getColor() +  '}';  }  } |

Второй тип фигуры — квадрат. Он задается одной из своих вершин и длиной стороны, размером. Также переопределим вычисление площади и метод toString().

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28.  29.  30.  31.  32.  33.  34.  35.  36. | package org.stepic.java.shapes;  public class Triangle extends Shape {  private final Point a;  private final Point b;  private final Point c;  public Triangle(Point a, Point b,  Color color) {  super(color);  this.a = a;  this.b = b;  this.c = c;  }  @Override  public double getArea() {  return Math.abs(  (a.getX() – c.getX()) \*  (b.getY() – c.getY()) -  (b.getX() – c.getX()) \*  (a.getY() – c.getY())  ) / 2;  }  @Override  public String toString () {  return "Triangle{" +  "a=" + a +  ", b=" b +  ", c=" c +  ", color=" + getColor() +  '}';  }  } |

И третий тип — треугольник. Задается 3‑мя своими вершинами. И так же мы здесь определяем методы вычисления площади и метод toString().

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28.  29.  30.  31.  32.  33.  34.  35.  36.  37.  38.  39.  40.  41.  42.  43.  44.  45.  46.  47.  48.  49.  50.  51.  52.  53.  54.  55.  56.  57. | package org.stepic.java.shapes;  public class Main {  public static void main(String[] args) {  Circle circle = new Circle(  new Point(0, 0), 1,  Color.BLACK);  Triangle triangle = new Triangle(  new Point(0, 0),  new Point(1, 0),  new Point(0, 1),  Color.RED);  Square square = new Square(  new Point(5, 5), 2,  Color.BLUE);  Shape shape = triangle;  Object object = triangle;  triangle = (Triangle) object;  Shape[] shapes = {circle, triangle,  square};  printArrayElements(shapes);  Shape maxShape =  findShapeWithMaxArea(shapes};  System.out.println(  "Shape with max area: " +  maxShape);  }  private static void printArrayElements(  Object[] objects) {  for (int i = 0; i < objects.length;  i++)  System.out.println(i + ": " +  objects[i]);  }  }  private static Shape  findShapeWithMaxArea(  Shape[] shapes) {  Shape maxShape = null;  double maxArea =  Double.NEGATIVE\_INFINITY;  for (Shape shape : shapes)  double area = shape.getArea();  if (area > maxArea) {  maxArea = area;  maxShape = shape;  }  }  return maxShape;  }  } |

Давайте теперь попробуем сделать с этим что‑то полезное. В методе main(), главного класса программы, создадим по одному экземпляру каждой из фигур при помощи оператора new, вызывающего конструктор. Имея на руках экземпляр конкретной фигуры, мы можем присвоить его в переменную базового класса shape или object. Такое присваивание всегда возможно и не требует оператора приведения типа. При этом объект, находящийся по ссылке, никак не меняется. Он продолжает оставаться экземпляром класса Triangle. Просто мы начинаем смотреть на него через ссылку более общего типа. Через эту ссылку нельзя вызывать ничего специфичного для треугольника, но можно вызывать все, что объявлено на уровне класса Shape и что по определению наследования так же доступно во всех подклассах этого класса. Приведение типа в обратную сторону от базового класса к классу наследнику возможно только в явном при помощи оператора приведения типа. Причем, если мы не угадали с типом объекта, то произойдет ошибка во время исполнения программы. Угадать нам поможет оператор instanceof. Однако частое использование instanceof — это признак неправильного использования объектно-ориентированного программирования. Один из немногих случаев, когда instanceof действительно нужен — это реализация метода equals(). Давайте теперь соберем все наши фигуры в массив и распечатаем содержимое массива. Для печати массива я завел небольшой утилитный метод, принимающий массив объектов. Почему объектов, а не фигур? Потому что данный метод не несет никакой специфики геометрических фигур и в переданных объектах здесь вызовется только метод toString(). Благодаря такому обобщению этим методом можно будет потом воспользоваться для печати любых других объектов. А еще мы можем взять и найти фигуру с максимальной площадью. Для этого я тоже завел здесь небольшой метод. Он в цикле обходит массив и запоминает элемент с максимальной площадью, которую в конце и возвращает. Здесь используется метод getArea(), объявленный в классе Shape и переопределенный в каждом из подклассов. Теперь запустим нашу программу и посмотрим, что получится. Итак, распечатались все элементы массива и найдена фигура с максимальной площадью — это квадрат.

Этот пример иллюстрирует важную вещь. Имея базовый класс для некоторого набора родственных классов, мы можем писать обобщенный код, оперирующий этим базовым классом. Коду будет не важно какие конкретные экземпляры ему переданы. Это возможно благодаря полиморфизму, то есть вызову той версии метода, которой соответствует объекту, находящемуся в данный момент по ссылке. Хотя наш код оперирует переменной типа Shape, но по факту вызовется та версия метода getArea(), которая определена в конкретном подклассе в зависимости от того на какой объект указывает наша переменная типа Shape.

3.4.5 О правильном использовании наследования

Наследование — это хорошо, но важно его правильно использовать. Если вы в какой‑то момент засомневаетесь стоит ли вам наследовать какой‑то свой класс от другого базового класса, то принять правильное решение вам может помочь принцип подстановки, сформулированный в 1987 году Барбарой Лисков. Данный принцип гласит: если S является подтипом T, то все объекты базового класса T в программе могут быть заменены на объекты произвольного класса S без изменения каких‑либо желательных свойств программы. Другими словами, поведение унаследованных классов должно соответствовать, то есть не должно противоречить поведению заданному базовому классам или поведение классов наследников должно быть ожидаемо для кода, оперирующими переменными базового класса. В нашем примере с геометрическими фигурами нарушением принципа подстановки был бы такой подкласс Shape, в котором метод getColor(), например, кидал бы исключение. Или метод getArea() возвращал бы вместо площади периметр фигуры. Такой класс нарушал бы контракт, заданный базовым классом Shape и передача такого объекта в код оперирующей переменной типа Shape привела бы к неправильным результатам работы программы.

И второй нюанс с наследованием. Даже в том случае, когда наследование удовлетворяет принципу подстановки следует еще раз смотреть альтернативный вариант с композицией, то есть вместо наследования класса S от класса T можно попробовать включить в класс S ссылку на экземпляр T. Наследование — это очень сильный тип связи между классами. Его чрезмерное использование может со временем привести к тому, что код будет трудно развивать и поддерживать. Композиция — это значительно более легковесный и гибкий тип связи. В нашем примере все фигуры включали в себя один или несколько экземпляров класса Point. И это значительно более предпочтительно, чем наследовать фигуры от класса Point.

3.5 Абстрактные классы и интерфейсы

3.5.1 Абстрактный класс

Класс может соответствовать не только конкретной сущности предметной области, но и какому-то абстрактному понятию. Например, Shape — абстрактная геометрическая фигура. Мы, конечно, можем писать код, использующий переменные типа Shape, но за ними будут всегда скрываться конкретные круги, квадраты и треугольники. Просто геометрической фигуры без типа существовать не может.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18. | package org.stepic.java.shapes;  public abstract class Shape {    private final Colore colore;    public Shape(Color color) {  return color = color;  }  public Color getColore() {  return color;  }  public double getArea() {  return Double.NaN;  }  } |

Для того, чтобы выразить это в программе, используется ключевое слово abstract. Если класс объявлен как абстрактный, это означает что нельзя создавать его экземпляры. Создать можно только экземпляр класса наследника, не являющегося абстрактным.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16. | package org.stepic.java.shapes;  public abstract class Shape {    private final Colore colore;    public Shape(Color color) {  return color = color;  }  public Color getColore() {  return color;  }  public abstract double getArea();  } |

В абстрактном классе может быть все то же, что и в обычном: поля, конструкторы и методы. Но некоторые методы могут абстрактными, т.е. без реализации. В данном классе мы должны сделать абстрактным метод getArea(), поскольку на данном уровне иерархии наследования мы не можем предоставить ему никакую разумную реализацию. Все неабстрактные классы наследники должны предоставить реализацию этого метода. Наличие абстрактных методов в неабстрактных классе приведет к ошибки компиляции.

Абстрактные классы решают сразу две задачи. С одной стороны, они определяют набор публичных методов, то есть, по сути, внешний контракт данного класса. С другой стороны, они могут содержать непубличные поля и методы, т.е. какие-то детали реализации. Бывает так, что нам нужно описать только внешний контракт программной сущности, не закладывая никаких деталей реализации. Предполагается, что возможные реализации настолько различны, что ни будут иметь ничего общего между собой. Например, сервис, который получает все заказы клиента по его идентификатору может быть реализован поверх реляционной базы данных или поверх XML-файла, содержащего все данные. Или, что удобно в тестах, просто хранить все данные в памяти. Чтобы описать такой сервис мы могли бы завести абстрактный класс, в котором есть только публичные абстрактные методы. Однако у конкретной реализации сервиса по объективным причинам может быть необходимость отнаследоваться от какого-то другого класса. И это проблема, потому что в Java, как мы помним, множественной наследование запрещено.

3.5.2 Интерфейс

К счастью, решение есть. В качестве альтернативы абстрактному классу, в котором все методы публичные и абстрактные, в Java введена отдельная сущность — интерфейс.

|  |  |
| --- | --- |
| 1.  2.  3.  4. | public interface OrderService {  Order[] getOrdersByClient(long clientId);  } |

Все методы интерфейса по определению публичные и абстрактные, поэтому явно писать данные модификаторы не требуются. Поля в интерфейсе могут быть только публичные, статические и финальные, то есть, по сути, константы. Ничего непубличного в интерфейсе объявить нельзя. Интерфейс служит для объявления контракта объекта по отношения к внешнему миру.

Хотя в интерфейсе все же может быть какой-то исполняемый код. Во-первых, могут быть статический методы, обязательно публичные. Во-вторых, могут быть так называемые default‑методы, то есть методы с реализацией по умолчанию. Хотя default‑методы немножко размывают границу между интерфейсами и абстрактными классами, они были добавлены в язык не случайно.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6. | package org.stepic.java.orders;  public interface OrderService {  Order[] getOrdersByClient(long clientId);  } |

Представьте себе, что в программе объявлен интерфейс, который предлагается реализовывать другим программистом.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9. | package org.stepic.java.orders;  public interface OrderService {  Order[] getOrdersByClient(long clientId);  Order[] getOrdersByClient(long clientId,  LocalData data);  } |

Через некоторое время в интерфейс добавляется еще один метод для получения списка заказов по клиенту за конкретную дату, чтобы, по возможности, не загружать в память лишние ненужные данные. Что происходит в этот момент? У всех программистов, реализовавших этот интерфейс, перестанет компилироваться код их классов, так как появился нереализованный абстрактный метод. Чтобы этого не происходило в интерфейсах добавили поддержку default-методов.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14. | package org.stepic.java.orders;  public interface OrderService {  Order[] getOrdersByClient(long clientId);  default Order[] getOrdersByClient (  long clientId, localData date) {  Order[] allOrders =  getOrdersByClient(clientId);  return Orders.filterByDate(  allOrders, date);  }  } |

Разработчик интерфейса при добавлении нового метода может задать реализацию по умолчанию, которая будет использоваться, если в классе, реализующим интерфейс, этот метод не определен. Разумеется, остается возможность эту реализацию переопределить на какую-то более оптимальную.

Каким образом интерфейсы помогают решить проблему множественного наследования? Оказывается, в Java можно объявлять классы, реализующие произвольное количество интерфейсов.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9. | public class OrderServiceImpl  extends ServiceBace  implements OrderService {  public Order[] getOrdersByClient(  long clientId) {  // ...  }  } |

Реализуемые классом интерфейсы перечисляются после ключевого слова implements через запятую. Причем класс может реализовывать интерфейсы в дополнении к наследованию от произвольного класса.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | package java.lang;  public interface CharSequence {    int length();    char charAt(int index);    CharSequence subSequence(int start,  int end);  } |

В стандартной библиотеке Java есть много интерфейсов. Например, CharSequence — последовательность символов, в которой можно узнавать длину, запрашивать символ по индексу или брать подпоследовательность. Реализациями CharSequence являются знакомые нам классы String и StringBuilder.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | package java.lang;  pablic interface Appendable {  Appenfable append(CharSequence csq);  Appendable append (CharSequence csq,  int start, int end);  Appendable append(char c);  } |

Или интерфейс Appendable — нечто, куда можно дописывать символы. Одна из реализации Appendable — тот же StringBuilder. Выделение подобных интерфейсов позволяет авторам стандартной библиотеки, ну нам тоже, писать максимально обобщенный код, допускающий самые разнообразные реализации последовательности символов.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | package java.lang;  @FunctionalInterface  public inntervace Runnable {  void ren();  } |

Есть в стандартной библиотеке и совсем простые интерфейсы. Например, Runnable — абстрактное нечто, которое можно запустить.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | packaje java.lang;  @FunctionalInterface  public interface Comporator<T> {    int comporator(T o1, T o2);  } |

Или Comporator. Он умеет сравнивать пары объектов и говорить какой из них больше. Если первый объект меньше второго, то возвращается отрицательное число. Если объекты равны — возвращается 0. Если первый объект больше второго — возвращается положительное число.

Отличительной особенностью таких интерфейсов является то, что в них ровно один абстрактный метод. Такие интерфейсы называются функциональными. Это можно дополнительно подчеркнуть, поставив аннотацию @FunctionalInterdace. При наличии этой аннотации компилятор проверит правда ли в интерфейсе единственный абстрактный метод и, если нет, то выдаст ошибку.

3.5.3 Функциональный интерфейс (а также лямбда‑выражения и ссылки на методы)

Функциональные интерфейсы обладают одним приятным свойством — их реализацию можно компактно записать в виде лямбда‑выражения или ссылки на метод. Давайте посмотрим это на примере.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13. | package org.stepic.java.timer;  public class Timer {  public long measureTime(  Runnable runnable) {  long startTime =  System.currentTimeMillis();  runnable.run();  return System.currentTimeMillis() -  startTime;  }  } |

Предположим, что нам надо написать программу, которая замеряет время работы различных вычислений. Код замера времени хочется написать один раз. И затем переиспользовать. Поэтому мы напишем такой класс с методом meacureTime(), принимающим измеряемое вычисление в виде экземпляра Runnable. Метод сначала засекает текущее время, затем запускает вычисления, затем подсчитывает и возвращает потраченное количество миллисекунд. Метод 10 возвращает количество миллисекунд, прошедшее с полуночи 1 января 1970 года. Но в данном случае нас интересует только разница.

Хочу сразу предостеречь: делать такие измерения и верить их результатам можно только если вычисление занимает как минимум сотни миллисекунд. Если нужен более чувствительный эксперимент, то для этого потребуется другие специальные инструменты. В аду есть специальный котел для тех, кто при помощи этого метода пытается померить какой оператор интернета быстрее работает — префиксный или постфиксный, ну или другие вещи подобного порядка.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24. | package org.stepic.java.timer;  import java.math.BigDecimal;  public class Main {  public static void main(String[] args) {  Timer timer = new Timer();  long time = timer.measureTime(  new BigDecimalPower());  System.out.println(time);  }  private static class BigDecimalPower  implements Runnable {  @Override  public void run() {  new BigDecimal("1234567").pow(  100000);  }  }  } |

Как воспользоваться этим методом? Давайте пойдем в главный класс нашей программы и посмотрим. Метод meacureTime() принимает экземпляр Runnable или более точно экземпляр класса, реализующий интерфейс Runnable. Вот ровно такой класс нам и пришлось здесь объявить. Он реализует этот интерфейс и в своем методе run() описывает то вычисление, время работы которого мы собираемся измерять. Получилось довольно громоздко. Получается, что на одну строчку с вызовом метода meacureTime() нам нужно где-то в программе потратить еще как минимум 5 строчек на объявления соответствующего класса. Давайте попробуем немножко упростить этот код и сделаем его более компактным. Для этого мы воспользуемся существующей в языке возможностью декларировать классы прямо в теле метода.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22. | package org.stepic.java.timer;  import java.math.BigDecimal;  public class Main {  public static void main(String[] args) {  Timer timer = new Timer();  long time = timer.measureTime(  new Runnable() {  @Override  public void run() {  new BigDecimal(  "1234567").pow(  100000);  }  });  System.out.println(time);  }  } |

Вот так мы перенесли объявление класса прямо внутрь метода main() и более того прямо внутрь выражения с вызовом метода measureTime(). Здесь объявляется анонимный класс, реализующий интерфейс Runnable и сразу же создается его экземпляр. Этот класс анонимный, то есть он не имеет своего имени. Компилятор при компиляции программы сам на свое усмотрение сочинит для этого класса какое-то уникальное имя. Этот код уже немного лучше. Здесь меньше букв и второе его преимущество в том, что сразу в вызове measureTime() видно, что именно мы именно мы собираемся измерять. То есть не нужно прыгать по программе в поисках кода 15‑17. Но это все еще довольно громоздко. До Java 7 включительно с этим было ничего не поделать. Но в восьмерке была добавлена поддержка лямбда-выражений, о которых Java программисты мечтали много лет. Теперь этот же код можно переписать более компактно. И среда разработки нам в этом поможет (указатель на Runnable() (11 строка) и Alt+Enter).

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15. | package org.stepic.java.timer;  import java.math.BigDecimal;  public class Main {  public static void main(String[] args) {  Timer timer = new Timer();  long time = timer.measureTime(  () -> new BigDecimal(  "1234567").pow(100000));  System.out.println(time);  }  } |

11‑12 — это лямбда-выражение. Оно начинаются с круглых скобок, в которых указываются параметры. В данном случае метод run() не предполагает никаких параметров, поэтому скобки пустые. Кстати, параметры здесь указываются без типов, потому что типы известны компилятору из объявления интерфейса. После скобок идет стрелочка, а за ней тело лямбда-выражения. В данном случае оно у нас состоит из единственного выражения.

|  |  |
| --- | --- |
| 1.  2.  3. | long time = timer.measureTime(() -> {  new BigDecimal("1234567").pow(100000)  }); |

Если бы здесь было что-то более сложное, это надо было заключить в фигурные скобки. Такое лямбда‑выражение порождает объект, реализующий интерфейс Runnable с указанным телом метода run(). Лямбдой можно заменить только реализацию функционального интерфейса, то есть интерфейса с единственным абстрактным методом. Это может быть как интерфейс со стандартной библиотеки, так и ваш собственный. И еще одна возможность доступная для функциональных интерфейсов — это получение экземпляра интерфейса при помощи ссылки на метод с подходящей сигнатурой. Давайте я немного порефакторю этот код с помощью среды разработки и покажу как это выглядит (выделяем 2 строчку и Ctrl+Alt+M, bigDecimalPower, Alt+Enter).

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19. | package org.stepic.java.timer;  import java.math.BigDecimal;  public class Main {  public static void main(String[] args) {  Timer timer = new Timer();  long time = timer.measureTime(  Main::bigDecimalPower);  System.out.println(time);  }  private static void bigDecimalPower() {  new BigDecimal("1234567").pow(  100000);  }  } |

11 — это выражение представят собой ссылку на метод. Ссылка на метод состоит из имени класса или переменной, 2‑х двоеточий и имени метода. Значением такого выражения является объект, реализующий требуемый функциональный интерфейс и вызывающий код 16‑17 при вызове своего главного метода.

Это было первое и довольно краткое знакомство с синтаксисом лямбда-выражений и ссылок на методы. Более подробно мы будем о них говорить в модуле, посвященным коллекциям. И тогда они раскроют свой истинный потенциал.

