Язык Ява

Михаил Балкаров

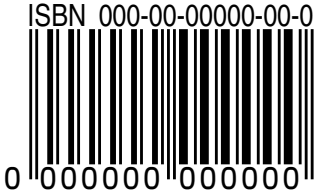
Издание 0, 2019

ББК 00.000
Б00
УДК 000.00 -

Б00 **Балкаров М. А.**
Язык Ява —
Москва.: Изд-во xxxx, 2020 — 000 с.
ISBN 000-00-00000-00-0

Рассматривается язык Ява с точки зрения профессионального программиста.
Для широкого круга читателей, желающих ознакомится с языком Ява и его библиотеками. Предполагается знание основ программирования.

Формат 60 × 90/16. Усл. печ. л. 00. Бумага офсетная №1.
Гарнитура Libertino.



Предисловие которое стоит прочитать

Перед вами, очередная книга по языку Java. Таких книг сотни, если не тысячи, есть вполне прилично переведенные да и изначально написанные на русском. В чем причина появления и отличия именно этой? Автору потребовались материалы для учебного курса и он не долго думая оформил их именно в таком виде. Благо опыт написания и издания у него накоплен уже достаточно богатый[2, 4, 1]. Есть уже даже курс по обучению программированию, тоже оформленный в виде книги [3].

Нужна ли Вам эта книга? По факту это конспекты вводного курса по языку Ява. В соответствии с задачами курса автор описывает реализацию на Java базовых концепций алгоритмического, классового и функционального подхода к программированию. Для обучения программированию не самый лучший материал, впрочем как и язык, при всех своих достоинствах, не для начинающих.

Конечно Ява на порядок превосходит тот же не к ночи будет помянутый Pascal или C, в качестве первого языка для изучения, но по субъективному мнению автора учится нужно на действительно простых, но при этом мощных языках, вроде Python или SmallTalk. При всей специфичности последнего, textenglishSmallTalk по прежнему отличное средство прототипирования больших проектов. Ну а Python в топе популярности среди языков общего назначения и притом четвертого поколения. Ява при всех своих несомненных достоинствах и нахождении в том же топе, все же третье поколение лишь с элементами четвертого.

Цели курса и соответственно книги, научить:

- ориентироваться в Java технологиях;
- работать с примитивными типами Java;
- использовать ветвление и циклы Java ;
- создавать методы на Java;
- работать с массивами и со строками Java;
- разрабатывать классы на Java;
- использовать наследование и полиморфизм;
- разрабатывать абстрактные классы и интерфейсы;
- обрабатывать ошибки, возникающие в программе;
- работать с файловой системой;
- использовать новые возможности классов из пакета NIO 2;
- пользоваться коллекциями Java;
- использовать в программе параметризованные типы Java;
- применять знания о паттернах проектирования

К сожалению, учитывая традиционно сжатые сроки, при подготовке к печати не удалось как следует вычитать текст, поэтому грамотность, верстка и язык местами слегка хромают. Опять же, в силу ограниченности автора и времени, раскрытие затронутых тем, в некоторых местах книги было сознательно оборвано. Что из этого в итоге получилось, судить конечно читателю.

Пожелания к читателю

Автор предполагает, что читатель знает программирование. Это не означает, что требуются знания хакеров, да и знания хакеров, это в основном социальная инженерия, а вовсе не хитрые программы. Не требуются даже большой опыт кодирования или знания конкретного языка. Основная причина такого пожелания, то что в этой книге не объясняются сами принципы программирования и термины, кроме специфичных для языка.

Разумеется и общая компьютерная грамотность предполагается. На компьютере выполняются программы, делающие или не делающие что то определенное. Программы и данные при этом хранятся в файлах на диске. Поставляются программы в виде дистрибутивов, тоже файлов. Файлы эти можно создавать, редактировать или уничтожать с помощью других программ. Файлы для поддержания порядка, лежат в папках-директориях. Программы можно запускать на исполнение компьютером. Запуская дистрибутив устанавливают новые программы. И все это можно делать если хватает привилегий пользователя, а если не хватает то не можно.

iЗдесь и далее значок **i**отмечает определение которое обязательно требуется заучить. Значок **i**помечает типичные ошибки о которых стоит помнить. Значок **i**, важная дополнительная информация.

Благодарности

Безусловно их будет здесь побольше. А пока хочу традиционно поблагодарить своих студентов за интересные вопросы во время тренингов. И вообще книга появилась на свет именно для них, как результат попытки сделать курс максимально простым и полезным.