4 Обработка ошибок, исключения, отладка

4.1 Знакомство с исключениями

4.1.1 Способы возврата ошибки

В любой программе может случиться ошибка. Для этого есть 1000 и 1 причина. Может быть пользователь ввел неправильные данные, может быть отвалилась необходимая в программе база данных или другой важный сетевой ресурс, может быть сбоит чип оперативной памяти и случайным образом портит данные в нем хранящиеся. Причины могут быть разные, но непреложный факт состоит в том, что от всех ошибок не застраховаться, а, следовательно, нашим программам приходится иногда с ними сталкиваться и как-то на них реагировать.

В разных языках существует разная культура обработки ошибок, определяемая как возможностями языка платформы, так и исторически сложившимися традициями. Чтобы дальнейший рассказ не был слишком теоретическим, давайте рассмотрим такой пример.

|  |  |
| --- | --- |
| 1.  2.  3.  4. | public interface Calculator {  double calculate(String expr);  } |

Предположим, вы пишите подключаемую в разные программы библиотеку, вычисляющую математические выражения. Лицо этой библиотеки интерфейс Calculator и какая‑то его реализация. В интерфейсе есть метод, принимающий строку с выражением и возвращающий обратно double, вычисленное значение выражения. И вот в выражении, поданным на вход содержится ошибка. Например, в нем встречается число, где в качестве десятичного разделителя используется запятая вместо точки. Как Calculator на это реагировать?

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | public class CalculatorImpl  inplements Calculator {  @Override  public double calculate(String expr) {  // ...  System.exit(1);  // ...  }  } |

Самый простой по реализации, но и самый плохой по последствиям способ — это вызов System.exit(), то есть программа немедленно завершится, упадет, по сути, потеряв все накопленные в памяти данные. Раз и все: нет ни программы, ни данных! Я имею в виду всю большую программу, включающую Calculator в качестве одного из своих компонентов.

Недостатки такого поведения понятны. Программа рухнула, пользователю придется ее перезапускать и начинать все что он делал сначала. Хорошо еще, если программа сообщает в чем причина ошибки. А то как пользователь догадается, что на самом деле программа упала потому, что неделю назад он ввел где-то число в неправильном формате. В общем, System.exit() это не самое дружелюбное поведение со стороны нашей библиотеки Calculatorа.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | public class CalculatorImpl  inplements Calculator {  @Override  public double calculate(String expr) {  // ...  return Double.NaN;  // ...  }  } |

Есть способ немного получше. В случае ошибки в выражении наша библиотека будет возвращать в качестве результата какое-то специальное значение, например, NaN, по которому вызывающая сторона поймет, что случилась ошибка. Но нужно, чтобы у возвращаемого типа было свободное значение, которое можно зарезервировать под ошибку. Это не всегда возможно. Например, генератор случайных чисел может сгенерировать абсолютно любое число. Где нам взять отдельное значение под ошибку? А если видов ошибок несколько и их надо различать, то потребуется несколько значений.

Да и нашем случае NaN может совершенно законно получиться при некоторых вычислениях. Например, при делении 0 на 0 или при умножении 0 на бесконечность. В общем, как обозначить ошибку не понятно. А даже если мы бы выделили какое-то специальное значение, то вызывающий код некрасиво оброс бы if для своевременного отлова этой ошибки.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | public class CalculatorImpl  inplements Calculator {  @Override  public Result calculate(String expr) {  // ...  return Recult.error();  // ...  }  } |

Чтобы обойти проблему с резервированием значения под ошибку в возвращаемом типе, можно пойти дальше и у метода завести выходной параметр специально под флаг ошибки. Или возвращать из метода не double, а простенький объект‑пару, состоящую из числа результата и признака ошибок, примерно так:

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | class Recult {  double value;  boolean isEror;  } |

Тогда весь диапазон типа результата будет использоваться по своему назначению. Правда if для отслеживания ошибки в вызывающем коде никуда не денутся.

Есть еще вариант с заведением под признак ошибки отдельного поля или метода в самом классе. То есть после вызова метода класса calculate() вызывающий код должен будет дернуть еще один метод Calculator, чтобы узнать а можно ли верить возвращенному только что результату. Очевидно, это еще более усложняет вызывающий код, а также ставит крест на использовании одного экземпляра Calculator из нескольких потоков.

4.1.2 Знакомство с исключениями

Все эти способы уведомления частей программы о случившихся ошибках имеют серьезные недостатки. К счастью, есть решение, которое их лишено. И оно состоит в использовании исключений.

Исключение — это событие, которое случается в процессе работы программы и прерывает стандартный ход его исполнения. Программа переходит в специальный режим поиска обработчика внештатной ситуации. Такой обработчик может предпринять какие‑то действия и вернуть программу в штатный режим работы. Ну а если обработчика не нашлось, то JVM все-таки завершит программу, но обеспечит достаточно подробный диагностический вывод, по которому если не пользователь, то хотя бы программист поймет куда копать.

Исключения поддержаны на уровне языка Java, то есть ключевых слов и синтаксических конструкций. Кроме того, они пользуются на уровне виртуальной машины, которая сообщает о многих своих проблемах, бросая исключения. Ну и, разумеется, вся стандартная библиотека тоже активно использует исключения.

Приведу несколько самых популярных исключений. Какие-то из них вы наверняка уже получали, когда писали свои первые программы.

|  |  |
| --- | --- |
| 1.  2.  3.  4. | Object nullRef = null;  // java.lang.NullPointerException  nullRef.toString(); |

3 или Java-программисты зовут его просто NPE. Это исключение выбрасывается JVM, когда идет обращение по нулевой ссылке. Другими словами, есть переменная ссылочного типа, куда положили null. Потом пытаются по этой ссылке вызвать метод или обратиться к полю. В ответ виртуальная машина кинет такое исключение, которое при отсутствии обработчика приведет к аварийной остановке и распечатке в консоли информации об исключении.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9. | java.lang.NullPointerExceptoin  at.org.stepic.java.exception.Test.baz(  Test.java:19)  at.org.stepic.java.exception.Test.bar(  Test.java:14)  at.org.stepic.java.exception.Test.foo(  Test.java:10)  at.org.stepic.java.exception.Test.main(  Test.java:6) |

Распечатано будет следящее: тип исключения, возможно, какое-то сообщение с деталями и cтек-трейс. Вся цепочка вызовов, начиная от точки входа в программу и заканчивая методом, где, собственно, случилась ошибка. Здесь же для каждого метода видно имя исходного файла и строку в нем.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | int[] array = {1, 2, 3};  Array[10];  // java.lang.ArrayIndexOutOfBoundsException  // StringIndexOutOfBoundsException |

Другое исключение 4 случается если попытаться обратиться к несуществующему элементу массива по отрицательному числу индекса или по индексу большему или равному длине массива. Аналогично исключение 5 случиться, если обраться к несуществующему символу в строке.

|  |  |
| --- | --- |
| 1.  2.  3.  4. | new FileInputStream("not\_existing\_file");  // java.io.FileNotFoundException  // java.io.IOException |

Исключение 3 случится при попытке открытия несуществующего файла. Оно является подклассом более общего 4, экземпляры которого бросаются при самых разных ошибках ввода/вывода.

|  |  |
| --- | --- |
| 1. | // OutOfMemoryError |

Исключениями представлены и разные ошибки самой виртуальной машины. Например, когда JVM не хватает памяти на создание нового объекта, она бросает исключение типа 1, которое можно даже попытаться обработать. Другое дело, что обычно делать этого не стоит, потому что в этой ситуации предпринять что-то разумное довольно проблематично.

4.1.3 Класс Throwable

Исключения в Java — это объект, экземпляр какого-то класса.

|  |  |
| --- | --- |
| 1. | // java.lang.Throwble |

Все классы исключений наследуются от 1, который в свою очередь наследуется от Object. Ключевое свойство экземпляров Throwble и его подклассов возможность быть брошенным.

|  |  |
| --- | --- |
| 1.  2.  3. | throw new IllegalStateException(  "Invalid user. " +  "Please replace user and continue."); |

Бросание выглядит следующим образом: ключевое слово throw и за ним любое выражение типа Throwble. Обычно здесь просто создается экземпляр нового исключения.

Важный момент! Стек-трейс, который мы увидим в консоли при печати такого исключения, определяется тем, где был создан экземпляр исключения, а не тем откуда оно было выброшено. У класса исключения традиционно есть конструктор, принимающий строку сообщения. Это сообщение, поясняющее проблему, будет напечатано вместе со стек‑трейсом.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18. | package java.lang;  public class Throwable {  public String getMessage() { /\*...\*/ }  public void printStackTrace() { /\*...\*/ }  public StackTraceElement[]  getStackTrace() { /\*...\*/ }  public Trowable getCause() { /\*...\*/ }  public Trowable[] getSuppressed() {  /\*...\*/ }  // ...  } |

В классе Throwable объявлены различные методы. Перечислю некоторые из них.

getMessate() возвращает текстовую строку с описанием проблемы. Эта та самая строка, в которой можно было передать в конструктор. Ее задача упростить диагностику проблемы, объяснить пользователю что именно произошло. Хорошая идея не просто написать все сломалось, а включить подробности. Например, текущее состояние объекта и параметры метода, где все произошло. В случае с Calculatorом, которому пришла некорректная срока с выражением, можно добавить сюда часть выражения, с которой возникли проблемы при разборе.

printStackTrace() печатает стек‑трейс в стандартный поток ошибок. Но если вы пишите обработчик исключения, который только и делает что печатает стек‑трейс и завершает программу, то такой обработчик можно вообще не писать, так как именно это JVM сделает и сама, если исключение не найдя обработчика вылетит из метода main().

getStackTrace() возвращает в стек‑трейс то же, что печатается в консоли в виде массива Java объектов StackTraceElement[], каждый из которых содержит имя класса, имя метода, имя файла и номер строки. То есть стек-трейс можно получить и проанализировать. Хотя потребность такая возникает редко.

Дальше есть метод getCause(). Он возвращает другое исключение, явившиеся причинной данного. Или null, если данное исключение является первичным и не вызвано каким-то другим. Если дойдет дело до печати сообщения в консоль, то будет распечатана вся цепочка исключений. cause(), как и message(), может быть задан при создании экземпляра исключения. Там есть соответствующий перегруженный конструктор.

И еще есть getSuppressed(). Этот метод возвращает одно или более исключений, которое мы заглушили в процессе обработки основного исключения. Поясню на примере. Представим, что мы читаем их файла. И вдруг случилось ошибка ввода/вывода, вылетело исключение. Мы начали ее обрабатывать, попытались закрыть файл, но ту бац — случилась еще одно исключение, потому что файл совсем плох и даже закрыть его не получается. И того у нас на руках два исключения. Что с ними делать? Как их обрабатывать? Существует такое соображение, что первое исключение обычно более важно. Именно оно описывает первопричину проблемы, а все остальное, что случилось потом с высокой степенью вероятности это последствия, наведенка. Поэтому первое исключение точно терять не следует. С другой стороны, последующие исключения могли быть совсем про другое и не связанны с первым. Поэтому совсем игнорировать их тоже не следует. Именно для этого и существует массив Suppressed исключений. Одно исключение выбирается в качестве главного, а остальные под него подвешиваются. И все это в таком виде бросается. Далее это попадает к вызывающему коду, и он пусть разбирается что с этим делать. Ну а если дело дойдет до печати ошибки в консоль, то Suppressed исключения тоже будут распечатаны.

4.1.4 Типы исключений

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | // java.lang.Error;  // java.lang.Exception;  // java.lang.RuntimeException; |

Исключения в Java делятся на 3 группы. Это 1, 3 и 5. Если нарисовать как они наследуются друг от друга, то получится следующая картинка:

|  |  |  |
| --- | --- | --- |
|  | Т |  |
|  |  |  |
| Er |  | Ex |
|  |  |  |
|  |  | REx |

Error и подклассы — это ошибки виртуальной машины.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | // java.lang.Error;  // java.lang.Exception;  // java.lang.RuntimeException; |

Например, 1 означает что виртуальной машине не хватило памяти; 3 означает что виртуальная машина не смогла найти какой-то из классов, на который ссылается программа; 5 означает что класс найден, но там что-то не так с байт-кодом.

JVM таким образом сообщает, что ей стало плохо. Хотя и не настолько, чтобы сразу умереть. Общее у этих ошибок то, что их не надо пытаться обрабатывать. Помочь виртуальной машине вы скорее всего уже не сможете.

Exception и RuntimeException соответствуют ошибкам уровню программы: некорректные параметры, некорректное состояние, недопустимое действие. Ошибки такого сорта можно уже пытаться как-то обрабатывать. Разница между ними состоит в том, что Exception и его подклассы, кроме RuntaimeException, являются проверяемыми (checked), а RuntimeException и его подклассы — непроверяемыми (unchecked).

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12. | import java.io.Exception;  public class ExeptionDemo {  public void someMethod() {  // ...  throw new IOEception(  "Failed to read file");  // ...  }  } |

Проверяемые исключения являются предметом особого контроля компилятора. Он внимательно следит за тем, чтобы такие исключения декларировались и обрабатывались. Характерный пример проверяемого исключения IOException — ошибка ввода/вывода. Если попытаться бросить такое исключение, то код не скомпилируется. Компилятор будет требовать, чтобы либо в этом же методе был обработчик этого исключения, либо на методе было явно написано, что из него может вылететь IOException. Написать это можно следующим образом:

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10. | import java.io.Exception;  public class ExeptionDemo {  public void someMethod()  throws IOExeption {  // ...  }  } |

Если мы напишем это на методе, то аналогичная проверка будет выполнена компилятором и во всех местах вызова нашего метода. То есть либо IOEception должен быть сразу обработан, либо возможность его вылета обозначена в объявлениях всех методов, вызывающих наш метод. Таким образом компилятор обеспечивает, что в сигнатуре метода видны все проверяемые исключения, которые могут из него вылетать. Их может быть несколько, перечисленных через запятую.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | // java.lang.NullPointerException;  // java.lang.ArrayIndexOutOfBoundException;  // java.lang.ArtithmeticException; |

Непроверяемые исключения такого внимания компилятора не получают. Их можно свободно бросать из любой точки программы, не декларируя. Примеры непроверяемых исключений это 1, 3 и 5, вылетающий при делении на 0. Такие ошибки могут случиться в любой точке программы и было бы крайне неудобно, если бы компилятор требовал их везде явно декларировать или обрабатывать.

4.1.5 Пример собственного типа исключения

Давайте посмотрим, как применить исключения к нашему классу Calclulatorу. Заведем специальный тип исключения, который будет соответствовать ошибкам вычисления выражения. Отдельный тип исключения позволит вызывающему коду четко отличать ошибки вычислений выражения от других ошибок в программе.

Исключения декларируются как обычный класс, прямо или косвенно наследующий Excception или RuntimeException в зависимости от того, хотим мы делать его проверяемым или нет.

Делать исключение проверяемым или нет дело вкуса и традиции конкретной группы программистов. Есть лагерь, выступающий за проверяемые исключения и за дополнительный контроль со стороны компилятора. Другой лагерь говорит, что остальные популярные современные языки программирования отказались от проверяемых исключений, потому что это не лучшая идея. Они загромождают и усложняют код. А плохой и ненадежный код можно написать несмотря ни на какие проверки компилятора.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14. | public class CalclulatorException  extends RuntimeException {  public CalclulatorException(  String message) {  super(message);  }  public CalclulatorException(  String message,  Throwable cause) {  super(message, cause);  }  } |

Давайте сделаем наше исключение непроверяемым. Для исключения обычно заводят конструкторы, принимающие message и cause. Можно добавить полей и методы, если они требуются.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12. | public class CalclulatorImpl  implements Calclulator {  @Override  public double calclulate(String expr) {  // ...  throw new CalculatorException(  "Unsupported operator found";  // ...  }  } |

Теперь в ответ на некорректное выражение Calclulator будет бросать исключение. Поскольку оно непроверяемое, то никаких дополнительных слов в программе писать не требуется. А вот если бы мы сделали его проверяемым, то надо было бы задекорировать его на уровне метода и более того в интерфейсе Calclulator.

Дело в том, что при наследовании или реализации интерфейса нельзя декларировать переопределенному методу больше проверяемых исключений, чем указано у базового метода. Это нарушало бы контракт базового класса. Зато контракт можно менять в другую сторону: бросать в переопределенном методе меньше исключений, чем задекларировано у родителя.

Кстати, точка входа в программу, метод main(), может декларировать и бросать любые проверяемые исключения. Все что вылетит из него приведет к распечатке стек-трейса и остановке программ.

4.2 Обработка исключений. Try-catch

4.2.1 Ловим исключение

Мы с вами обсудили что такое исключения и как их бросать. Давайте теперь научимся их обрабатывать. В данном примере мы в цикле читаем вводимые пользователем выражение и вычисляем их. В случае ошибки в выражении будем вежливо просить пользователя повторить ввод.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20. | for (;;) {  System.out.println("Enter expression: ");  String expr = readUserInput();  if(expr == null ||  "exit".equalsIgnoreCase(expr)) {  break;  }  try {  double result =  calculator.calculatore(expr);  System.out.println(  "Result: " + result);  } catch (CalculatorException e) {  System.out.println(  "Bad expression: " +  e.getMessage());  System.out.print(  "Please try again: ");  }  } |

Для обработки исключений в языке есть конструкция try‑catch. Внутри try — блок кода, где могут влучаться исключения. А далее один или более блок catch с классом исключения, именем переменной, куда мы сложим это исключение на время обработки, и, собственно, кодом обработчика. Чтобы не пугать пользователя стек‑трейсами, будем показывать ему только текстовое сообщение из исключения. Внутри catch‑блока видны те же идентификаторы, что и перед блоком try. А все что объявлено внутри блока try, в catch-блоке уже не доступно.

Если за время работы try-блока исключение не вылетело, то блоки catch не исполняются. Управление передается на код после них. Если исключение случилось и выполнился catch‑блок, то после него управление тоже передается туда же. Блоков catch у одного try может быть несколько. Они просматриваются в порядке следования и выбирается первый с подходящим типом исключения. Проверка делается как на instanceof, то есть с точностью до подкласса.

|  |  |
| --- | --- |
| 1.  2.  3. | // catch (Exception e)  // catch (Trowable t) |

Например, 1 означает обработку любого типа исключения, проверяемого или непроверяемого, кроме ошибок JVM. А 3 поймает вообще все, включая ошибки виртуальной машины. Выполнится может только один, первый подходящий блок catch. Другие уже не будут выполняться.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15. | try {  // ...  } catch (FirstException e) {  e.printStackTrace();  } catch (SecondException e) {  e.printStackTrace();  }  // since Java 7 can be replace with:  try {  // ...  } catch (FirstException e |  SecondException e) {  e.printStackTrace();  } |

Начиная с Java 7 стало возможно в одном catch-блоке обрабатываться несколько типов исключений. Это удобно, когда исключения не являются подклассами друг друга, но тем не менее должны обрабатываться одинаково.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6. | InputStream is = new FileInputSream("a.txt");  try {  readFromInputStream(is);  } finally {  is.close();  } |

После try может быть еще один блок finally. Он выполняется в любом случае, было исключение или нет, исполнился какой-то catch или нет. Finally исполняется последним. В этом блоке обычно занимаются высвобождением ресурсов, закрытием файла, снятием блокировок. finally не является обработчиком исключения. То есть если из блока try вылетело исключение, то после обработки блока finally это исключение полетит дальше.