Благодаря сотрудничеству с Андреем Сапрыкиным CiSchool.ru , у автора по прежнему имеются стимулы совершенствоваться и писать.

Копирайт

Используемые листинги, схемы и рисунки, дизайн и верстка авторские. Верстка выполнена в пакете LaTeX 2$_\varepsilon$- 6 марта 2020 г.Шрифты Linux Libertine, Linux Biolinum. При оформлении использовались свободно распространяемые программы Blender, Gimp и InkScape, а также собственные разработки автора.

Книга традиционно вендор-нейтральна, во всяком случае в той мере, в которой это получилось в рамках моего ограниченного кругозора. Все упоминания компаний и марок приведены исключительно в иллюстративных целях. Если кого то из владельцев оных названий и марок не устраивает их использование, достаточно одного письма и в следующем издании они будут заменены на соответствующее короткое русское слово.

Связаться с Михаилом Балкаровым можно по почте mailto:mbalkarov@gmail.com

Список литературы

[1] Михаил Балкаров. *Инженерные системы ЦОД*. 1-е изд. Третья книга автора. Киев: ИД “Аванпост-Прим”, 2014. ISBN: 978-617-502-060-9.

[2] Михаил Балкаров. *Охлаждение серверных и ЦОД. Основы*. 1-е изд. Первая книга автора. Киев: ИД “Аванпост-Прим”, 2011. ISBN: 978-617-502-024-1.

[3] Михаил Балкаров. *Программирование? Это очень просто*. 2019.

[4] Михаил Балкаров. *Электроснабжение серверных и ЦОД. Основы*. 1-е изд. Вторая книга автора. Киев: ИД “Аванпост-Прим”, 2012. ISBN: 978-617-502-035-7.

Оглавление

Глава 1

Введение в Java технологии

1.1. Особенности языка Java

Java или Ява, язык программирования общего назначения появившийся в 1995 году в компании Sun, ныне принадлежащий компании Oracle . Назван язык в честь острова, так что Ява, вполне корректное русское название. Его автором является James Gosling . Язык изначально разрабатывался для создания приложений для сети Интернет. По иронии судьбы, ту нишу под которую он изначально предлагался, позже полностью занял подобный по синтаксису, но совершенно другой по идеологи язык JavaScript.

Еще при проектировании был предпринят ряд мер, удачных и не особо, по реализации высокоэффективных операций с числами и массивами, а также оптимизации скорости исполнения программ. Формально язык является интерпретируемым, но интерпретируется не сам язык, а достаточно хорошо оптимизированные **байткоды**, последовательность байт содержащая команды специальной **виртуальной машины (JVM)** . Сама виртуаль-

ная машина, обычно программа с названием java.

Исходные тексты программ хранятся в файлах с расширением **.java**. После компиляции этих файлов при помощи программы с названием javac, создаются файлы с расширением **.class** содержащие байткоды.

$$file.java \xrightarrow{\text{javac}} file.class$$

Эти файлы могут собираться вместе образуя обычный zip архив, но с расширением **.jar**. В нем же могут хранится ресурсы и дополнительная информация о программе.

$$*.class \xrightarrow{\text{jar}} file.jar$$

После обработки исходных текстов программой javadoc создается документация в формате HTML описывающая программу.

$$file.java \xrightarrow{\text{javadoc}} *.html$$

Передавая команде java при запуске в качестве параметра имя файла .class или имя .jar как параметр опции **-jar** , мы указываем исполняемую программу.

$$file.jar \xrightarrow{java-jar} zzzzzz...$$

В общем случае, программа на Яве, не способна преобразовать собственный текстовый файл в байткоды и выполнить его, за исключением тривиального случая вызова компилятора javac. Этим язык отличается от большинства классических интерпретаторов. В состав стандартных библиотек включен интерпретатор JavaScript, но особой популярности не получил. Также есть возможность использовать утилиту JShell которая интерпретирует обрабатываемые строки как настоящий интерпретатор.

С другой стороны, подход с созданием виртуальной машины работающей с промежуточным языком байткодов стал классическим в силу его действительно высокой вычислительной эффективности. А сама виртуальная машина java является обще-

принятым способов реализации языков высокого уровня. Благо спецификации виртуальной машины открыты и модель лицензирования позволяет ее использовать практически без ограничений.

1.2. Описание Java технологий

Платформой (Platform) - называется комплект из виртуальной машины, вспомогательных программ и библиотек, для обеспечения создания и выполнения программ написанных на языке Ява. Лицензионные соглашения для разных платформ и их разных версий могут отличаться. Есть примеры аппаратной реализации платформы Ява в виде микропроцессоров непосредственно выполняющих байткоды.

Можно выделить три основные платформы:

- Standard Edition (Java SE) - распространенный язык общего назначения. Лицензия позволяет использовать его практически без ограничений, как коммерчески, так и для других целей. К примеру если требуется исходные тексты, то имеется OpenJDK базирующийся на том же самом коде с лицензией GPL. В этом курсе мы рассматриваем именно Java SE.
- Java ME Embedded - реализация платформы для использования в микропроцессорах и микроконтроллерах. Существует вариант с минимальными требованиями к ресурсам, для приложений выполняемых на смарт-картах.
- Enterprise Edition (Java EE) - Расширение SE дополнительными инструментами и библиотеками. В рамках этой платформы было реализовано большое количество технологий ориентированных на уровень большой организации. В состав входят библиотеки для клиент-серверных приложений, web-приложений, обмена информацией, эффективной работы с базами данных и многое другое.

Платформы Явы также традиционно делятся на:

Java Development Kit (JDK) - Полный набор библиотек и утилит позволяющий разрабатывать приложения.

Java Runtime Environment (Platform) - Минимальный набор

библиотек и утилит предназначенный на выполнение ранее разработанных приложений.

В качестве упражнения предлагается найти откуда скачивать на сайте oracle.com нужную вам версию JDK. Конечно JDK, мы ведь собираемся программировать. Возможно потребуется логин, но регистрация к счастью бесплатная и несложная. Запускаете файл инсталлятора, подтверждаете внесение изменений в систему и видите знакомую картинку, меняются только цифры версии.

Рис. 1.1: Диалог инсталляции JDK

Стоит отметить, что на один компьютер можно поставить много версий Явы и это потенциальный источник нетривиальных ошибок.

1.3. Использование IDE

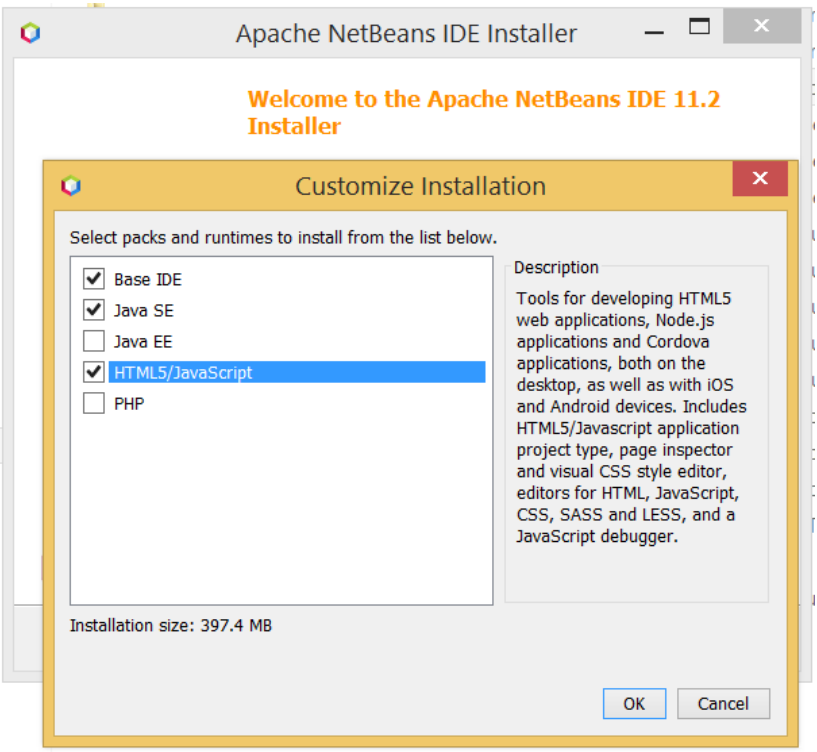


Рис. 1.2: Диалог выбора состава IDE

В современной практике программирования, практически всегда используется специализированная под язык или конкретную задачу среда разработки. Этим решается целый комплекс задач. От банальной подсветки синтаксиса, облегчающей понимание логики и упрощающей поиск ошибок, до автоматической генерации кода и вызова необходимых служебных утилит. Средой может быть даже простой текстовый редактор, с набором специализированных макроопределений.