Это характерный часто встречающийся пример использования блока try-catch. Но у него есть один недостаток. Если в блоке try случилось исключение, и исключение также случилось при вызове close(), то дальше из этого блока полетит исключение, брошенное из close(), а исходное будет потеряно.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10. | InputStream is = new FileInputSream("a.txt");  try {  readFromInputStream(is);  } finally {  try {  is.close();  } catch (IOException e) {  // ignore  }  } |

Поэтому можно встретить еще такой паттерн. Исключения из close() здесь явно отлавливается и игнорируется, чтобы не перебивать исходную ошибку. Это немного лучше, но тоже не очень красиво.

Из-за того, что всем Java‑разработчикам приходилось как-то хитрить и некрасиво выкручиваться из этой ситуации с исключениями в finally, в Java 7 была добавлена новая разновидность try — try с ресурсами. Выглядит эта конструкция следующим образом.

|  |  |
| --- | --- |
| 1.  2.  3.  4. | try (InputStream is =  new FileInputSream("a.txt")) {  readFromInputStream(is);  } |

После ключевого слова try в круглых скобках выделяется ресурсы, с которыми будет работать этот блок. Их может быть несколько. В таком случае они перечисляются через точку запятой. Гарантируется, что при выходе из блока все ресурсы будут освобождены. Точнее, что на каждом из ресурсов будет вызван метод close(). Аналогично блоку finally это случится при любом раскладе, было исключение или не было. Но в отличие от того примера блока finaly, который мы видели на предыдущих слайдах, здесь исключение из close() не перебьет собой исходные исключения, а будет добавлено в него в качестве заглушенного Suppressed.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12. | InputStream is = new FileInputSream("a.txt");  try {  readFromInputStream(is);  } catch (Trowable t) {  try {  is.close();  } catch (Trowable t2) {  t.addSupressed(t2);  }  throw t;  }  is.close(); |

То, что реально происходит в блоке try с ресурсами можно иллюстрировать следующим фрагментом кода. Он может не передавать все нюансы, но демонстрирует идею.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | package java.lang;  public interface AutoCloseable {  void close() throws Exception;  }  // java.lang.AutoCloseable |

Кстати, ресурс — это любой объект, реализующий интерфейс 7. В этом интерфейсе всего один метод close(). Это тот самый метод, который надо вызвать для освобождения ресурса. Можно свободно реализовать этот интерфейс в своих собственных классах. И тогда появится возможность использовать их в блоке try с ресурсами.

4.2.2 Обрабатываем исключение

Обсудим теперь что можно делать в блоке catch в ответ на пойманные исключения. Во‑первых, нужно ответить себе на вопрос: обладаем ли мы в данном месте программы информацией, необходимой для принятия решения о том, как нужно реагировать на исключения. Вполне возможно, что нет. В таком случае правильным решением будет пробросить это же или другое исключение дальше вызывающему коду. Если мы решаем пробросить то же самое исключение, то и блок try‑catch в этом месте программы вообще не нужен. Пусть себе исключение спокойно летит до того уровня, где оно может быть нормально обработано.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13. | try {  // code throwing MyException;  } catch (MyException e) {  // ???  }  // Double.parseDouble()  // NumberFormatException  // CalculatorException  // throw new CalculatorException(e) |

Рассмотрим такой пример. Наша библиотека Calculator может в своей реализации использовать метод 7 для разбора чисел из входного выражения. Если число оказалось в некорректном формате, то 7 бросает исключение 9. Это означает ошибку в выражении, которую нам надо сообщить коду, вызывающему Calculator. По договоренности Calculator при проблемах с выражением должен бросать 11, поэтому в нашем примере обработка 9 будет заключаться в заворачивании этого исключения в 11 и выбросе этого нового исключения 13. Зачем класть внутрь нового исключения исходное исключение? Для того, чтобы при печати такого исключения было максимум информации для разбора.

Итак, исключение долетело до того места программы, где действительно достаточно информации для принятия решения об обработке. Мы можем решить, что исключение фатальное, и немедленно завершить программу. Или, наоборот, мы можем решить, что исключение безвредное, на работоспособность программы оно не повлияет, можно продолжать. Или как в примере с интерактивным Calculator можем попросить пользователя ввести другое выражение.

Правильная обработка зависит от конкретной задачи. Но в любом случае рекомендуется залогировать полную ошибку со всем стрек-трейсом, чтобы упростить себе в будущем разбор этой проблемы, если что-то пойдет не так.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | try {  // code throwing MyException;  } catch (MyException e) {  // ???  }  // e.printStackTrace() |

В простых маленьких программах можно просто распечатать ошибку в консоли, вызвав метод 7. Но лучше воспользоваться каким-то специальным средством для логирования. Это мы обсудим в следующем уроке.

4.2.3 Рекомендации по использованию исключений

Несколько рекомендаций по правильному использованию исключений. Когда вы пишите какой-то метод в вашем классе, то подумайте о том какие гарантии вы можете дать в вызывающем у вас коду. Можете ли вы дать сильные гарантии, то есть обещать, что несмотря на выброс исключения, объект останется в исходном корректном состоянии, в нем ничего не сломается и этим объектом можно пользоваться дальше. Или вы сможете дать слабые гарантии, то есть если состояние объекта и изменилось, то оно осталось корректным, и объектом можно пользоваться дальше. Возможно, вы гарантируете только то, что несмотря на исключения не случилось утечек ресурсов, а объект к использованию уже не пригоден. Ну и в худшем случае, вы не можете гарантировать ничего из вышеперечисленного.

Конечно, чем более сильные гарантии вы можете обеспечить, тем удобнее другим программистам будет пользоваться вашим классом. К этому стоит стремится. Гарантии отсутствие утечек можно обеспечить правильным использованием try‑finaly и try с ресурсами. А чтобы объект не переходил в некорректное состояние, можно, например, сначала тщательно валидировать все входные параметры метода. Или если у метода делается какое-то многоэтапное вычисление, то промежуточные результаты этих вычислений сохранять сначала в локальных переменных. И только когда все вычисление успешно отработало, то менять состояние объекта, присваивая новые значения его полям.

И второе замечание. Механизм обработки исключений try‑catch следует применять для действительно исключительных ситуаций. Не надо пытаться реализовать обычную условную логику при помощи try-catch. Во‑первых, это менее эффективно. Порождение и обработки исключений несут ненулевые накладные расходы. Во‑вторых, можно случайно обработать не ту ошибку, которую вы хотели, если в try‑блоке происходит много всего.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | String string;  try {  string = object.toString();  } catch (NullPointerException e) {  string = “null”;  }  System.out.println(string); |

Это плохой пример. Так делать не надо. Вместо этого надо использовать обычный условный оператор.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | String string = object == null ? “null” :  object.toString();  Systen.out.println(string);  // Objects.toString(object)  // java.util |

Кстати, скажу по секрету, есть утилитный метод 6, который реализует логику 1. Поэтому здесь можно использовать его. Класс Objects находится в пакете 8.

4.3 Логирование

4.3.1 Класс Logger

В предыдущем уроке, обсуждая обработку исключений, мы с вами говорили о том, что полезно сохранить куда-то исключения с полным стек-трейсом для последующего разбора. Это не единственный случай, когда нам может захотеться из программы сохранить какое‑то диагностическое сообщение на будущее. Например, при отладке какого-то сложного алгоритма мы можем захотеть увидеть подробный протокол работы программы вплоть до того, что вызвался такой-то метод с такими-то параметрами или в такую-то переменную записано такое-то значение. Или вот еще пример. Программа сервер может выводить диагностическое сообщение каждый раз, когда ей приходит клиентский запрос. Так мы можем собирать статистику о том, какие клиенты, сколько раз, с каких адресов присылали нам запросы и сколько миллисекунд заняла обработка каждого запроса.

Такие вещи можно, конечно, писать в стд аут или в стд эрр, но есть способ лучше. В стандартной библиотеке Java есть набор классов для организации логирования, и, воспользовавшись ими, мы автоматически получаем следующие бонусы. Во‑первых, точечную настройку логирования: включение и отключении на уровне отдельных пакетов или классов. Во‑вторых, настройку уровней детальности логирования. В‑третьих, настройка места, куда логи будут писаться. И, в‑четвертых, настройка форматов, в котором эти логи будут ввестись. Причем это может удобно настраиваться в конфигурационном файле, то есть запустили программу с одними параметрами, получили полный отладочный протокол всего, что внутри программы происходило. Запустили программу с другими параметрами, получили режим промышленной эксплуатации, в которой программа залогировала в начале какую-то важную информацию, например, параметры, с которыми она запущена. А потом она не будет писать ничего лишнего пока все идет нормально. Для того, чтобы получит такое гибкое логирование в программе даже не нужен ни один if. Достаточно правильно воспользоваться теми средствами, которая предоставляет нам стандартная библиотека.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24. | package org.stepic.java.logging;  import java.util.logging.\*;  public class LogDemo {  private static final Logger LOGGER =  Logger.getLogger(  LogDemo.class.getName());  // ...  }  // java.util.logging  // org.stepic.java.logging.LogDemo  // org.stepic.java.logging  // org.stepic.java  // org.stepic  //org |

Эти замечательные классы лежат в пакете 14. Основной класс Logger. Именно через него наша программа будет писать логи. Logger можно получить, вызвав статический метод getLogger(). Этот метод возвращает нам Logger с указанным именем. Обычная практика в каждом классе, где требуется логирование, завести финальное статическое поле, куда положить Logger с именем, соответствующим полному имени класса. В данном примере у нас получиться логгер с именем 16. Такая практика с именование логгеров в честь классов программы не случайна. Во‑первых, каждый класс получает свой индивидуальный логгер и появляется возможность управлять настройками логирования с точностью до класса. Для одного класса логирование включено, для соседнего выключено. Во‑вторых, благодаря длинным именам, разделенным точками, логгеры выстраиваются в иерархии в соответствии с именами. В данном примере у нас есть логгер класс LogDemo с соответствующим именем, а также логгеры родители с именами 18, 20, 22, 24 и самый верхний корневой логгер с именем пустой строкой. Обработка логируемых сообщений идет от конкретного логгера, через который сообщение вброшено на вверх, к корню. И на каждом уровне может быть какая-то своя обработка. Например, на уровне 22 можно организовать вывод всех событий в какой-то специальный лог файл. На корневом уровне можно только выводить в консоль самые критические ошибки.

4.3.2 Вывод сообщений в лог

Итак, мы получили логгер для текущего класса. Давайте залогируем через него какое-либо сообщение. Это делается следующим образом.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | LOGGER.log(Level.INFO, "I'm logging");  // SERVER, WARNING, INFO,  // CONFIG, FINE, FINER, FINEST  LOGGER.warning("We have a problem!");  // setLevel(Level.WARNING) |

У логгера есть метод с говорящим названием log. Первым параметром этот метод принимает Level — уровень логирования. Это степень серьезности данного сообщения. INFO — это информационное сообщение о нормальном ходе исполнения программы. Есть 2 более серьёзных уровня и 4 менее серьезных. На уровне SERVER логируются серьёзные ошибки программы. Это когда совсем ужас‑ужас. На уровне WARNING логируются предупреждения. CONFIG предназначен для логирования каких‑то конфигурационных параметров. А FINE, FINER и FINEST для совсем детального логирования что программа делает вплоть до входа и выхода в методы и прочих мелочей. Для удобства на каждый уровень логирования в логгере есть одноименный метод. Он вбрасывает сообщения соответствующего уровня. Логгер можно сконфигурировать таким образом, чтобы он игнорировал все сообщения с уровнем ниже заданного. Например, логгер, настроенный на уровень warning, будет принимать только сообщения только с уровнями SEVERE и WARNING и тихонько отправлять в дев нул все остальное. Настройка делается либо в коде при помощи вызова метода 8, либо в конфигурационном файле.

Часто нужно залогировать не просто статическую строчку, а добавить туда какую-то динамическую информацию. Например, значение параметров метода или текущее состояние объекта. Это можно сделать 2 способами.

|  |  |
| --- | --- |
| 1.  2. | LOGGER.log(Level.FINEST,  “Current value of x is” + x); |

Первый очевидный способ конкатенировать все что нам нужно в строку и передать результат в метод log(). У этого подхода есть один недостаток, проявляющийся в программах, работающих под большой нагрузкой. Даже если логирование данного уровня отключено, все равно программа постоянно будет заниматься конкатенацией строк и тратить на это заметное время. Поэтому был придуман другой способ.

|  |  |
| --- | --- |
| 1.  2. | LOGGER.log(Level.FINEST,  “Current value of x is {0}”, x); |

Передавать в метод log() фиксированную строку, в которой специальным образом обозначены места {0} для подстановки. И отдельно передавать значения x, которые будут туда подставляться. Если значение одно, то оно передается непосредственно.

|  |  |
| --- | --- |
| 1.  2.  3. | LOGGER.log(Level.FINEST,  “Point coordimates are ({0}, {1})”,  new Object[] {x, y}); |

А если их несколько, то нужно завернуть их в массив Object. Подстановка параметров в строку происходит внутри логгера и только при условии, что указанный уровень логирования включен, то есть строчка действительно попадет лог и не будет выкинута. Отдельно поддержан случай, когда требуется получить в логер распечатку стек-трейс исключений.

|  |  |
| --- | --- |
| 1.  2. | LOGGER.log(Level.SERVER,  “Unexpected exception”, e); |

Для этого достаточно передать в лог исключение 3-тьим параметром. Символы подстановки в сообщении при этом не нужны.

Кстати, тот факт, что в логгер передаются только константные строки с местами для подстановки, позволяет делать локализацию, то есть перевод логируемых сообщений на любой язык. При этом логгер внутри себя будет использовать строчку как ключ для поиска перевода в словаре. Разумеется, такой словарь вы должны будете предоставит. Java не настолько сурова, чтобы делать автоматически машинный перевод любого текста на любой язык. Если перевод не будет найден, то переданная строчка используется как есть.

4.3.3 Тонкая настройка: Handler, Formatter

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | // java.util.logging.Handler  // java.util.logging.ConsoleHandler  // java.util.logging.FileHandler  // java.util.logging.SocketHandler  // .addHandler(...) |

Если копнуть чуть глубже в реализацию логирования, то мы увидим, что Logger не сам решает, как именно он должен обрабатывать пришедшее сообщение. Он делегирует это решение другому объекту 1. Handler — это абстрактный класс, у которого в стандартной библиотеке есть три конкретных наследника. Это 3, выводящий сообщение в консоль, 4, логирующий файл, и 5, отправляющий лог сообщения по сети. Handler задается в конфигурации или прикрепляется к Logger вызовом метода логгера 7.

|  |  |
| --- | --- |
| 1.  2.  3.  4. | java.util.logging.Formatter  java.util.logging.SimlpeFormatter  java.util.logging.XMLFormater |

И еще один аспект — в каком формате сообщение записывается в лог. За это отвечает еще один объект 1. Handler имеет ссылку на Formatter и делегирует ему всю работу по превращению залогированного сообщения в окончательный вид, пригодный к выводу в файл или передачи по сети. В стандартной библиотеке есть 2 реализации: 3, выводящий все в более-менее человекочитаемом текстовом виде, и 4, производящий машиночитаемый лог в формате XML. Комбинируя Logger, Handler и Formatter можно гибко настраивать что куда и как программа будет писать. К тому же можно реализовать свой собственный Handler или Formatter, если стандартные не походят.

4.3.4 Пример использования

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28.  29.  30.  31.  32.  33.  34.  35.  36.  37.  38.  39. | package org.stepic.java.logging;  import ...  public class LogDemo {  private static final Logger LOGGER =  Logger.getLogger(  LogDemo.class.getName());  public static void main(String[] args) {  LOGGER.log(Level.FINE,  "Started with argument: {0}",  Arrays.toString(args));  try {  randomlyFailingAlgoritm();  } catch (IllegalStateException e) {  LOGGER.log(Level.SEVERE,  "Exception caught", e);  System.exit(2);  }  LOGGER.fine("Finished successfully");  }  private static void  randomlyFailingAlgorithm() {  double randomNumber = Math.random();  LOGGER.Log(Level.FINE,  "Generated random number:  {0}", randomNumer);  if (randomNumber < 0.5) {  throw new IllegalStateException(  "Invalid phase of the  Moon");  }  }  } |

Давайте рассмотрим работу с логерами на примере небольшой демонстрационной программы. Это программа детально логгирует все что в ней происходит. Она делает это на уровне FINE. И только в случае серьезной ошибки она залогирует исключение вместе со стек-трейсом на уровне SEVERE. Если запустить эту программу, то мы увидим следующее.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12. | Sep 13, 2015 4:50:02 PM  org.stepic.java.logging.LogDemo main  SEVERE: Exception caught  java.lang.IllegalStateException: Invalid  phase of the Moon  at org.stepic.java.logging.LogDemo.  randomlyFailingAlgorithm(  LogDemo.java:29)  at org.stepic.java.logging.LogDemo.main(  LogDemo.java:16) <5 internal  calls> |

В случае ошибки она залогирует SEVER стректрейс исключение или в случае, если ничего не произошло, то она не залогирует ничего. То, что она логировала на уровне FINE, в логе не появляется. Это потому, что по умолчанию логгер настроен на уровень INFO. То есть он молча игнорирует все, что приходит ему на более детальных уровнях. Например, FINE.

Это поведение можно переопределить в config файле. Он может выглядеть, например, следующим образом.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | # To use this config start JVM with  parameter:  # -Djava.util.logging.config.file=logging.  properties  .level=ALL  .handlers=java.util.loggin.ConsoleHanlder  java.util.logging.ConsoleHandler.level=ALL |

Здесь мы включаем специальный уровень логирования ALL, который разрешат видеть записи на абсолютно всех уровнях. Самый подробный лог. И кроме того, мы говорим, что вывод будет идти в консоль. За это и отвечает консоль Handler. Зачем нам явно говорить, что вывод должен идти в консоль. Мы ведь и так его уже видели в консоли. Дело в том, что если запустить виртуальную машину с переопределенным конфигом логирования, то все стандартные настройки аннулируются. Если здесь в нашем конфиге явно не указать этот Handler, то лог сообщения никуда не попадут. Поэтому мы его явно указываем. Чтобы запустить виртуальную машину с нашими настройками логирования, нужно передать ей специальный параметр 3. Сейчас мы это сделаем. И если теперь запустить нашу программу, то, вуаля, мы получили полный лог.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10. | Sep 13, 2015 4:51:43 PM  org.stepic.java.logging.LogDemo main  FINE: Started with ergements: [a, b, c]  Sep 13, 2015 4:51:43 PM  org.stepic.java.logging.LogDemo  randomlyFailingAlgorithm  FINE: Generated random number: 0.791  Sep 13, 2015 4:51:43 PM  org.stepic.java.logging.LogDemo main  FINE: Finised successfully |

Самый детальный. В том числе сообщение уровня FINE.