В данном курсе предполагается использование среды разработки NetBeans в настоящее время принадлежащей Apache и скачиваемой соответственно с https://netbeans.apache.org/

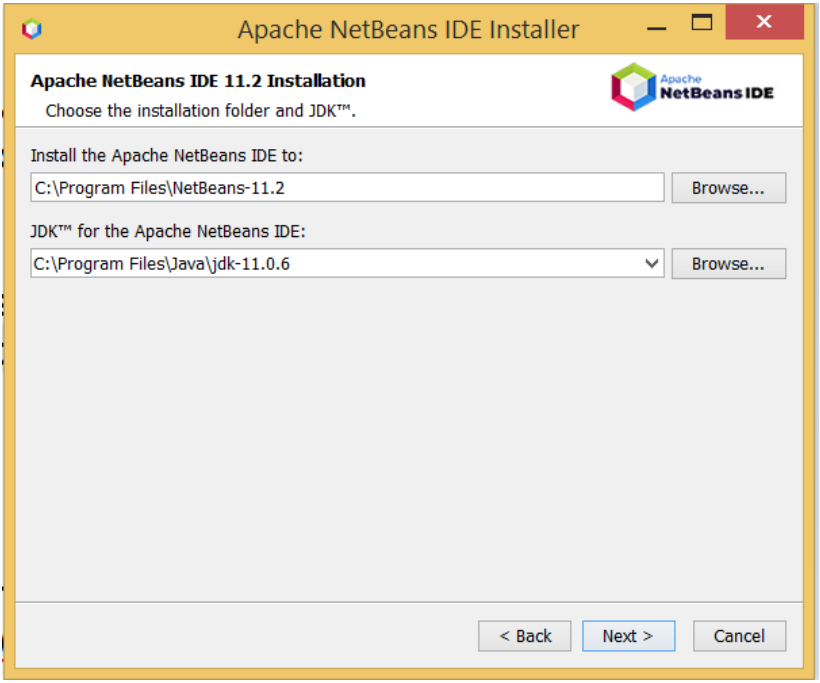


Рис. 1.3: Диалог выбора JDK для IDE

Начинать следует с установки той версии JDK с которой вы будете работать. Далее, скачав файл инсталляции NetBeans и запустив его, выбираете кнопкой **Customize** то, что нужно установить см. Рис 1.2.

Разумеется вы можете выбрать больше и доставить потом разнообразную поддержку.

Далее правильно выберите JDK для NetBeans см. Рис 1.3.

Выбор IDE в данном курсе на самом деле исключительно вопрос вкуса. Вы сможете с тем же успехом работать с любым другим вариантом из существующих. Это могут быть требования вашей организации, конкретного проекта или ваших привычек и опыта.

iОбязательно изучите IDE, она позволяет автоматизировать большую часть рутинных операций необходимых в любом проекте. В данном курсе мы будем использовать всего несколько процентов возможностей этого инструмента.

Попытаемся создать новый проект Java, среда запросит активацию возможностей, с чем мы естественно согласимся. Далее новые проекты под задания мы будем создавать указанным на Рис. 1.4 способом.

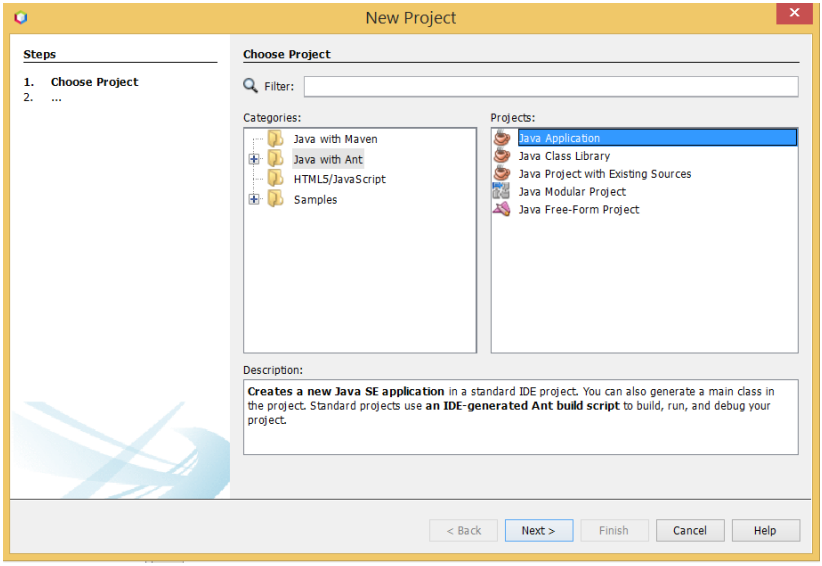


Рис. 1.4: Создание нового проекта в IDE

Глава 2

Введение в язык программирования Java

2.1. Языковые лексемы Java

Для интересующихся деталями на сайте Oracle есть формальное описание языка https://docs.oracle.com/en/java/javase/ Изучать его весело и познавательно, но занимает увы слишком много времени. Так что в этом курсе обойдемся достаточно нестрогими определениями и описаниями.

Ява изначально создавалась с полной поддержкой кодировки Unicode, а именно UTF-16. К сожалению в итоге реализация оказалась достаточно кривой. К примеру кодировка исходных файлов на языке Ява это UTF-8, файлы с UTF-16 при этом вызывают не очевидные ошибки. Кодировка строк внутри скомпилированного байткода UTF-16, но не простая, а с поддержкой 32 битового кодирования, так что их длинна может плавать. С файловым вводом выводом все еще хуже, но это разберем в соответствующей теме.

Лексический анализатор нарезает текст программы на:

- **WhiteSpace** - пробелы в которые входят переводы строки любого типа и табуляции. Игнорируются, могут быть добавлены в любом количестве;
- **Comment** - - классические комментарии аналогичные языкам C и C++. То есть парные /* */ и одинарные // до конца строки. Игнорируются, но могут обрабатываться javadoc;
- **Token** - все остальное.

В свою очередь Token разделяется на:

- **Identifier** - идентификаторы, буквы цифры подчеркивания и доллары. Начинается не с цифры, не может быть одиночным подчеркиванием, доллары не рекомендуются;
- **Keyword** - специальные идентификаторы зарезервированные в языке. Использовать не по назначению, в отличии от того же Fortran нельзя;
- **Literal** - самоопределенная константа. Строки, числа, символы, логические true и false, null;
- **Separator** - скобки и прочие знаки препинания. Содержимое этой строки “( ) { } [ ] ; , ... @ ::”;
- **Operator** - операция, достаточно похоже на все те же C и C++.

```
1  /*
2   * To change this license header, choose License Headers in Project Properties.
3   * To change this template file, choose Tools | Templates
4   * and open the template in the editor.
5   */
6  package javaapplication1;
7
8  /**
9   *
10  * @author daddym
11  */
12 public class JavaApplication1 {
13
14     /**
15      * @param args the command line arguments
16      */
17     public static void main(String[] args) {
18         // TODO code application logic here
19         System.out.print("Привед медвед!");
20     }
21
22 }
```

Листинг 2.1: Первая программа

Листинг 2.1 созданный IDE по умолчанию неплохо иллюстрирует синтаксис языка. Добавлена строка для печати приветствия. Почему она не сработает корректно при запуске из командной строки Windows?

2.2. Введение в систему типов языка Java

Начать наверное стоит с того, что Ява сильно типизированный язык, во всяком случае пытается быть таковым. Все идентификаторы имеют определенный и неизменный тип, от момента создания до освобождения выделенной им памяти. Использовать идентификатор до определения, запрещается. Синтаксис определения подобен C и C++.

Сама система типов языка Ява является компромиссом между его объектной ориентированностью и вычислительной эффективностью. Типы в первую очередь делятся на примитивные (PrimitiveType) и ссылочные (ReferenceType) не примитивные.

Примитивные типы полностью аналогичны типам в традиционных языках, их тип полностью обрабатываются и проверяется компилятором.

Тип	Инициализация	Размер	Boxing
boolean	false	1 бит	**Boolean**
byte	0	1 байт	**Byte**
short	0	2 байта	**Short**
int	0	4 байта	**Integer**
long	0L	8 байт	**Long**
char	\u0000	2 байта	**Character**
float	0.0F	4 байта	**Float**
double	0.0D	8 байт	**Double**

Таблица 2.1: Простые типы Ява

Упаковка (Boxing) - Преобразование примитивного типа в соответствующий ссылочный. То есть в полноценный экземпляр объекта класса.

Распаковка (Unboxing) - Преобразование ссылочного типа в соответствующий примитивный.

Инициализация по умолчанию запрещена для обычных переменных, но использует при создании массивов без явных инициализаторов.