5 Ввод-вывод, доступ к файловой системе

5.1 Доступ к файловой системе

5.1.1 java.io.File работа с путями

В этом модуле мы поговорим о том, как из Java‑программы полазать по диску, по файловой системе, и что-нибудь прочитать или записать. Тема, казалось бы, достаточно простая, но на самом деле здесь есть что обсудить на целый модуль. Дело в том, что в Java за ввод/вывод отвечает достаточно большое количество классов, и некоторые из них даже дублируют функциональность друг друга, поэтому важно сложить общую картину и расставить все по полочкам. Начнем мы с того, как бегать по файловой системе: искать файлы, анализировать их свойства, копировать, перемещать, создавать, удалять файлы и директории. Содержимое файлов нас на данном этапе интересовать не будет.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9. | // on Windows:  File javaExecutable = new File(  "C:\\jdk1.8.0\_60\\bin\\java.exe");  File networkFolder = new File(  "\\\\server\\share");  // on Unix  File IsExecutable = new File("usr/bin/ls"); |

Задача доступа к файловой системе в Java решает класс File. Он живет в пакете java.io, где находятся и другие классы, отвечающие за ввод/вывод. Хотя не все, некоторые важные классы находятся в других пакетах. Я о них тоже расскажу. Экземпляры класса File представляют файлы и, как ни странно, директории на диске. Экземпляры создаются при помощи конструктора, принимающего строчку с путем. Строчка передается в формате, понятным текущей операционной системе. Например, под Windows путь может начинаться с буквы диска и двоеточия, а в качестве разделителя использоваться обратный слеш, который в строковых литералах нужно экранировать, поэтому он здесь удвоен. Под Unix никаких букв диска не бывает, разделителем является обычный прямой слеш.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9. | String sourseDirName = "src";  String mainFileName = "Main.java";  String mainFilePath = sourseDirName +  File.separator +  mainFileName;  // File.separator  // File.separatorChar |

Если вы в программе формируете пути к файлам, то пользуйтесь константами 8 или 9. Эти константы зависят от текущей операционной системы, под которой запущена Java‑программа и позволяют собирать пути, понятные данной платформе. Разница между этими двумя константами в том, что 8 — это String, а 9 — это char. В данном случае нет разницы что использовать.

|  |  |
| --- | --- |
| 1.  2. | // pathSeparator  // pathSeparatorChar |

Кстати, в классе File есть еще константы 1 и 2. Это символ, которым разделяются набор путей. Помните давным‑давно в первом модуле я говорил, что в класс Path под Windows пути разделяются точкой запятой, а под Unix двоеточием. Вот этот символ как раз и можно достать из этой константы.

|  |  |
| --- | --- |
| 1.  2. | File mainFile = new File(  sourseDirName, mainFileName); |

И еще один способ собрать путь с правильным разделителем — это использовать конструктор File с 2‑мя параметрами. Он внутри себя склеит два компонента пути, используя нужный разделитель.

Экземпляр классов File можно создать как с абсолютным путем, так и с относительным.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | File absoluteFile = new File(  "/usr/bin/java");  absoluteFile.isAbsolute(); // true  absoluteFile.getAbsolutePath();  // /usr/bin/java |

Абсолютный путь — это когда он однозначно указывает на файл, начиная с корня файловой системы.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9. | File relativeFile = new File("readme.txt");  relativeFile.isAbsolute(); // false  relativeFile.getAbsolutePath();  // /home/stepic/readme.txt  // getAbsolutePath()  // getAbsoluteFile(), |

Относительный — это когда чтобы найти файл нужно знать некую базовую директорию. Относительный путь можно превратить в абсолютный при помощи метода 8, возвращающего String, или аналогичного метода 9, возвращающего экземпляр File. Путь будет разрешен относительно текущей директории Java‑процесса, то есть из какой директории была запущена JVM. А если путь уже был абсолютным, то ничего не измениться.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10. | File file = new File("/usr/bin/java");  String path = file.getPath();  // /usr/bin/java  String name = file.getName(); // java  String parent = file.getParent(); // /usr/bin  // getParentFile() |

Имея на руках экземпляр класс File можно вынуть из него исходный путь в виде строки при помощи метода getPath(). Или не весь путь, а только его последний компонент, то есть имя файла или директорию при помощи метода getName(). Замечу, что расширение отдельно не выделяется, то есть getName() возвращает имя вместе с расширением. Отдельного метода для получения расширения нет. Если вам вдруг оно потребуется, то напишете пару строк кода с использования индексов из substring(). Если getName() возвращает последний компонент пути, то getParent() возвращает путь, из которого последний компонент вырезан, то есть, по сути, родительскую директорию. Родственный ему метод 10 возвращает то же самое, но не в виде строки, а в виде экземпляра File.

Предположим, у нас есть два экземпляра класса File. Как нам понять, указывают ли они на один и тот же файл на диске. Можно было бы сравнить из абсолютные пути, но есть одно проблема. В пути могут быть штуки вроде точки, обозначающей текущую директорию, 2-х точек, обозначающий переход к родительской директории, или символических ссылок, которые могут указывать вообще куда‑угодно. Из-за этого различные пути вообще могут на самом деле вести к одному файлу.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10. | File file = new File("./prg/../symlink.txt");  String canonicalPath =  file.getCanonicalPath();  // "/home/stepic/readme.txt"  // getCanonicalPath()  // getCanonicalFile()  // java.io.IOException |

В классе File() есть метод 7 и похожий на него 8. Этот метод приводит любой путь к каноническому виду, сначала вычисляя абсолютный путь, затем вырезая из него все избыточные компоненты, вроде точек, и разрешая символические ссылки. Приведя два пути к каноническому виду их уже можно сравнивать как строки. Методы 7 и 8 могут бросать исключения типа 10. Дело в том, что для разрешения символических ссылок необходимо обращение к диску, а оно может завершится ошибкой ввода/вывода, которое и будет выведено в виде исключения. Все операции, упомянутые до этих 2‑ух методов, обращение к диску не требовали, поэтому такое исключение не бросали. Данное исключение является проверяемым, поэтому компилятор потребует от вас либо явно отловить его при помощи try‑catch, либо задекларировать его на вашем методе при помощи ключевого слова throws.

5.1.2 java.io.File модифицирующие операции

Существование объекта File не привязано к существованию файла или директории на диске. Конструктор File ничего на диске не создает.

|  |  |
| --- | --- |
| 1.  2.  3.  4. | File java = new File("/usr/bin/java");  java.exists(); // true  java.isFile(); // true  java.isDirectory(); // false |

Таким образом путь может спокойно указывать на несуществующий файл или директорию. У объекта есть методы для проверки существования и типа: файл это или директория. Когда мы убедились, что это файл, и он существует, то можем запросить по нему дальнейшую информацию.

|  |  |
| --- | --- |
| 1.  2.  3. | File java = new File("/usr/bin/java");  java.length(); // 1536  java.lastModified(); // 1231914805000 |

Например, размер, который возвращается методом getLength(), или время последней модификации в миллисекундах с 1970 года. Если вдруг файл не существует, то эти методы будут возвращать нули, а не будут бросать исключения.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | File usrbin = new File("/usr/bin");  usrbin.exists(); // true  usrbin.isFile(); // false  ustbin.isDirectory(); // true  usrbin.list(); // String[]  usrbin.listFiles(); // File[] |

Если экземпляр File соответствует директории, то мы можем получить ее содержимое при помощи методов list() и listFiles(). Разница между ними в том, что первый возвращает массив строк, а второй массив экземпляров File. Если мы запрашиваем содержимое несуществующей директории, то вместо массива нам придет null. Содержимое возвращается только на один уровень. Если нужно получить все содержимое, включая поддиректории, то придется ручками написать рекурсивный обход.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | File[] javaSourceFiles = dir.listFiles(  f -> f.getName().endWith(".java"));  // java.oi.FileFiler:  // boolean accept(Fie pathname)  // java.io.FilenameFilter:  // boolean accept(File dir, String name) |

Есть перегруженные варианты методов list() и listFiles(), принимающие фильтр. Фильтр — это объект, реализующий интерфейс либо 4, либо 7. Оба эти интерфейса содержат ровно по одному абстрактному методу accept(), поэтому их можно инстанцировать при помощи лямбда‑выражений, чем мы и воспользовались в примере.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | try {  boolean success = file.createNewFile();  } catch (IOExeption e) {  // handle error  } |

Наконец, мы добрались до методов, позволяющих что-то в файловой системе поменять.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | try {  boolean success = file.createNewFile();  } catch (IOExeption e) {  // handle error  }  // createNewFile()  // IOExeption |

Первый такой метод 7 атомарно создает новый пустой файл. Он возвращает булевское значение: true, если все прошло успешно, или false, если файл уже существовал и создать его не удалось. Он может бросить 8, если в процессе создания случилась ошибка. Например, нет прав или не существует директория, в которой мы пытаемся его создать.

Вообще этот метод используется не очень часто. Потому что обычно нас интересует не просто создание файла, а запись туда каких-то данных. А классы, которые занимаются записью данных в файл, прекрасно умеют сами этот файл и создавать,

|  |  |
| --- | --- |
| 1. | // createNewFile() |

то есть явно вызывать 1 перед этим не требуется. Метод 1 нужен в тех случаях, когда нам важно убедиться, что файл создан именно нашей программой именно сейчас, чтобы случайно не начать писать в файл, созданный кем-то из вне. Гарантируется, что проверка существования и создания этого пустого файла в методе 1 происходит атомарно, то есть никакая другая программа или пользователь между этими двумя операциями вклиниться не может.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | File dir = new File("a/b/c/d");  boolean success = dir.mkdir();  boolean success2 = dir.mkdirs(); |

Для создания директорий есть методы mkdir() и mkdirs(). Разница между ними следующая. Метод mkdir() за один вызов может создать максимум одну директорию. Если нам нужно, как в примере, создать иерархию сразу из 4‑х вложенных директорий, то метод mkdir() с этим не справится. Нас спасет второй метод mkdirs(), который за один вызов может создать все уровни вложенности, столько, сколько нужно.

|  |  |
| --- | --- |
| 1. | boolean success = file.delete(); |

Метод delete(), очевидно, удаляет файл или директорию. Не очевидно здесь то, что в случае директории она обязательно должна быть пустой, иначе она удалена не будет. Поэтому часто можно встретить вручную написанный рекурсивный обход для удаления всех вложенных файлов и поддиректорий.

|  |  |
| --- | --- |
| 1. | boolean success = file.renameTo(targetFile); |

Метод renameTo() переименовывает файл. Он принимает не строчку с новым именем, а экземпляр File. То есть, в принципе, при помощи этого метода можно так переименовать файл, что он окажется в другой директории. Но тут есть нюансы. Например, известно, что переименование файла выполняется платформенно-зависимым способом и не гарантируется, что можно перенести файл с одной файловой системы на другую. А поскольку файловые системы могут быть смонтированы в произвольной директории, о чем мы из Java‑программы не знаем, то можно полагаться только на надежное переименование файла без переноса в другую директорию.

И опять, чтобы обойти это ограничение, программистам приходится либо писать свой метод, либо подключать какую-то стороннюю библиотеку, где такой метод уже написан. Например, Apache Commons. Такой метод будет поблочно читать исходный файл, записывать данные в новый файл, потом исходные файлы удалять. В случае с директорией, это нужно проделать рекурсивно для всего содержимого. Обратите внимание, что метода копирования в классе File вообще нет. Его тоже нужно либо писать руками, либо подключать из какой-то сторонней библиотеки.

5.1.3 java.nio.file.Path новый API

Все рассмотренные методы модификации файловой системы возвращают булевское значение: true, в случае успеха, или false, если что-то не получилось. Это, конечно, не самое удачное решение проектировщиков стандартной библиотеки. Лучше бы все методы в случае проблем бросали исключения, как метод createNewFile(). При текущем подходе приходится, во‑первых, писать везде if для проверки возвращаемого значения. Во‑вторых, в случае ошибки нам не откуда взять информацию о ее причинах. К сожалению, изменить поведение класса File теперь уже не представляется возможным из соображения обратной совместимости, которая в платформе Java особенно тщательно поддерживается. Осознавая все проблемы текущего API, его неудобства и недостатки, разработчики Java в 7 версии добавили абсолютно новый набор классов для работы с файловой системой.

|  |  |
| --- | --- |
| 1. | java.nio.file |

Новые классы живут в пакете 1. Новый API более стройный. Нет булевских флажков, нет дабросайт исключения. Кроме того, новые классы предоставляют более широкий набор возможностей. Можно работать с расширенными атрибутами файлов, например, с правами доступа или с владельцем. Можно работать с символическими ссылками, в частности, создавать их или узнавать куда ссылка ведет. Ну и конечно появились человеческие методы перемещения и копирования файла.

|  |  |
| --- | --- |
| 1. | // java.nio.file.Path |

Аналог класса File в новом API — это интерфейс 1. Он также представляет собой путь к файлу или директории, построенной по правилам текущей операционной системы. Объект Path не привязан к существованию файла на диске. Это, грубо говоря, просто строчка, обернутая набором методов для синтаксических манипуляций.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | Path path = Paths.get("prj/stepic");  File fromPath = path.toFile();  Path fromFile = fromPath.toPath(); |

Экземпляры Path получаются через класс Paths и его статический метод get(). Также добавлены методы конвертации Path в File и обратно.\

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12. | Path java = Paths.get("/usr/bin/java");  java.isAbsolute(); // true  java.getFileName(); // java  java.getParent(); // /usr/bin  java.getNameCount(); // 3  java.getName(1); // bin  java.resolveSibling("javap");  // /usr/bin/javap  java.startsWith("/usr"); //true  Paths.get("/usr").relativize(java);  // bin/java |

У интерфейса Path есть аналоги всех синтаксических операций, присутствовавших в классе File. А также добавлены многие удобные методы, которых у File не было. Например, получение компонента пути по его индексу. Или проверка того, является ли один путь префиксом другого. Или вычисление относительного пути между двумя местами в файловой системе.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10. | Path java = Paths.get("/usr/bin/java");  Files.exists(java); // true  Files.isRegularFile(java); // true  Files.size(java); // 1536  Files.getLastModifiedTime(java).toMillis();  // 123191485000  Files.copy(java,  Paths.get("/usr/bin/java\_copy"),  StandardCopyOption.REPLACE\_EXISTING); |

В интерфейсе Path, в отличии от класса File, нет методов доступа к файловой системе. Когда надо залезть на диск, то это делается через класс Files и его статические методы. Там есть аналоги всех операций, которые были в классе File и даже больше. Наконец-то, есть человеческие методы copy() и move(), не имеющие ограничений, связанных с нахождением файлов на одной файловой системе. Опционально туда можно передать дополнительные параметры. Например, что делать, если целевой файл уже существует.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10. | Path usrbin = Paths.get("/usr/bin");  Files.exists(usrbin); // true  Files.isDirectory(usrbin); // true  try (DirectoryStream<Path> dirStream =  Files.newDirectoryStream(usrbin)) {  for (Path child : dirStream) {  System.out.println(child);  }  } |

В новом API заметно отличается получение содержимого директории. Теперь это делается при помощи отдельного объекта DirectoryStream, получаемого при помощи статического метода 6, куда, кстати, дополнительно можно передать фильтр. DirectoryStream представляет собой директорию, открытую на чтение. Это ресурс операционной системы, который надо освобождать. Поэтому DirectoryStream надо использовать в блоке try с ресурсами, который гарантирует вызов метода close() при завершении блока. Этот код выглядит более громоздко, чем при использовании класса File и его методов list() и listFiles(), но на то есть причина. Директории бывают очень большими и не всегда хочется зачитывать их содержимое в память сразу целиком. Новый API позволяет обрабатывать содержимое в потоковом режиме небольшими порциями.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | Path dir = Paths.get("a/b/c/d");  Files.createDirectory(dir);  Files.createDirectories(dir); |

Для создания директорий есть методы 3 и 5. Разница между ними точно такая же, как между методами mkdir() и mkdirs() в классе File.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27. | Path directory = Paths.get("/tmp");  Files.walkFileTree(directory,  new SimpleFileVisitor<Path>() {  @Override  public FileVisitResult visitFile(  Path file,  BasicFileAttributes attrs)  throws IOException {  Files.delete(file);  return FileVisitResult.CONTINUE;  }  @Override  public FileVisitResult  postVisitDirectory(Path dir,  IOException exc)  throws IOException {  if (exc == null) {  Files.delete(dir);  return FileVisitResult.CONTINUE;  } else {  throw exc;  }  }  }); |

Что касается удаление директории со всем содержимым, то такого метода у нас до сих пор нет. Зато был добавлен общий механизм рекурсивного обхода директории, при помощи которого можно реализовать как удаление, так и копирование и все остальное, что потребуется. Для этого есть метод 3, который принимает директорию и объект, реализующий интерфейс FileVisitor. В интерфейсе есть методы, вызываемые в процессе обхода. Например, перед входом в директорию, после выхода из директории, после нахождения файла или после того, как файл нашли, но не смогли прочитать его атрибуты. В данном примере мы воспользовались классом SimpleFileVisitor и переопределили только два из его четырех методов. SimpleFileVisitor сам по себе ничего не делает и удобен как раз в таких случаях, чтобы от него унаследоваться и переопределить одни‑два метода, а не все четыре, большая часть из которых будут пустышками. Кстати, методы FileVisitor должны возвращать специальный FileVisitResult, говорящий что делать дальше. FileVisitResult — это перечисление, у которого есть такие значения как продолжить, остановится и пропустить. Это может выглядеть немного устрашающе, но, к счастью, можно один раз реализовать себе такой FileVisitor, создать его общий экземпляр, записать в какой-нибудь public static final переменную и использовать по всей программе. После этого удаление директории будет записываться в одну строчку.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25. | Path zipPath = Paths.get(  "jdk1.8.0\_60/src.zip");  try (FileSystem zipfs =  FileSystems.newFileSystem(  zipPath, null))  for (Path path :  zipfs.getRootDirectories()) {  Files.walkFileTree(path,  new SimpleFileVisitor<Path>(  ) {  @Override  public FileVisitResult visitFile(  Path file,  BasicFileAttributes  attrs)  throws IOExeption {  System.out.println(file);  return FileVisitiResult.  CONTINUE;  }  })  }  } |

Новый API не только строен и функционален, но и более гибко устроен внутри. В частности, он позволяет программистам реализовывать различные виртуальные файловые системы. Например, в стандартной библиотеке есть реализация файловой системы поверх ZIP-архива. Мы открываем ZIP и работаем с его содержимым как с обычным файлами при помощи тех же классов. Это абсолютно прозрачно для нашего кода, который один раз написан и может быть переиспользован для разных файловых систем без изменений. Ничто не мешает нам реализовать файловую систему, работающую с файлами внутри любого другого типа архива или вообще на удаленном компьютере.

Новый набор классов в пакете java.nio.file рекомендуется для использования во всех новых Java‑программах, так как он по многим параметрам лучше старого. Однако, все понимают, что от старого API теперь уже не избавится. За 20 лет его существования в мире написано невообразимое количество кода, использовавшего класс java.io.File и этот код уже теперь никуда не денется. Поэтому Java-программистам приходится знать два разных API для доступа к файловой системе и использовать один из них в зависимости от обстоятельств.

5.2 Потоки байт

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27. | package java.io;  public abcstract class InputStream  implements Closeable {  public abstract int read()  throws IOExeption;  public int read(byte b[])  throws IOExeption {  return read(b, 0, b.length);  }  public int read(byte b[], int off,  int len) throws IOExeption {  // ...  }  public long skip(long n)  throws IOExeption {  // ...  }  public void close() throws IOExeprtion {}  // ...  } |

Давайте посмотрим на основные методы, определенные в этих классах. И начнем с класса InputStream. Первый метод read(). Он возвращает следующий байт из входного потока и сдвигается дальше, то есть трижды вызвав метод read(), мы получим три идущих подряд байта из потока. Возвращаемое значение имеет тип int, но фактически из потока считывается только один байт. Более широкий тип int нужен, чтобы иметь возможность обозначить конец потока. Когда конец достигнут, метод вернет -1. Если вернулась не -1, то фактически считанный байт можно получить, приведя int к byte при помощи оператора приведения типа, то есть фактически взяв младшие 8 бит int. При ошибках ввода/вывода этот и остальные методы будут бросать проверяемое исключение IOExeption. Дальше я не буду каждый раз это повторять.

Читать большие объемы данных побайтно не очень эффективно, поэтому есть еще 2 метода, которые читают из потока блок байтов и записывают их в переданный массив. Первый метод принимает только массив байтов и пытается считать из потока количество байт, равное размеру массива. Второй метод принимает еще два числа: индекс элемента массива, начиная с которого этот массив нужно заполнять, и количество байт, которое нужно считать их входного потока и записать в массив. По возвращению из метода массив будет заполнен данными из входного потока. Возвращенное методом значение типа int означает количество фактически считанных байт. Оно может быть меньше, чем мы просили, если поток совсем закончился или просто в данный момент в нем недостаточно данных. На возвращенное значение обязательно надо смотреть. Ровно столько байт было записано в переданный массив и ровно столько байт мы можем дальше использовать. Если вернулась -1, это означает конец потока.

А еще есть метод skip(), который просто подпускает заданное количество байт и никуда их не сохраняет. Он возвращает количество байт, которое фактически удалось пропустить. Это количество также может быть меньше, чем мы просили.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26. | package java.io;  public abstract class OutputStream  implements Closeable, Flushable {  public abstract void write(int b)  throws IOExeption;  public void write(byte b[])  throws IOExeption {  write(b, 0, b.length);  }  public void write(byte b[], int off,  int len) throws IOExeption {  // ...  }  public void flush() throws IOExeption {  // ...  }  public void close() throws IOExeprtion {}  // ...  } |

Второй класс OutputStream содержит аналогичные методы для записи байтов по одному или блоками. Метод write() от int выводит в поток один байт. Младшие 8 бит переданного значения типа int. Когда у нас большой объем данных для записи, то лучше воспользоваться перегруженными вариантами метода write(), принимающего массив. Первый берет все содержимое массива, а второй — указанное количество байт, начиная с указанного индекса в массиве. Метод flush() сбрасывает возможные промежуточные буфера, где конкретная реализация OutputSream может накапливать данные перед передачей операционной системе. Просто вызов метода write() не гарантирует, что данные тут же будут переданы операционной системе для обработки.

В обоих классах есть метод close(), закрывающий поток и освобождающий связанные с ним ресурсы. Закрытие включает в себя также и сброс буферов, то есть вызывать flush() перед close() не требуется. Рекомендуется использовать стримы в блоке try с ресурсами, чтобы метод close() гарантировано был вызван.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | int totalBytesWritten = 0;  byte[] buf = new byte[1024];  int blockSize;  while ((blockSize = inputStream.read(buf))  > 0) {  outputStrem.write(buf, 0, blockSize);  totalBytesWritten += blockSize;  } |

Для иллюстрации работы со стримами приведу часто используемую идиому копирования данных из входного потока в выходной. Нам потребуется временный буфер, размером, например, 1 Кб. Мы в цикле вызываем метод read(), передавая туда наш временный буфер и обязательно сохраняя результат — количество прочитанных байт. Смотрим на это количество. Если это положительное число, значит мы что-то прочитали и можем вывести этот выходной поток, иначе поток закончился и копирование завершается.

5.2.2 Конкретные наследники

Напомню, что классы InputStream и OutputStream абстрактные и ничего не говорят о том, откуда данные читаются или куда пишутся. Давайте посмотрим с какими источниками и получателями данных можно‑таки работать и откуда брать экземпляры соответствующий конкретных классов‑наследников.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12. | InputStream inputStream =  new FileInputStream(  new File("in.txt"));  OutputStream outputStream =  new FileOutputStream(  new File("out.txt"));  // FileInputStream  // FileOutputStream  // java.oi.File |

Самая насущная потребность — это работа с файлами на диске. И она покрывается классами 9 и 10. В конструктор они принимают либо строку с путем к файлу, либо экземпляр 12.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | InputStream inputStream =  Files.newInputStream(  Paths.get("in.txt"));  OutputStream outputStream =  Files.newOutputStream(  Paths.get("out.txt"));  // java.nio.file.Path  // Files.newInputStream()  // Files.newOutputStream() |

Если у нас на руках есть экземпляр нового класса 9, то получить поток, соответствующий ему, можно при помощи методов 10 и 11. Конкретный класс потока в этом случае мы не знаем. Он зависит от того, на какую реальную или виртуальную файловую систему указывает данный путь.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10. | try (InputStream inputStream =  Main.class.getResourseAsStream(  "Main.class")) {  int read = inputStrem.read();  while (read >= 0) {  System.out.ptintln("%02x", read);  read = inputStream.read();  }  } |

Есть возможность прочитать ресурс из текущего класса Path, с которым запущена виртуальная машина. В зависимости от того, упакована ли наша программа в JAR это будет либо файл внутри архива, либо файл на диске. Эта деталь от нас скрыта. Мы просто получаем InputStream соответствующим содержимому файла. 2 — класс может прочитать свой собственный байт-код. Предлагается, что дело происходит в классе с именем Main.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18. | try (Socket socket = new Socket(  "ya.ru", 80)) {  OutputStream outputStream =  socket.getOutputStream();  outputStream.write(  "GET / HTTP/1.0\r\n\r\n".  getBytes());  outputStream.flush();  InputStream inputStream =  socket.getInputStream();  int read = inputStream.read();  while (read >= 0) {  System.out.println((char) read);  read = inputStream.read();  }  } |

Стримы используются и для обмена данными по сети. Надо только сначала открыть сетевое соединение Socket, а затем получить из него OutputStream, в который можно что-то выводить, и InputStream, из которого можно прочитать результат.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | byte[] data = {1, 2, 3, 4, 5};  InputStream inputStream =  new ByteArrayInputStream(data);  ByteArrayOutputStream outputStream =  new ByteArrayOutputStream();  // ...  byte[] result = outputStream.toByteArray();  // ByteArrayInputStream  // ByteArrayOutputStream |

Иногда данные уже записаны в память виртуальной машины, и мы хотим скормить их какому-то коду, принимающему InputStream. Для этого есть класс 10. Аналогично можно подсунуть коду, записывающему данные в OutputStream, экземпляр класса 11. А потом получить записанные данные в виде массива byte[]. Такой трюк удобно использовать для тестирования своего кода.

5.2.3 Потоки-обертки

Методы read() и write() в классах InputStream и OutputStream универсальны, но слишком низкоуровневые и поэтому не очень-то удобны. Чтобы записать в поток какие-то структурированные типизированные данные, например, строки, числа, объекты, нам придется изрядно попотеть, чтобы сначала преобразовать это все в массив байтов. К счастью, разработчики Java это учли и придумали такую штуку: один стрим может быть завернут в другой. Внутренний стрим занимается реальным вводом/выводом, а внешний его декорирует, добавляет какую‑то полезную функциональность.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24. | package java.io;  public class DataOutputStream  extends FilterOutputStream  implements DataOutput {  public DataOutputStream(  OutputStream out) {  // ...  }  public final void writeInt(int v)  throws IOExeption {  out.write((v >>> 24) & 0xFF);  out.write((v >>> 16) & 0xFF);  out.write((v >>> 8) & 0xFF);  out.write((v >>> 0) & 0xFF);  intCount(4);  }  // ...  }  // DataOutputStream |

Например, класс 24 оборачивает любой другой произвольный OutputStream и добавляет к низкоуровневым методам write() удобные методы для записи всех примитивных типов в Java. А также метод writeUTF() для записи строк. Внутри, разумеется, эти значения преобразуются в байты и выводятся в тот самый обернутый OutputStream.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28. | package java.io;  public class DataInputStream  extends FilterInputStream  implements DataInput {  public DataInputStream(  InputStream in) {  // ...  }  public final int readInt()  throws IOExeption {  int ch1 = in.read();  int ch2 = in.read();  int ch3 = in.read();  int ch4 = in.read();  if ((ch1 | ch2 | ch3 | ch4) < 0) {  throw new EOFExeption();  return ((ch1 << 24) + (ch2 << 16) +  (ch3 << 8) + (ch4 << 0));  }  // ...  }  // DataInputStream  // DataOutputStream |

Также есть класс 27 который оборачивает другой произвольный InputStream и предоставляет методы для чтения всех примитивных типов и строк. Связка 27 и 28 позволяют удобно обмениваться примитивными значениями и строками между 2‑мя программами.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24. | byte[] originalData = {1, 2, 3, 4, 5};  ByteArrayOutputSream os =  new ByteArrayOutputStream();  try (OutputStream dos =  new DeflaterOutputStream(os)) {  dos.write(otiginalData);  }  byte[] defaultData = os.toByteArray();  try (InflaterInputStream iis =  new InflaterInputStream(  new ByteArrayInputStream(  dafaultedData))) {  int read = iis.read();  while (read >= 0) {  System.out.prinln("%02x", read);  read = iis.read();  }  }  // DeflaterOutputStream  // InflaterInputStream |

И еще один интересный пример — класс 23 реализует сжатие записываемых данных по алгоритму Deflater на лету, а симметричный ему 24 по ходу чтения распаковывает данные, сжатые этим алгоритмом. Обернув обычные стримы в Inflater и Deflater мы можем абсолютно прозрачно для всего кода программы поддержать сжатие данных. Здесь уместно будет добавить, что закрытие внешнего стрима закрывает также и вложенные в него стримы. Поэтому достаточно одного вызова close() на самом верхнем внешнем стриме.

5.3 Потоки символов

5.3.1 Абстрактные Reader и Writer

Данные бывают не только двоичные, но и текстовые. Для работы с текстовыми данными нужны специализированные классы. Более удобные, чем низкоуровневые потоки byte, InputStream и OutputStream. Такие классы в стандартной библиотеке Java есть и их целое семейство. Сейчас мы с ними познакомимся.

|  |  |
| --- | --- |
| 1.  2. | // java.io.Reader  // java.io.Writer |

Начнем мы также с базового абстрактного уровня, на котором есть два абстрактных класса 1 и 2. При первом взгляде на их устройство может случится дежавю — где‑то мы уже это видели.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28. | package java.io;  public abstract class Reader  implements Readable, Closeable {  public int read() throws IOExeption {  // ...  }  public int read(char cbuf[])  throws IOExeption {  return read(cbuf, 0, cbuf.length);  }  public abstract int read(char cbuf[],  int off, int len)  throws IOExeption;  public long skip(long n)  throws IOExeption {  // ...  }    public abstract void close()  throws IOExeption;  // ...  } |

Класс Reader устроен абсолютно аналогично InputStream. У него точно такие же методы read(), только вместо массивов байтов они принимают массивы символов. Ну а метод read() без параметров возвращает тот же int, но в этом int нужно смотреть не на младшие один байт, а на младшие два байта.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28. | package java.io;  public abstract class Writer  implements Appendable, Closeable,  Flushable {  public void write(int c)  throws IOExeption {  // ...  }  public void write(char cbuf[])  throws IOExeption {  write(sbuf, 0, sbuf.length);  }  public abstract void write(char sbuf[],  int off, int len)  throws IOExeption;  public abstract void flush()  throws IOExeption;  public abstract void close()  throws IOExeprtion;  // ...  } |

Класс Writer аналогичен классу OutputStream только с заменой byte на char. И еще у него есть парочка перегруженных вариантов метода write(), принимающих строку, не влезший в этот код.

5.3.2 Конкретные наследники

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9. | Reader reader = new InputStreamReader(  inputStream, "Utf-8");  Charset charset = StandardCharsets.UTF\_8;  Writer writer = new OutputStreamWriter(  outputStream, charset);  // InputStreamReader  // OutputStreamWriter |

Откуда мы можем читать и куда можем писать символы. Во‑первых, можно превратить производный поток байтов в поток символов, завернув его в 8 или в 9. При этом указывается кодировка, которая будет определять правила превращения символов в байты и обратно.

Кодировки бывают разные, простые и не очень, фиксированные или переменной длины. Например, широко используемая кодировка UTF‑8 кодирует каждый символ последовательностью от одного до шести байт и может представить любой символ Unicode. Более простая кодировка, например, cp866, знакомая еще со времен DOS, кодирует один символ одним байтом и поэтому может представить всего 256 различных символов. Тут, конечно, о полном покрытии Unicode никакой речи не идет.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13. | Reader reader = new InputStreamReader(  inputStream, "Utf-8");  Charset charset = StandardCharsets.UTF\_8;  Writer writer = new OutputStreamWriter(  outputStream, charset);  // Charset.forName("...")  // StandardCharsets  // java.nio.charset  // InputStreamReader  // OutputStreamWriter  // Charset.defaultCharset() |

Передать кодировку можно либо в виде строки с именем, либо в виде объекта типа charset. Экземпляр charset можно получить либо по тому же строковому имени кодировки при помощи 8 от строки, либо взять готовые уже экземпляр из стандартного набора, доступного в классе 9. В этот класс вынесен минимальный набор кодировок, которые все JVM обязаны поддерживать. Это такие кодировки как ASCII, UTF-8 и UTF-16. Через 8 можно получить больше различных кодировок, но тут уже никто не сможет гарантировать вам переносимость вашей программы. Классы Charset и 9 живут в пакете 10. Если в конструктор 11 или 12 не передать кодировку, то будет использоваться системная кодировка по умолчанию. Узнать эту системную кодировку, получить ее в явном виде можно из метода 13. Эта кодировка зависит от текущей операционной системы и ее настроек. Поэтому переносимые программы не должны на нее полагаться. Рекомендуется все‑таки при создании Reader и Writer указывать кодировку явно.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9. | Reader reader = new FileReader("in.txt");  Writer writer = new FileWriter("out.txt");  Reader reader2 = new InputStreamReader(  new FileInputStream("in.txt"),  StandardCharsets.UTF\_8);  Writer writer2 = new OutputStreamWriter(  new FileOutputStream("out.txt"),  StandardCharsets.UTF\_8); |

Возвращаемся к вопросу о том откуда берутся Reader и Writer. Для чтения и записи текстовых файлов есть специализированный классы FileReader и FileWriter. Но почему‑то они не позволяют явно указать кодировку, поэтому для работы с текстовыми файлами в фиксированной известной нам кодировки обычно используются конструкции 4 и 7.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19. | Reader reader = new CharArrayReader(  new char[] {'a', 'b', 'c'});  Reader reader2 = new StringReader(  "Hellow World!");  CharArrayWriter writer =  new CharArratyWriter();  writer.write("Test");  char[] resultArray = writer.toCharArray();  StringWriter writer2 = new StringWriter();  writer2.write("Test");  String relultString = writer2.toString();  // CharArrayReader  // StringReader  // CharArrayWriter  // StringWriter |

А еще есть классы 16 и 17 для чтения символов из массива или строки в памяти. И есть симметричные им 18 и 19 для записи в память. Их удобно использовать для тестирования кода.

5.3.3 Классы-обертки

Как и в случае со стримами всякая удобная и полезная функциональность добавляется Reader и Writer при помощи вкладывания их друг в друга.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17. | package java.io;  public class BufferedReader extends Reader {  public BufferedReader(Reader in) {  // ...  }  public String readLine()  throws IOExeption {  // ...  }  // ...  }  // BufferedReader |

Например, класс 17 оборачивает произвольный Reader и добавляет к нему буферизацию. То есть даже если мы читаем из 17 символы по одному он на самом деле запрашивает их у нижележащего ридера большими блоками и сохраняет в своем внутреннем буфере. Это должно положительно сказываться на производительности в случае больших объемов данных. Польза 17 этим не ограничивается. Он добавляет удобный метод readLine(), читающий с потока целую строку до ближайшего символа конца строки. Сам символ, разделяющий строки, не возвращается. Если мы дочитали до конца и поток закончился, то получим из метода null.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12. | try (BufferedReader reader =  new BufferedReader(  new InputStreamReader(  new FileInputstream("in.txt"),  StandardCharsets.UTF\_8))) {  String line;  while ((line = reader.readLine()) !=  null) {  // process line  }  } |

Раньше, чтобы построчно прочитать и обработать файл приходилось городить что-то на подобии кода выше. FileInputstream вкладывали внутрь InputStreamReader, который вкладывали внутрь BufferedReader и читали при помощи метода readLine() .

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23. | try (BufferedReader reader =  Files.newBufferedReader(  Paths.get("in.txt"),  StandardCharsets.UTF\_8)) {  String line;  while ((line = reader.readLine()) !=  null) {  // process line  }  }  List<String> lines = Files.readAllLines(  Paths.get("in.txt"),  StandardCharsets.UTF\_8);  for (String line : lines) {  // process line  }  // java.nio.file  // Files.newBufferedReader()  // Files.readAllLines |

Теперь с появлением пакета 21 есть удобный метод 22, который скрывает внутри себя многоэтажную конструкцию с вложенными стримами. Если файл небольшой, можно сделать еще проще — воспользоваться другим удобным методом 23, который вернет строки в виде списка сразу все.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16. | try (BufferedWriter writer =  Files.newBufferedWriter(  Paths.get("out.txt"),  StandardCharsets.UTF\_8)) {  writer.write("Hello");  writer.newLine();  }  List<String> lines = Arrays.asList(  "Hellow", "world");  Files.write(Paths.get("out.txt"), lines,  StandardCharsets.UTF\_8);  // BufferedReader  // BufferedWriter |

Симметричный классу 15 класс 16 добавляет аналогичную буферизацию на запись. То есть копит большие куски данных и сбрасывает их по мере накопления в оборачиваемый writer. А еще 16 добавляет метод newLine(), выводящий в выходной поток разделитель строк, используемый на данной платформе. Хотя, конечно, полезность этого метода не идет ни в какое сравнение с методом readLine() с 15. Для небольших объемов данных можно воспользоваться удобным методом Files.write(), который принимает список строк и внутри себя делает все необходимое для его записи в выходной текстовый файл.

5.3.4 Форматированный вывод

Классы Reader и Writer универсальны, но не слишком удобны для повседневного использования. То есть ввод/вывод значений разных типов требует от программиста написание кода по конвертации в строку или массив символов и обратно. К счастью, такой код уже написан и добавлен в стандартную библиотеку. Форматированный ввод/вывод значений разных типов реализован в Java как надстройка, следующий уровень над Reader и Writer.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28.  29. | package java.io;  public class PrintWriter extends Writer {  public PrintWriter(Writer out) {  // ...  }  public void print(int i) {  // ...  }  public void println(Object obj) {  // ...  }  public PrintWriter ptintf(  String format, Object ... args) {  // ...  }  public boolean checkError() {  // ...  }  // ...  }  // PrintWriter |

Давайте сначала о выводе. Форматированный вывод обеспечивает класс 29. В нем есть методы print() и println(), перегруженные для всех примитивных типов, а также строк и объектов. 29 конвертирует все это в строки и спускает в нижележащий Writer для вывода. Объекты преобразуются в строки при помощи вызова метода toString(). А еще здесь есть метод ptintf(), знакомый по языку C. Он принимает строку‑шаблон и набор параметров, которые в эту строку будут подставляется. Места подстановки параметров обозначаются специальными последовательностями символов. Например, %s — это место подстановки строки, %d — это место подстановки целого числа, %3f — это место подстановки вещественного числа, которое будет округленно либо дополнено нулями до 3‑х цифр после запятой. Полное описание формата ищите в Javadoc. Существенное отличие методов 29 состоит в том, что они не бросают IOExeption. Вместо этого они молча устанавливают внутренний флаг ошибки, который можно проверить при помощи метода checkError(). С одной стороны, это удобно, потому что не надо возиться с проверяемым IOExeption. С другой стороны, мы так легко можем пропустить ошибку.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28.  29.  30.  31. | package java.io;  public class PrintStream  extends FileterOutputSream  implements Appendable, Closeable {  public PrintStream(OutputStream out) {  // ...  }  public void print(int i) {  // ...  }  public void println(Object obj) {  // ...  }  public PrintWriter ptintf(  String format, Object ... args) {  // ...  }  public boolean checkError() {  // ...  }  // ...  }  // OutputStream |

У класса PrintWriter есть близкий родственник класс PrintStream. На вид он очень похож, так как предоставляет те же самые методы print, println и ptintf и не бросает исключений. Но он оборачивает 31, а не Writer, и сам является 31. Внутри своих методов print он делает сразу два преобразования. Сначала из примитивных типов и объектов в строку, в потом сразу в байты, которые передаются нижележащему 31 для вывода. То есть получается этакий гибрид 31 и PrintWriter.

5.3.5 Форматированный ввод

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | // java.io.StreamTokenizer  StreamTokenizer streamTokenizer =  new StreamTokenizer(  new StringReader("Hellow world"));  // java.util.StringTokenizer  StringTikenizer stringTockenizer =  new StringTockenizer("Hellow world”);  // java.io.StreamTokenizer  // java.util.StringTokenizer |

Что является аналогом PrintWriter и PrintStream для удобного зачитывания примитивных значений и строк из потока символов. Таких классов в стандартной библиотеке Java есть несколько. Первый из них 10 умеет разбирать поток на слова и числа. Простенький, не очень богат по функциональности и не очень удобный в использовании класс. Второй 11 похож на 10, но читает из строки и почему‑то не поддерживает некоторые из возможностей 10. Например, не умеет отдельно распознавать числа. Он тоже не очень удобен. Оба эти класса существуют с Java 1.0, а с тех пор представление об удобстве программиста и о том, как делать хорошие классы заметно изменилось. Поэтому мы на них не будем подробно останавливаться. Рассмотрим сразу их современную, удобную и функционально-богатую альтернативу.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13. | Reader reader = new StringReader(  "abc|true|1,1e3|-42");  Scanner scanner = new Scanner(reader).  useDelimiter("\\|").useLocale(  Locale.forLanguageTag("ru"));  String token = scanner.next();  boolean bool = scanner.nextBoolean();  double dbl = scanner.nextDouble();  int integer = scanner.nextInt();  // java.util.Scanner |

Это класс 13. Он создается, например, на основе Reader, хотя есть и другие конструкторы. И позволяет парсить в нем все примитивные типы и строки. У сканера есть некоторое количество настроек, влияющих на его работу. Например, можно указать шаблон разделителя, который будет использоваться для нарезания входного потока на токены. По умолчанию это пробельные символы, но можно задать что-то свое. Например, как у нас вертикальные палки. И другой важный параметр — Locale. Locale отвечает за региональные настройки и, в частности, влияет на то, как форматируются числа. Например, с каким десятичным разделителем. В данном примере мы используем русскую Locale. И это позволяет нам распознавать числа, у которых в качестве десятичного разделителя используется запятая.

5.3.6 Стандартные потоки

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12. | package java.lang;  public final class System {  public static final InputStream in =  null;  public static final PrintStream out =  null;  public static final PrintStream err =  null;  // ...  } |

В заключение давайте заглянем в класс System и посмотрим, что из себя представляют объекты in, out и err, соответствующие стандартным потоком ввода, вывода и ошибок, при помощи которых программа взаимодействует с внешним миром и с консолью. System.in — это InputStream, поток байт. Если программе на вход подается текстовая информация, то удобно обернуть этот InputStream в InputStreamReader плюс BufferedReader или сразу в Scanner. System.out и System.err — это PrintStream, то есть можно выводить в них как двоичные данные при помощи метода write(), так и текстовые при помощи методов print, println и printf. Чем мы, кстати, и пользовались, начиная с нашей первой программы HelloWorld. Любопытно, что поля объявлены как финальные со значением null. На самом деле они конечно же будут инициализированы какими‑то конкретными объектами. Это сделает виртуальная машина и ей final не указ, но не пытайтесь повторить это у себя дома.

5.4 Продвинутые возможности

5.4.1 Запуск внешних процессов

Мы с вами рассмотрели уже целую кучу классов, отвечающих за ввод/вывод. Однако, как ни странно, это еще не все. В стандартной библиотеке Java есть еще не мало классов, имеющих отношение к вводу/выводу. Хотя они уже, конечно, нужны далеко не в каждой программе и предоставляют какие-то специальные продвинутые возможности. Подробный разговор о таких возможностях выходят за рамки базового курса, но небольшой обзор будет уместен.

Из Java‑программы можно запускать другие процессы на вашем компьютере. Достаточно знать путь у исполняемому файлу на файловой системе и параметр, который ему можно передать для запуска. Зачем это может быть нужно? Ну, например, может стоять задача организовать наблюдение за каким‑то серверным процессом и перезапускать его, если он вдруг зависнет или упадет. Ничто не мешает реализовать такое наблюдение в виде Java‑программы.

Или другой пример. До выхода Java 7 с ее новым API доступа к файловой системе java.nio.file нельзя было из Java создать, например, символическую ссылку в файловой системе или получить расширенные атрибуты файла или директории, такие как владельца и права доступа. Реализовывать это приходилось при помощи внешних средств: запуска внешней команды создания симлинка или внешней команды вывода расширенной информации о файле и директории с последующим разбором того, что она вернула. И хотя сейчас в Java 8 благодаря этому новому API java.nio.file конкретно эти две вещи можно уже делать, но всегда останутся задачи, которые для Java слишком низкоуровневы или платформенно-специфичны. И поэтому, в стандартной библиотеке их не будет. Например, форматирование диска или настройка параметров сетевого интерфейса, или изменения других параметров операционной системы. Это всегда будет требовать выхода за рамки Java. И поэтому время от времени приходится запускать внешние процессы.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28.  29.  30.  31.  32.  33.  34.  35.  36.  37.  38. | package org.stepic.java.io;  import java.io.\*;  import java.lang.ProcessBuilder.Redirect;  public class ProcessDemo {  public static void main(String[] args)  throws Exception {  ProcessBuilder processBuilder =  new ProcessBuilder();  processBuilder.command("ls", "-l").  directory(new File(  "/home/stepic/Vladykin")).  redirectInput(Redirect.from(  newFile("/dev/null"))).  redirectOutput(  Redirect.PIPE).  redirectError(  Redirect.INHERIT);  Process process =  processBuilder.start();  try (BufferedReader reader =  new BufferedReader(  newInputStreamReader(  process.getInputStream()))) {  reader.lines().forEach(  System.out::println);  }  int exitValue = process.waitFor();  if (exitValue != 0) {  System.err.println("Subprocess  terminated abnormally");  }  }  } |

Давайте посмотрим, как это делается. Всю подготовительную работу по запуску внешнего процесса делает класс ProcessBuilder. Ему можно указать запускаемую команду с параметрами, рабочую директорию, в которой будет запускаться новый процесс, а также ему можно сказать к чему будут подключаться стандартные потоки ввода, вывода и ошибок нового процесса. Можно, например, организовать PIPE между Java‑процессом и запускаемым процессом. Тогда мы со стороны Java‑программы сможем явно получить этот поток Stream и сможем в него писать или из него читать. Или новый процесс может унаследовать поток Java‑процесса. И все что тот процесс выведет в свой стдЭрр, попадет в стдЭрр нашего Java‑процесса. Или можно сделать даже так: перенаправить поток в файл или из файла. В данном случае мы направляем на вход запущенному процессу содержимое файла dev/null. В Unix системах это такой специальный виртуальный файл, в который можно выводить все что угодно, и оно там исчезает как в черной дыре. А при попытке чтении их этого файла он оказывается пустым. После того, как параметры запуска настроены, можно стартовать новый процесс методом start(). В результате мы получим экземпляр класса Process. Если какие‑то из потоков ввода/вывода настроены как PIPE, то мы можем взять у процесса этот поток и поработать с ним. Надо только помнить: то, что для запускаемого процесса является потоком вывода, со стороны Java‑процесса является потоком ввода и наоборот. В данном примере мы принимаем все, что запущенный процесс выводит, и записываем это в поток стандартного вывода нашего Java процесса, то есть, по сути, руками реализуем режим INHERIT. При помощи метода waitFor() можно дождаться завершение дочернего процесса и получить его код возврата. Ноль соответствует успешному завершению, а любые другие значения — какой-то ошибки. И давайте запусти, чтобы убедиться что оно работает.

5.4.2 Классы java.nio

Вторая группа классов ввода/вывода, которые могут быть иногда полезны, это классы пакета java.nio и java.nio.chanels. Эти классы более низкоуровневые, чем знакомые нам классы пакета java.io, поэтому они не так удобны для повседневных задач вроде чтения текстовых файлов. Однако в определенных ситуациях их использование позволяет достичь лучшей производительности и большей масштабируемости.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28.  29.  30.  31.  32.  33.  34.  35.  36.  37.  38.  39.  40.  41.  42.  43.  44.  45.  46.  47.  48.  49. | package org.stepic.java.io;  import java.io.IOExeption;  import java.nio.ByteBuffer;  import java.nio.chanels;  import java.nio.file.Path;  import java.nio.file.Paths;  public class NioDemo {  public static void main(String[] args)  throws IOException {  Path path Paths.get(“  src/org/stepic/java/io/  NoDemo.java”);  try (ReadableByteChannel in =  FileChannel.open(path);  WritableByteChanel out =  Chanels.newChanel(  System.out)) {  ByteBuffer buffer =  ByteBuffer.allocate(  1024);  while (in.read(buffer >= 0 ||  buffer.position() !=  0) {  buffer.flip();  out.wtite(buffer);  buffer.compact();  }  }  // try (InputStream in =  // Files.newInputStream(path);  // OutputStream out =  // System.out) {  //  // byte[] buffer = new byte[1024];  // int bytesRead;  // while ((bytesRead = in.read(  // buffer)) > 0) {  // out.write(buffer, 0,  // bytesRead);  // }  // }  }  } |

Основные сущности этого набора классов это Buffer, аналог массива, и Channel, канал, аналог Input- OutputStream. Ключевое свойство nio — неблокирующий ввод/вывод. Чтобы понять, что это такое, давайте вспомним знакомые нам классы InputStream и OutputStream. Когда мы вызываем на InputStream метод read(), любой из перегруженных вариантов, то программа как-бы становится на паузу, блокируется, в ожидании, пока из потока не будет прочитан хотя бы один байт. Если мы читаем, допустим, данные, которые нам посылают по сети с другого компьютера, да еще по не очень надежному каналу, то возможно ждать придется долго. Аналогично с методом write(). Он не вернет управление, пока все не запишет. Никакого способа прервать этого ожидания нет. И это называется блокирующий ввод/вывод. В противоположность ему неблокирующий ввод/вывод, предоставляемый классом из nio, — это возможность сразу прочитать или записать столько данных, сколько можно и ничего не ждать. Если в канале нет ни одного доступного для чтения байта или канал не готов ничего принять, значит метод сразу возвращает управление, ничего не делая и не ожидая. Располагая возможностью неблокирующего ввода/вывода можно, например, из одного треда работать сразу с многими каналами. Мы просто будем опрашивать их состояния и обрабатывать данные по мере поступления, то есть можно скажем написать программу сервер, которая в одном треде будет обслуживать сотни клиентов. При использовании классического java.io на каждого клиента потребовался бы свой тред.

Еще раз повторюсь, что данные классы представляют более низкоуровневый интерфейс ввода/вывода, который с одной стороны является более гибким и потенциально более производительным, но с другой стороны значительно менее удобным для повседневных задач. То есть прибегать к этим классам стоит в тех редких случаях, когда вы реально упираетесь в неэффективность блокирующего ввода/вывода.

5.4.3 Сериализация

И третья продвинутая возможность — это сериализация. Речь о том, что можно взять объект из памяти виртуальной машины, превратить его в набор байтов вместе со всеми значениями его полей и с этим набором байтов сделать все что угодно, например, записать в файл или отправить по сети. А потом этот набор байтов можно будет прочитать, декодировать и получить точно такой же объект.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28.  29.  30.  31.  32.  33.  34.  35.  36.  37.  38.  39.  40.  41.  42.  43.  44.  45.  46.  47.  48.  49.  50.  51.  52.  53.  54.  55.  56.  57.  58.  59.  60.  61.  62.  63.  64.  65.  66.  67.  68.  69.  70.  71.  72.  73.  74.  75.  76.  77.  78.  79.  80.  81.  82.  83.  84.  85.  86.  87.  88.  89. | package org.stepic.java.io;  import java.io.\*;  import java.nio.file.Files;  import java.nio.file.Path;  import java.nio.file.Paths;  import java.nio.time.LocalDate;  public class SerializationDemo {  public static void main(String[] args)  throws Exception {  Client orginalClient = new Client();  originalClient.setId(1);  originalClient.setName(“Chuck  Norris”);  otriginalClient.setBirDate(  LocalDate.of(1940, 3, 10));  Path path = Paths.get(“object.bin”);  try (ObjectOutputStream oos =  new ObjectOutputStream(  Files.newOutputStream(  path))) {  oos.writeObject(originalClient);  }  Client deserializedClient;  try (ObjectIntputStream ois =  new ObjectInputStream(  Files.newInputStream(  path))) {  deseralizedClient = (Client)  ois.readObject();  }  System.out.println(  deserializedClient.getId());  System.out.println(  deserializedClient.getName(  ));  System.out.println(  deserializedClient.  getBirthDate());  System.out.println(  deserializedClient.  getAgeInYears());  }  public static class Client  implements Serializable {  private long id;  private String name;  private LocalDate birthDate;  private transient int ageInYears;  public long getId() { return id; }  public void setId(long id) {  this.id = id;  }  public String getName() {  return name;  }  public void setName(String name) {  this.name = name;  }  public LocalDate getBirthDate() {  return birthDate;  }  public void setBirthDate(  LocalDate birthDate) {  this.birthDate = birthDate;  }  public int getAgeInYears() {  if (ageInYears == 0) {  ageInYears = birthDate.until(  LocalDate.now()).  getYears();  }  return ageInYears;  }  }  } |

Делается это очень просто. Для начала надо пометить ваш класс маркерным интерфейсом Serializable. Интерфейс называют маркерным, если в нем нет ни одного метода. Вот так и Serializable ни одного метода не объявлено. Это просто признак, по которому JVM понимает, что данный объект можно сериализовывать. И эту сериализацию JVM будет делать сама по своим правилам. Писать дополнительный код для этого не требуется. Иногда бывает так, что в нашем объекте есть поля, которые не надо сохранять. Это может быть какой-то кеш или другие данные, которые легко вычисляются по тому, что есть в объекте. Их потеря ничем не грозит. Зато позволит сохранить размер, занимаемый объектом при сериализации. Такие поля примечаются ключевым словом transient. JVM будет их игнорировать при записи. Объекты можно записывать в ObjectOutputStream, который может построить из любого другого OutputStream. Запись осуществляется методом writeObject(). Объект сериализуется вместе с другими объектами, на которые он ссылается. Разумеется, они тоже должны быть помечены как Serializable. Иначе при попытке их записи будет брошено исключение. Для чтения десириализации предназначен симметричный класс ObjectIntputStream и его метод readObject(). При этом мы получаем объект и все то, на что он ссылался. Конечно, тут все не так просто, есть много нюансов и подводных камней, о которых можно рассказывать очень долго, но давайте остановимся на этом минимальноработающем примере и убедимся, что Чаку Норрису никакая сериализация нипочем. Вывод в консоль:

|  |  |
| --- | --- |
| 1.  2.  3.  4. | 1  Chuck Norris  1940-03-10  75 |

6.2 Коллекции

6.2.1 Зачем нужны коллекции

Переходим ко второй теме, к коллекциям. Коллекциями в Java называют разнообразные контейнеры однотипных элементов, что‑то вроде массивов, но гораздо удобнее и функциональнее.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10. | int[] oldArray = ...;  int oldLength = oldArray.length;  int newLength = oldLength + 10;  int[] newArray = Array.copyOf(  oldArray, newLength);  newArray[oldLength] = newElement1;  newArray[oldLength + 1] = newElement2;  // ... |

Чем плохи и неудобны массивы. Ну, например, фиксированным при создании размером. Если вдруг в процессе работы программы у нас образовалось еще несколько объектов, которые надо добавить к имеющимся, то надо выделить новый массив (5: int[] newArray) большего размера, скопировать туда элементы старого массива (5 Array.copyOf(oldArray, ...)) и после них дописать новые элементы (8‑9). А потом выкинуть старый массив, чтобы его убрал сборщик мусора. Реализовать это руками можно, но довольно муторно. Не хотелось бы писать такой код в каждой программе, где нужны структуры данных, динамически меняющие свой размер. К счастью, стандартные коллекции прекрасно решают эту задачу, скрывая все утомительные возможности внутри.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5. | final int[] array = new int[] {1, 2, 3};  array[0] = 10;  array[1] = 11;  array[2] = 12; |

Кроме того, у массивов нет никакой защиты от изменения. Если массив доступен какому‑то коду на чтение (1: array), то он точно также доступен и для записи (2‑5). Любому элементу можно присвоить новое значение. Ключевое слово final (1) не поможет, так как оно действует на ссылку на массив, а к элементам не относится. Коллекции же могут разрешать чтение, но запрещать изменение своего содержимого.

Также у коллекции есть огромное количество удобных и полезных методов, начиная от нормальных реализаций equals и toString и заканчивая методами объединения, вычитания и пересечения коллекций и так далее. А еще есть специализированные коллекции, предназначенные для многопоточного использования. С их помощью несколько потоков могут одновременно работать с одним и тем же набором объектов, не ломая его. Но тема многопоточности точно не для нашего базового курса, поэтому такие коллекции мы сейчас рассматривать не будем. Классы коллекции являются Generic и параметризуются типом хранимых внутри элементов. А мы с вами помним ограничение Generic в Java. В качестве значения Generic‑параметра может использоваться только ссылочный тип, а примитивные нет, поэтому все коллекции в Java работают только со ссылочными типами. Если нужно хранить примитивы, то есть несколько путей. Во‑первых, примитивные значения можно превратить в объекты при помощи классов‑оберток. Во‑вторых, можно руками написать какой‑то класс, похожий на стандартную коллекцию, но хранящий внутри примитивы. В‑третьих, можно позаимствовать такой класс из какой‑то сторонней библиотеки. Но в предпоследних двух подходах нужно, что придется для каждого примитивного типа копипастом создавать отдельные класс коллекции. Параметризовать это уже никак не удастся.

6.2.2 Базовый интерфейс java.util.Collection

Теперь мы пробежимся по основным классам коллекции, которые есть в стандартной библиотеке. Пробежимся, потому что этих классов очень много и у каждого своя специфика и какие особенности работы с ним. Так что мой рассказ будет только кратким обзором, но никак не исчерпывающим справочником. В дополнении к моему рассказу я рекомендую почитать Javadoc и посмотреть полные списки методов, доступных в каждой коллекции. Их заметно больше, чем я перечислил на слайдах.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19. | package java.util;  public interface Collection<E>  exstends Iterable<E> {  int size();  boolean isEmpty();  boolean contains(Object o);  boolean add(E e);  boolean remove(Object o);  void clear();  // ...  } |

Коллекции живут в пакете java.util (1). Коллекции для многопоточных программ в пакете java.util.concurrent. Но туда мы договорились не лезть. Базовый интерфейс коллекции java.util.Collection (3). Он описывает самые общие действия, применимые к любой коллекции. Здесь определен набор методов. Таких как size() (6) — текущий размер коллекции, то есть сколько элементов в ней лежит. isEmpty() (8) — проверка пустоты. По сути, эквивалентно проверки равенства размера нулю, но может быть реализовано более эффективно. contains() (10) — проверка принадлежности объекта коллекции. add() (12) — добавление элементов в коллекцию. remove() (14) — удаление элемента из коллекции. И clear() (16) — очистка и удаление всех элементов из коллекции. Методы add() (12) и remove() (14) возвращают булевское значение, говорящее о том, изменилось коллекция или нет. При вызове add() она может не измениться, если такой элемент в ней уже есть, и коллекция следит за уникальностью элементов. А при вызове remove() может оказаться что такого элемента нет. В этих случаях будет возвращен false. Методы contains() (10), remove() (14), а для некоторых типов коллекций еще и метод add() (12) используют в своей работе метод equals() для сравнения переданного объекта с тем, что уже есть в коллекции. Поэтому для тех классов, экземпляры которых вы собираетесь хранить в коллекциях, важно обеспечить адекватную реализацию equals().

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6. | Collection<ComplexNumber> c = ...;  c.add(new ComplexNumber(1, 2));  boolean contains = c.contains(  new ComplexNumber(1, 2)); |

Адекватность метода equals() зависит от задачи. Где‑то подойдет унаследованный от Object реализация, сравнивающая ссылки на равно‑равно, а где‑то нужно сравнение содержимого. Например, для знакомого нам класса ComplexNumber скорей всего потребуется переопределить equals() для сравнения содержимого, так как естественно ожидать, что после добавления числа new ComplexNumber(1, 2) (3) последующая проверка принадлежности этого числа коллекции (5) вернет true вне зависимости от того тот же этот экземпляр или другой.

6.2.3 Обход коллекции, java.util.Iterator

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | Collection<Integer> collection = ...;  Iterator<Integer> it = collection.iterator();  while (it.hasNext()) {  Integer element = it.next();  System.out.println(element);  }  for (Integer element : collection) {  System.out.println(element);  } |

У коллекции есть метод iterator() (3), унаследованный от интерфейса java.util.Iterable и возвращающий экземпляр java.util.Iterator (3). Этот интерфейс позволяет единообразно обходить элементы любой коллекции, не задумываясь о ее внутреннем устройстве будь то дерево, связанный список или массив. У Iterator есть три метода. Первый hasNext() (4) — это проверка наличия следующего элемента. Второй next() (5) — это получение того самого следующего элемента. И третий метод, не нарисованный здесь, — это метод remove(), удаляющий текущие элементы из коллекции. Текущим считается тот элемент коллекции, который был последним возвращен из next(). Что логично, так как чтобы определить удалять нам элемент или нет, надо на него сначала посмотреть. Перед вызовом next() надо обязательно проверять наличие элемента при помощи hasNext(), так как при отсутствии следующего элемента метод next() просит исключение java.util.NoSuchElementException. Кусочек кода (3‑7) — это пример обхода коллекции при помощи Ierator. Цикл for each (9) позволяет записать то же самое короче. Он поддерживает не только массивы, но и любые экземпляры java.util.Iterable. По счастливому стечению обстоятельств интерфейс java.util.Collection как раз наследуется от Iterable.

|  |  |
| --- | --- |
| 1.  2.  3. | Collection<Integer> collection = ...;  collection.forEach(System.out::println); |

Второй способ записать обход коллекции более компактно — это воспользоваться методом forEach() (3) из того же интерфейса java.util.Iterable. Метод принимает в качестве параметра экземпляр java.util.function.Сonsumer<T>. Сonsumer — это функциональный интерфейс с единственным методом accept(T). И в данном случае мы пользуемся возможностью инстанцировать функциональный интерфейс (3: System.out::println) при помощи ссылки на метод.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9. | Collection<Integer> collection = ...;  for (Integer number : collection) {  if (number > 5) {  collection.remove(number);  }  }  // java.util.ConcurrentModificationException |

Однако, если в процесс обхода нужного удалять из коллекции некоторые элементы, то не одна из сокращенных записей нам не подойдет. Придется использовать Ierator явно. Попытка вызвать метод remove() (5) на самой коллекции в процессе обхода приведет к исключению (9). Коллекции не любят, когда их в процессе обхода меняют не через итератор.

6.2.4 Список, java.util.List

Мы с вами обсудили интерфейс java.util.Collection, описывающий какую‑то совсем абстрактную коллекцию. В этом интерфейсе не специфицирован ни способ хранения элементов, ни ограничение на их уникальность, ни ожидаемая эффективность реализации тех или иных операций. Эти детали уточняются в интерфейсах наследниках и в классах, их реализующих.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20. | package java.util;  public interface List<E>  extends Collection<E> {  E get(int index);  E set(int index, E element);  void add(int index, E element);  E remove(int index);  int indexOf(Object o);  int lastIndexOf(Object o);  Last<E> subList(int fromIndex,  int toIndex);  } |

Первая разновидность коллекции — это List (3). Простой список элементов, проиндексированных числами от нуля до длины списка минус один. Лист предоставляет доступ к элементам по их индексам, совсем как массив. Один и тот же элемент может встречаться в списке несколько раз по разным индексам. Пробежимся по методам. get() (6) — это получение элемента по индексу. set() (8) — это замена элемента по индексу. add() (10) — это добавление элемента по индексу. При этом элементы, начиная с указанного индекса (10: index) сдвигаются на одну позицию вправо и на освободившееся место встает новый элемент. Метод add(), унаследованный от коллекции, добавляет элемент в конец списка. Метод remove() (12) — удаляет элемент по индексу и возвращает его (12: E). Методы indexOf() (14) и lastIndexOf() (16) ищут элемент в данном списке и возвращают его индекс. Либо первый с начала (14), либо первый с конца (16), и если элемент не найден, то они вернут –1. И, наконец, метод subList() (18) возвращает часть списка в виде отдельного экземпляра List . Но важно, что изменение одного списка будут видны в другом, то есть это не копирование.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | List<String> words = ...;  words.subList(1, 3).clear();  int indexOfFoo = 10 +  words.subList(10, 15).indexOf("foo");  words.equals(...); |

Метод subList() позволяет реализовывать более сложные операции, которые не вынесены в интерфейс List в явном виде. Например, удаление элементов с индексами в диапазоне от 1 до 3 (3). Или поиск элемента среди элементов с индексами в диапазоне от 10 до 15 (5). Надо только не забыть отдельно отработать случай, когда возвращается –1 — признак того, что элемент не найден. Метод equals() у списка специфицирован таким образом, что он сравнивает списки по содержимому, то есть списки равны, если они содержат одни и те же элементы в одинаковом порядке (8).

|  |  |
| --- | --- |
| 1.  2.  3. | List<String> list1 = new ArrayList<>();  List<Integer> list2 = new LinkedList<>(); |

Общеупотребительных реализаций List две. Первое — это ArrayList<> (1), которая реализует List поверх массива. Дает максимально дешевый доступ к элементу по индексу, почти такой же дешевый как в обычном массиве. Добавление или удаление элементов уже не такое дешевое, так как может приводить к перевыделению массива или как минимум к сдвигу элементов в массиве. Вторая реализация в LinkedList (3). Основана на двусвязном списке. Позволяет эффективно добавлять и удалять элементы в начале или конце списка, так как не надо перемещать большие куски памяти или перевыделять массивы. Зато доступ по индексу дороже, так как чтобы получить элемент по индексу n требуется сделать n прыжков по указателям от начала списка. Сразу замечу, что при объявлении переменной лучше пользоваться максимально общим интерфейсом (1,3: List), а конкретный класс указывать только вместе с созданием экземпляра (1,3: ArrayList<>, LinkedList<>). Так можно будет в любой момент заменить реализацию коллекции, исправив всего одно место в программе. Это важно еще в том случае, когда вы пишите библиотечные методы для использования другими программистами. Входные параметры должны иметь максимально общий тип, чтобы один и тот же метод можно было бы переиспользовать с разными реализациями коллекций.

6.2.5 Очередь, java.util.Queue

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14. | package java.util;  public interface Queue<E>  extends Collection<E> {  boolean add(E e);  boolean offer(E e);  E remove();  E poll();  E element();  E peek()  } |

Вторая разновидность коллекций — это Queue (3), java.util.Queue, очередь. Это стандартная структура данных, обеспечивающая порядок FIFO (first in — first out) или первый вошел — первый вышел. Элементы в очередь добавляются с хвоста и извлекаются из головы очереди. Методы add() (6) и offer() (7) добавляет элемент в хвост очереди. Разница между ними следующая: если очередь ограничена по размерам и уже достигла этого максимального размера, то add() кидает исключение, а offer() просто возвращает false. Методы remove() (9) и poll() (10) извлекают первый элемент из головы очереди. В случае если очередь пуста remove() бросит исключение, а poll() вернет null. Методы element() (12) и peek() (13) позволяют подсмотреть элемент в голове очереди, не удаляя его. Разница между ними точно такая же: element() бросит исключение, peek() вернет null.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13. | package java.util;  public interface Deque<E> extends Queue<E> {  void addFirst(E e);  void addLast(E e);  boolean offerFirst(E e);  boolean offerLast(E e);  E removeFirst();  E removeLast();  } |

Наследник очереди — интерфейс java.util.Deque (3). Он отличается от очереди тем, что можно добавлять и удалять элементы как в хвосте, так и в голове Deque. Количество методов соответственно удваивается по сравнению с очередью. Каждую операцию можно делать в голове (5,8,10) или в хвосте (6,9,12) Deque.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13. | Deque<Object> deque1 = new ArrayDeque<>();  Deque<Integer> deque2 = new LinkedList<>();  deque2.offerLast(1);  deque2.offerLast(2);  deque2.offerLast(3);  Integer element;  while ((element = deque2.pollFirst())  != null) {  System.out.println(element);  } |

Реализации интерфейса Deque — две. Во‑первых, это ArrayDeque<> (1), Deque поверх массива. А, во‑вторых, как ни странно, это класс LinkedList<> (3), то есть класс LinkedList<> реализует сразу несколько интерфейсов. Он является одновременно и списком и Deque.

6.2.6 Множество, java.util.Set

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | package java.util;  public interface Set<E>  extends Collection<E> {  // ...  } |

Третья разновидность коллекции — это java.util.Set (3), множество. Это коллекция обеспечивает уникальность содержащихся в ней элементов, то есть Set не может содержать двух элементов, равных друг другу. При попытки добавить дубликат методом add() множество изменено не будет. Как раз в этом случае метод вернет false. Интерфейс Set не добавляет новых методов относительно интерфейса Collection. Он только уточняет семантику существующих в родительском интерфейсе методов. Поэтому если вы откроете объявление интерфейса Set в стандартной библиотеке, то вы увидите в нем все те же методы, что и в коллекции, но с другим Javadoc. Например, метод equals() определен так, что он будет сравнивать два множества по содержимому.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10. | Set<ComplexNumber> numbers = new HashSet<>();  numbers.add(new ComplexNumber(3, 3));  numbers.remove(new ComplexNumber(3, 3));  // equals(),hashCode()  a.equals(b);  a.hashCode() == b.hashCode(); |

Реализаций интерфейса Set несколько. Первая из них — это HashSet (1): реализация множества на основе хеш‑таблицы. Важную роль тут играет метод hashCode(). HashSet использует хеш‑код объекты для определения ячейки хеш‑таблицы, куда нужно положить этот объект. А метод equals() затем используется для сравнения с элементами, уже лежащими в этой ячейке хеш‑таблицы. Очень важно, чтобы реализация hashCode() и equals() были согласованными, то есть если (9), то (10). Если это соотношение нарушается, то HashSet будет работать с элементами неправильно, например вы будете добавлять элементы в HashSet, а потом их не находить. Или добавлять один и тот же с точки зрения equals() элементы, а он по факту добавиться много раз в разные ячейки хеш‑таблицы. И еще важный момент. Объект не должен меняться пока он лежит в HashSet, точнее не должны меняться его поля, влияющие на вычисление equals() и hashCode(). В крайнем случае можно удалить объект из HashSet, поменять его и добавить обратно, чтобы он добавился уже с новым хеш‑кодом и попал в новую ячейку хеш‑таблицы.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | Set<String> words1 = new HashSet<>();  words1.add(...);  for (String s: words) {...}  Set<String> words2 = new LinkedHashSet<>();  words2.add(...); |

Порядок обхода элемента HashSet итератором обусловлено хеш‑кодом элементов, точнее раскладка объектов по хеш‑таблицам, которые определяются хеш‑кодами. Если мы попробуем при помощи итератора обойти все элементы HashSet (4), то порядок обхода для стороннего наблюдателя будет казаться случайным. Если порядок элементов важен, то предлагается использовать другую реализацию Set — LinkedHashSet (6). Он отличается тем, что провязывает все элементы в список в порядке их добавления, что позволяет итератору обходить и возвращать все элементы в том порядке, в котором вы добавили Set.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16. | package java.util;  public interface SortedSet<E>  extends Set<E> {  SortedSet<E> subSet(E fromElement,  E toElement);  SortedSet<E> headSet(E toElement);  SortedSet<E> tailSet(E fromElement);  E first();  E last();  } |

От интерфейса Set наследуется интерфейс SortedSet (3), упорядоченное множество. Благодаря упорядоченности он обеспечивает обход элементов итератором в порядке возрастания элементов. А также поддерживает некоторые дополнительные операции. Например, headSet() (9) возвращает подмножество элементов, меньших чем преданный параметр (9: toElement). И это не копия, а живое множество, которое изменяется вместе с основным. И аналогичные ест методы tailSet (11), subSet (6), а также наименьший (13) и наибольший (15) элемент.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | SortedSet<String> words = new TreeSet<>();  words.add("aaa");  words.add("bbb");  words.add("ccc");  words.headSet("bbb").clear(); |

Реализация интерфейса SortedSet — это класс TreeSet (1). Внутри него самобалансирующийся двоичное дерево поиска, а именно красно‑черное дерево. Естественно, что TreeSet должен уметь сравнивать элементы друг с другом. Для этого либо элементы должны реализовывать интерфейс java.lang.Comparable<T> с методом compareTo(T), либо в конструктор TreeSet должен передаваться экземпляр java.util.Comparator<T> с методом compare(T, T), принимающем два параметра. То есть мы можем сравнивать объекты, которые сами не умеют сравниваться друг с другом.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10. | List<String> list = new ArrayList<>();  list.add("aaa");  list.add("aaa");  list.add("bbb");  list.add("aaa");  Set<String> set = new LinkedHashSet<>(list);  List<String> listWithoutDups =  new ArrayList<>(set); |

Поучительный пример эффективного использования коллекций — это удаление дубликатов из списка. Здесь мы пользуемся тем, что по соглашению у каждого класса коллекции есть конструктор, принимающий в качестве параметра другую коллекцию. При этом все элементы исходной коллекции копируются в новую. Поэтому мы просто создаем новый LinkedHashSet (7) от нашего исходного списка, который удалит дубликаты и при этом сохранит порядок следования элементов. А потом создаем обратно список из множества (10).

6.2.7 Отображение, java.util.Map

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20. | package java.util;  public interface Map<K, V> {  int size();  boolean isEmpty();  boolean containsKey(Object key);  boolean containsValue(Object value);  V get(Object key);  V put(K key, V value);  V remove(Object key);  void clear();  Set<K> keySet();  Collection<V> values();  Set<Map.Entry<K, V>> entrySet();  } |

И четвертый вид коллекции — это отображение или ассоциативный массив, представленный интерфейсом java.util.Map (3). Строго говоря, он не наследует интерфейс Collection, но идеологически тоже относится к коллекциям. В отличии от массивов, где элементы индексируются числами, Map позволяет индексировать элементы произвольными объектами. Например, строками, элементами перечисления, комплексными числами или чертом в ступе. Основные методы интерфейса Map такие. Это size() (5) — размер, количество пар ключ‑значение, хранящихся в данный момент. isEmpty() (6) — проверка пустоты. Метод containsKey() (8), проверяющий есть ли сейчас в Map такой ключ. Метод containsValue() (9), проверяющий есть ли такое значение. Метод get() (11: key), получающий значение по ключу (11). Метод put() (12), сохраняющий значение по ключу. Метод remove() (14), удаляющий пару ключ‑значение с заданным ключом (14: key). И метод clear() (15) — очищающий Map. Методы put() (12) и remove() (14) возвращают значения, которое было по данному ключу перед выполнением запрошенной операции. Поскольку интерфейс Map не расширяет Collection и не расширяет Iterable, то встает вопрос как его обходить. Тут все зависит от ваших потребностей. Если нужны только ключи, то пользуйтесь методом keySet() (17), который вернет вам Set (17: Set<K>), то есть множество ключей. Это множество, потому что дубликаты ключей в Map не допускаются. Если вам нужны только значения, то пользуетесь методом values() (18), который вернет вам коллекцию всех значений (18: Collection<V>). Это коллекция не множество, потому что в общем случае про уникальность значений нам ничего не известно. Если при обходе вам одновременно требуются и ключ, и значение, то пользуйтесь методом entrySet() (19), который вернет вам множество (19: Set<>), Entry (1: Entry<>), где Entry — это простенький объект, содержащий пару ключ‑значение.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17. | Map<A, B> map = new HashMap<>();  for (A key : map.keySet()) {...}  for (B value : map.values()) {...}  for (Map.Entry<A, B> entry :  map.entrySet()) {  System.out.printf("%s => %s\n",  entry.getKey(),  entry.getValue());  }  map.forEach((k, v) -> System.out.printf(  "%s => %s\n", k, v));  // java.util.function.BiConsumer<T, U> |

Примеры использования эти трех методов я собрал на этом слайде. (3) обходим все ключи. (5) обходим все значения. А (8) —все пары ключ‑значение. Также я привел еще один пример с использованием метода forEach (14), который принимает экземляр (17). Это функциональный интерфейс с методом accept(T, U) с двумя параметрами: первый типа T, второй типа U. И мы передаем в качестве экземпляра этого BiConsumer вот такое лямбда‑выражение (14).

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10. | Map<String, String> map1 = new HashMap<>();  map1.put("foo", "bar");  map1.put("bar", "baz");  map1.remove("bar");  SortedMap<String, String> map2 =  new TreeMap<>();  map2.put("foo", "bar");  map2.put("bar", "baz");  map2.subMap("bar", "foo").clear(); |

Что касается реализации интерфейс Map. У нас есть HashMap (1), есть LinkedHashMap и есть TreeMap (7). Их устройство и отличия друг от друга абсолютно аналогично соответствующим реализациям интерфейса Set.

|  |  |
| --- | --- |
| 1.  2.  3.  4. | java.util.Vector  java.util.Stack  java.util.Dictionary  java.util.Hashtable |

В стандартной библиотеке есть еще несколько классов‑коллекций, которые еще появились в версии Java 1.0 и в настоящий момент не рекомендуются к использованию. Они оставлены лишь для обратной совместимости, а их функционал полностью покрывается современными коллекциями, которые мы с вами рассмотрели.

6.2.8 Преобразование в массив и обратно

|  |  |
| --- | --- |
| 1.  2.  3. | Collections.shuffle(list);  Collections.sort(list); |

В работе с коллекциями иногда помогает утилитный класс java.util.Collections, в котором есть методы, например, для перетасовывания списка (1) или для сортировки списка (3) и многие другие.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | Set<String> set =  Collections.unmodifiableSet(  originalSet);  set.remove("abc");  // throws java.lang.  UnsupportedOperationException |

В классе java.util.Collections есть также семейство методов с именами, начинающимися на unmodifiable. В частности, unmodifiableSet() (2), unmodifiableList(), unmodifiableMap() и так далее. Эти методы возвращают объект‑обертку (1: Set<String>), реализующую тот же интерфейс, что и переданный параметр (3: originalSet), и предоставляющий доступ на чтение к элементам исходной коллекции. Но при этом любая попытка изменить содержимое коллекции (5) через этот объект‑обертку приведет к выбросу (6). То есть мы можем при передаче коллекции в какой‑то метод защититься от его случайного там изменения.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6. | List<Integer> list = ...;  Object[] array1 = list.toArray();  Integer[] array2 =list.toArray(  new Ineger[list.size()]); |

Периодически приходится конвертировать коллекции в классический массив. Для этого в интерфейсе Collections есть метод toArray() (3), который возвращает содержимое коллекции в виде массивов Object[] (3). Это связано с ограничением Generic в Java, о котором мы с вами подробно говорили. Внутри параметризованного класса нельзя создавать экземпляры или массивы типа (1: Integer), указанного в качестве параметра Generic. Если нам хочется получить массив того типа (5: Integer), элементы которого действительно лежат в коллекции, то надо воспользоваться другим перегруженным вариантом метода toArray() (5), который принимает этот массив правильного типа (6: new Ineger[list.size()]). Если массив, подходящего размера, то метод toArray() сложит все элементы в этот самый переданный массив и его же и вернет. Если переданный массив не того размера, которого нужно, то метод toArray() внутри себя перевыделит массив правильного размера и вернет уже новый массив.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | String[] array = {"A", "B", "C"};  Set<String> set1 =  new HashSet<>(Arrays.asList(array));  Set<String> set2 = new HashSet<>();  Collections.addAll(set2, array); |

Выполнить обратную конвертацию из массива в коллекцию тоже не трудно. Можно, например, превратить массив в список при помощи метода Arrays.asList() (4), а потом передать результат в конструктор любого другого класса коллекции (4: new HashSet<>()). Или можно пойти другим путем и создать сначала пустую коллекцию (6), а потом при помощи утилитного метода Collections.addAll() (7) добавить туда все элементы из массива (7: array).

6.3 Функциональные интерфейсы

6.3.1 Зачем нужны функциональные интерфейсы?

В модуле, посвященному объектно-ориентированному программированию, мы с вами затронули тему функциональных интерфейсов и связанных с ними ссылок на методы и лямбда‑выражения. Но тогда нас больше всего интересовало в словосочетании функциональный интерфейс второе слово — интерфейс. Мы их рассматривали именно в этом разрезе. Теперь давайте сконцентрируемся на первом слове — функциональный.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10. | package java.util;  @FunctionalInterface  public interface Comparator<T> {  int compare(T o1, T o2);  // any number of default or static  methods  } |

Напомню, что интерфейс называется функциональным, если в нем ровно один абстрактный метод. Подчеркнуть это и дополнительно проверить на стадии компиляции можно при помощи аннотации @FunctionalInterface. Дефолт методы и статические методы при этом не в счет. Их в функциональном интерфейсе может быть сколько угодно, как и статических полей.

Зачем в язык были добавлены функциональные интерфейсы. Дело в том, что в Java нет указателей на функцию как в C или C++ или делегатов как в C#. Поэтому, когда в Java требуется передать в метод какой-то кусочек программной логики, то метод объявляется как принимающий экземпляр какого-то класса или интерфейса. А в месте вызова этого метода такой экземпляр нужно как-то создать, зачастую объявив там же анонимный класс с подходящей реализацией метода.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10. | File directory = ...;  File[] javaSourceFiles = directory.listFiles(  new FileFilter() {  @Ovveride  public boolean accept(File file) {  return fie.getName().endsWidth(  ".java");  }  }); |

Например, условия для фильтрации файлов, возвращаемых методом listFiles() до Java 7 включительно приходилось записывать в виде 4‑10. Здесь все кроме одной строчки избыточно. Содержательная часть, логика фильтрации, — это только одно выражение 7‑8, а все остальное технические детали, шелуха. Конечно, это очень неудобно и громоздко, поэтому Java‑программисты много лет смотрели на другие языки и завидовали тому, что в них есть лямбда‑выражения или замыкания, какой‑то компактный синтаксис для передачи куска кода в качестве параметра в другой код. Есть даже шутка, что Java сделала замыкание популярными, не поддерживая их.

6.3.2 Стандартные функциональные интерфейсы java.util.function

И вот, наконец, в Java 8 было введено такое понятие как функциональный интерфейс. Именно благодаря функциональным интерфейсам стало возможно избавиться от монструозного синтаксиса с анонимными классами и использовать компактные лямбда‑выражения и ссылки на методы. Это открыло возможность писать на Java программы в функциональном стиле, то есть в стиле, характерном для функциональных языков программирования. Это когда программа записывается как последовательное применение функций к некоторым значениям и другим функциям, а не как сложная структура из циклов, условных операторов и перекладывания значений переменных туда‑сюда.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | package java.io;  @FunctionalInterface  public interface FileFilter {  Boolean accept(File pathname);  } |

Давайте поподробнее разберем эти нововведения языка. Ну а объявление функционального интерфейса никаких проблем не представляет, и мы уже знаем, как это делать. Не забывайте только ставить аннотацию @FunctionalInterface, чтобы компилятор валидировал, что интерфейс действительно является функциональным. Интерфейсы FileFilter или, например, java.util.Comparator, который я тоже уже упоминал, существуют в Java давно. Просто в 8 они стали функциональными и получили аннотацию @FunctionalInterface. Однако, в Java 8 появилось целое семейство, целый пакет функциональных интерфейсов самого общего вида на самые разные случаи жизни. Когда вы понимаете, что для решения задач вам нужно передавать куда‑то кусок программной логики, то вместо того, чтобы придумывать свой функциональный интерфейс, посмотрите сначала на стандартный. Очень вероятно, что подходящий вам интерфейс там найдется, не надо будет выдумывать велосипед. И другие программисты вам скажут спасибо, поскольку вы будете говорить с ними на одном стандартном общепринятом языке.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | package java.util.function;  @FunctionalInterface  public interface Consumer<T> {  void accept(T t);  } |

Эти стандартные интерфейсы живут в пакете java.util.function. Их более 40, но они четко делятся на 5 семейств. Я расскажу в целом о семействах, не пытаясь перечислить все интерфейсы поименно. Первое семейство — это Consumer<T> или потребители. Это те, кто принимают какое‑то значение, но ничего не возвращают взамен. Этот Consumer<T> — это дженерик. Он параметризован типом объекта, который он будет принимать. А есть еще отдельные интерфейсы IntC, LongC и DoubleC, принимающие значение соответствующих примитивных типов. Они существуют потому, что дженерики в Java не могут параметризоваться примитивами. Так что разработчикам стандартной библиотеки пришлось здесь пойти на определенный копипаст. А еще бывают Consumer, принимавшие по 2 параметра. Один из них — это интерфейс BiConsumer<T, U> с двумя дженерик параметрами и соответственно его метод accept() принимает 2 параметра указанных типов и ничего не возвращает: void accept(T, U).

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | package java.util.function;  @FunctionalInterface  public interface Supplier<T> {  T get();  } |

Второе семейство — это Supplier, поставщики. Они не принимают ничего в качестве параметра, а просто возвращают какое‑то значение. Это у нас дженерик Supplier, параметризованный тем типом, который мы возвращаем. А есть еще отдельно BooleanS, IntS, LongS и DoubleS.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | package java.util.function;  @FunctionalInterface  public interface Predicate<T> {  boolean test(T t);  } |

Третье семейство — это Predicate. Их функциональны метод принимает значение какого‑то типа, а наружу отдает булевское значение — истину или лож. Есть специализированные Predicate: IntP, LongP и DoubleP. А также есть разновидность Predicate, принимающего сразу 2 параметра — это бинарный Predicate — BiPredicate.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | package java.util.function;  @FunctionalInterface  public interface Function<T, R> {  R apply(T t);  } |

Четвертое семейство, самое плодовитое, — это функции, Function. Function принимает аргумент и возвращает значение какого‑то типа. В общем случае эти типы разные. Бывают функции, принимающие два параметра. Это интерфейс BiFunction. А также есть масса специализаций функции, где на месте одного или обоих параметров стоят примитивы. Например, DoubleF — это функция, принимающая double и возвращающая объект T. Или LongToIntF — это функция, принимающая long и возвращающая int. Или ToIntF — это функция, принимающая объект T и возвращающая int. И таких специализаций еще много.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | package java.util.function;  @FunctionalInterface  public interface UnaryOperator<T>  extends Function<T, T> {  // apply is inherited from Function  } |

А пятое семейство — это операторы. Оператор — это частный случай функции, когда на входе и на выходе значение одного и того же типа. Унарный оператор (UnaryOperator) принимает один параметр, а бинарный оператор (BinaryOperator) — 2 параметра. Заведены отдельные интерфейсы под унарные и бинарные операторы над примитивными int, long, и double.

Эти функциональные интерфейсы, конечно, не покрывают все возможные потребности, которые могут возникнуть в жизни. В тех случаях, когда подходящего интерфейса в стандартной библиотеке нет, например, функция от 3‑х параметров, то придется такой интерфейс написать самостоятельно. Но вся стандартная библиотека обходится этими перечисленными стандартными функциональными интерфейсами.

6.3.3 Инстанцирование функциональных интерфейсов

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | class IntSquare implements IntUnaryOperator {  @Ovveride  public int applyAsInt(int operand) {  return operand \* operand;  }  } |

Функциональные интерфейсы можно инстанцировать 3 способами. Во‑первых, можно честно завести именованный или анонимный класс, реализующий интерфейс. Но это громоздко. Если данный класс не будет переиспользоваться, то совершенно излишне. К тому же тут нет ничего интересного или нового, на чем хотелось бы останавливаться подробнее.

Во‑вторых, можно использовать лямбда‑выражения.

|  |  |
| --- | --- |
| 1.  2.  3. | IntUnaryOperator square = x -> {  return x \* x;  }; |

Компилятор знает ожидаемый интерфейс и поэтому не требует от вас повторять имя этого интерфейса, а также имя метода, типа параметров и тип возвращаемого значения справа от знака равенства. Вы просто объявляете имена параметров и тело метода, непосредственно исполняемый код.

|  |  |
| --- | --- |
| 1. | IntConsumer print = x -> System.out.print(x); |

Если тело метода состоит из единственного выражения, и это не return, то фигурные скобки вокруг тело метода можно опустить,

|  |  |
| --- | --- |
| 1. | IntUnaryOperator cube = x -> x \* x \* x; |

а если это был return, то вместе со скобками убираем и ключевое слово return тоже.

Интересный вопрос: к каким переменным и как можно обращаться в теле лямбда‑выражения.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6. | pubic class Demo {  public void foo() {  IntUnaryOperator square =  x -> x \* x;  }  } |

Очевидно, что можно обращаться к параметрам лямбды, а также свободно объявлять и использовать внутри тела лямбда‑выражения любые переменные.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9. | pubic class Demo {  private int counter;  public void foo() {  IntSupplier sequence =  () -> counter++;  }  } |

Менее очевидно то, что можно обращаться к полям того класса, внутри которого объявлена лямбда. Причем их можно как читать, так и писать.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | pubic class Demo {  public void foo() {  int bonus = 10;  IntUnaryOperator bonusAdder =  (x) -> x + bonus;  }  } |

И наконец, можно обращаться к переменным, объявленным внутри того метода, где создана лямбда. Но здесь есть ограничения. Переменные должны быть эффективно‑финальными, то есть значения им должны быть присвоено ровно один раз до создания лямбды, после чего меняться оно уже не может. Это то же самое, как если бы мы записали ключевое слово final и его можно написать, но компилятор этого не требует.

|  |  |
| --- | --- |
| 1.  2.  3. | int[] counter = new int[] {0};  IntSupplier sequence = () -> counter[0]++; |

Отсюда, в частности, следует, что из лямбда нельзя присваивать новое значение переменным содержащего ее метода. Для обхода этого ограничения иногда используют трюк с массивом единичной длины. Ссылка на массив является эффективно финальной, однако на содержимое массива это ограничение не распространяется и его можно обновлять.

И, в‑третьих, функциональный интерфейс можно инстанцировать при помощи ссылки на метод. Это самый простой на вид способ, но у него есть несколько нюансов.

|  |  |
| --- | --- |
| 1.  2. | ToIntFunction<String> intParser =  Integer::parseInt; |

Мы можем ссылаться на статический метод, указывая имя класса, 2 двоеточия, имя статического метода.

|  |  |
| --- | --- |
| 1.  2. | Consumer<Object> printer =  System.out:println; |

Или мы можем ссылаться на нестатический метод, указывая конкретный объект, 2 двоеточия, имя нестатического метода.

|  |  |
| --- | --- |
| 1.  2. | Function<Object, String> objectToString =  Object::toString; |

Еще один способ сослаться на нестатический метод — это указать имя класса, 2 двоеточия, имя нестатического метода, но тогда первый передаваемый в месте вызова функционального интерфейса параметр будет тем самым объектом, на котором данный нестатический метод будет вызван.

|  |  |
| --- | --- |
| 1.  2. | IntFunction<String[]> arrayAllocator =  String[]::new; |

А еще можно ссылаться на конструктор. Тогда пишут имя класса, 2 двоеточия и ключевое слово new.

6.3.4 Дополнительные возможности

У стандартных функциональных интерфейсов, кроме собственно главного абстрактного метода, есть еще интересные статические и дефолтные методы, позволяющие делать разные полезные вещи.

|  |  |
| --- | --- |
| 1.  2. | IntPredicate isOdd = x -> x % 2 != 0;  IntPredicate isEven = isOdd.negate(); |

Например, если у нас есть некоторый предикат, то обратный ему можно получить при помощи метода negate().

|  |  |
| --- | --- |
| 1.  2. | IntPredivate p1 = ..., p2 = ...;  IntPredicate p3 = p1.and(p2); |

Или если у нас есть 2 предиката, то мы можем построить их конъюнкцию при помощи метода and.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | Consumer<Object> printer =  System.out::println;  List<Object> objects = new ArrayList<>();  Consumer<Object> collection = objects::add;  Consumer<Object> combinedConsumer =  printer.andThen(collectior); |

Если у нас есть 2 Consumer, то мы можем их скомбинировать в новый 3-ий Consumer, который вызовет сначала первое действие, а затем второе. То есть в данном случае сначала напечатает объект в консоль, а потом добавит в список.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | DoubleUnaryOperator sqare = x -> x \* x;  DoubleUnaryOperator sin = Math::sin;  DoubleUnaryOperator complexFunction1 =  sin.andThen(square);  DoubleUnaryOperator complexFunction2 =  sin.compose(square); |

Похожим образом можно строить композиции функций при помощи методов andThen и compose(). Разница между этими методами в порядке применения. Метод andThen() сделает нам функцию , а метод compose() сделает нам функцию .

И в заключении еще один пример. Кажется, самый плодовитый на методы функциональный интерфейс — это компаратор. И с его помощью можно сделать следующее.

|  |  |
| --- | --- |
| 1.  2.  3. | Comparator<Double> absoluteValueComparator =  (a, b) -> Double.compare(  Math.abs(a), Math.abs(b)); |

Вместо явного описания алгоритма сравнения 2-х double по их абсолютной величине,

|  |  |
| --- | --- |
| 1.  2.  3. | Comparator<Double> absoluteValueComparator2 =  Comparator.comparing(  Math::abs, Double::compare); |

мы можем вызвать метод comparing(), которому передать первым параметром метод, который будет применяться к каждому значению перед тем как выполняется сравнение, и второй параметр — это метод, который будет сравнивать результаты применения первого метода к исходным значениям, чтобы получить финальный результат.

6.4.2 Пример и общая схема

|  |  |
| --- | --- |
| 1. | int sum = IntStream.iterate(1, n -> n + 1) |

Метод iterate() в классе IntStream принимает число, это первый элемент последовательности. И вторым аргументом функцию, которая вычисляет следующий элемент последовательности по предыдущему. Возвращается Stream, тот самый IntStream. Он представляет собой бесконечную последовательность целых чисел, начиная с единицы. Но вычисляется она еще не начала, потому что стримы ленивы и нечего не делают пока их не припрут к стенке. Дальше каждый следующий метод в цепочке добавляет какое‑то преобразование и возвращает преобразованный стрим.

|  |  |
| --- | --- |
| 3. | .filter(n -> n % 5 == 0 && n % 2 != 0) |

Здесь мы фильтруем стрим, оставляем только числа, которые делятся на 5 и не делятся на 2.

|  |  |
| --- | --- |
| 5. | .limit(10) |

Затем из бесконечной последовательности берем только первые 10 элементов

|  |  |
| --- | --- |
| 7. | .map(n -> n \* n) |

и возводим каждый из них в квадрат.

|  |  |
| --- | --- |
| 9. | .sum(); |

И наконец, суммируем все элементы и получаем число‑результат.

Последняя операция суммирования является спусковым крючком для того, чтобы стрим начал выполнять все те действия, которые мы ему передали на предыдущих шагах. Здесь стриму приходится перебороть свою лень и таки вычислить результат.

И заметьте: информация о том, что нам нужны только первые 10 элементов дошла до источника данных, то есть до генератора последовательности целых чисел. И он будет работать не бесконечно, а только пока не наберет нужное количество, удовлетворяющих предикату элементов. Использование стримов всегда состоит из 3‑х частей. Первая часть — это получение стрима, откуда будут браться элементы последовательности. Есть много разных способов получить стрим. Вторая часть — это 0 или более промежуточных операций преобразований. Стрим их просто запоминает, но пока не выполняет. И третья часть — это единственная терминальная операция, запускающая весь процесс вычислений, и которая собственно должна стать его полезным результатом.

6.4.5 Терминальные операции

|  |  |
| --- | --- |
| 1.  2.  3. | Stream<BigInteger> bigInts = ...;  BigInteger sum = bigInts.reduce(  BigInteger.ZERO, BigInteger::add); |

И наконец, терминальная операция reduce() позволяет вычислить свертку элементов стрима, то есть результат применения некоторого бинарного оператора к каждой паре элементов стрима пока от стрима не останется один единственный элемент. Это и есть результат свертки, и он возвращается в качестве результата из метода reduce(). Если стрим был пуст, то возвращается некоторое нулевое значение.

Важно, что вызвать на стриме терминальную операцию можно только один раз. После вызова терминальной операции стрим считается более непригодным к использованию и этот объект остается только выбросить. Если к тем же элементам нужно применить еще какое‑то преобразование, посчитать по ним какое‑то другое значение, то надо заново сконструировать стрим, снова настроить его и запустить какой‑то терминальной операцией.

6.4.6 Два примера

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9. | public static BigInteger factorial(int n) {  return IntStream.rangeClosed(1, n)  .mapToObj(i -> BigInteger.valueOf(i))  .reduce(BigInteger.ONE,  BigInteger::multiply);  } |

На закуску приведу пару примеров того, как написать при помощи стримов известные нам уже алгоритмы. Например, алгоритм вычисления факториала. Здесь мы можем породить интервал целых чисел от 1 до n в виде стрима. Затем каждое из этих чисел превратить в BigInteger, а затем вычислить свертку при помощи операции умножения. Результат будет тем же самым факториалом, который мы с вами вычисляли в одном из домашних заданий.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19. | public static boolean isPalindrome(  String s) {  StringBuilder leftToRight =  new StringBuilder();  s.chars().filter(  Character::isLetterOrDigit)  .map(Character::toLowerCase)  .forEach(  leftToRight::appendCodePoint);  StringBuilder rightToLeft =  new StringBuilder(  leftToRight).reverse();  return leftToRight.toString()  .equals(rightToLeft.toString());  } |

И второй пример определение того, что строка является палиндромом, то есть одинаково читается слева направо и справа налево. Основная сложность здесь это отфильтровать символы строки, оставить только буквы и цифры и проверять палиндромность уже результирующей строки. Эту задачу мы можем решить следующим образом: получить стрим символов из строки, отфильтровать только те, которые является только буквами или цифрами, привести каждый из них в нижний регистр и сохранить результат в StringBuilder. Ну а дальше остается реверсировать и сравнить 2 строки.