Ссылочные типы, на нашем уровне рассмотрения, это объекты классов, и темы рассматриваемые далее, объекты реализующие интерфейсы и массивы. Отдельно выделяются ссылочные типы null и String. Кроме ссылки на объект соответствующего типа, любая переменная ссылочный типа может указывать на null, что означает отсутствие ссылки или на объект производного класса. Это заодно инициализация по умолчанию для ссылочных типов.

В отличии от примитивных типов, для ссылочных переменных можно узнать тип методом **.getClass()** от переменной и потом получить его имя методом **.getName()**. Этот метод подходит и для массивов. Или же класс получается вызовом:

```
Class.forName("myclass.MyClass");
```

Третий способ, работающий и с именами примитивных классов, поле **class**:

```
System.out.println(int.class );
```

Предварительно разумеется следует проверить на неравенство интересующей ссылки null:

```
 if(null !=  object){ ;}
```

или таким образом:

```
if( java.util.Objects.isNotNull(object)){ ;}
```

. По определению, попытки обращения относительно null, вызывают ошибку выполнения. Тем не менее

```
(null == null) == true
```

также по определению.

2.3. Работа с примитивными типами и константами

Константы в языке Ява задаются указанием модификатора **final** в определении переменной. Согласно соглашениям, имена констант полагается записывать всеми большими буквами. Если такое имя состоит из нескольких слов, они разделяются подчеркиванием.

Значение может присваиваться константам в месте определения или далее по потоку выполнения, но не более одного раза. Причем так, чтобы компилятор это смог понять. Если модификатор **final** присутствует в описании параметра метода, внутри метода этому параметру присваивание запрещено.

Целочисленные литералы задаются как числа в двоичной, восьмеричной, десятичной и шестнадцатеричной нотации. Для задания long литерала к нему в конце приписывается буква L. Для задания float используется F в конце литерала, для double можно указать D. Следует учитывать максимально допустимые значения из таблицы 2.2.

Тип	Минимум	Максимум
boolean	false	true
byte	-128	127
short	-32_768	32_767
int	-2_147_483_648	2_147_483_647
long	-9_223_372_036_854_775_808	9_223_372_036_854_775_807
char	\u0000 (0)	\uffff (65_535)
float	Float.NEGATIVE_INFINITY	Float.POSITIVE_INFINITY
double	Double.NEGATIVE_INFINITY	Double.POSITIVE_INFINITY

Таблица 2.2: Простые типы: минимумы и максимумы

2.4. Операции языка Java

Стоит пожалуй начать с типа boolean, для этого типа в Ява предоставлен классический набор операций

x	y	!x	!y	x & y	x | y	x ^ y
false	false	true	true	false	false	false
false	true	true	false	false	true	true
true	false	false	true	false	true	true
true	true	false	false	true	true	false

Таблица 2.3: Таблица истинности

Кроме того, имеется набор присваиваний, включая совмещенные с операцией над адресатом:

```
= &= |= ^=
```

в классическом смысле языка C и C++ и не менее классические:

```
&& ||
```

выполнение которых прерывается после достижения однозначного результата. То есть при первом true для цепочки || и первом false для цепочки &&. Классический пример:

```
if((null != obj) && obj.valid()){;}
```

Если объект не существует, вызова его метода и далее ошибки выполнения не произойдет.

2.5. Преобразование простых типов

Начнем опять же с boolean, это тип который требуется для всех условных операторов. В неявном виде он преобразуется только в Boolean и соответственно обратно. В явном виде он не преобразуется, что впрочем не составляет проблем, поскольку операции сравнения выдают именно его.

Примитивные численные типы преобразуются по умолчанию и без предупреждения по правилам сохранения точности:

- $double \Leftarrow float$
- $float \Leftarrow long$
- $long \Leftarrow int$
- $int \Leftarrow char$
- $int \Leftarrow short$
- $short \Leftarrow byte$

Также для численных типов операции упаковки и распаковки при необходимости выполняются без предупреждения.

Самый неоднозначный тип в смысле преобразования это String, хоть формально он и не относится к примитивам, но включен компилятором в семантику выражений. Если в операции сложения появляется объект такого типа, все остальные объекты преобразуются в него при помощи метода toString().

Для явного указания преобразования используется оператор каста. Это имя типа в круглых скобках перед выражением которое преобразовывается. Не обязательно компилятор разрешит конкретное преобразование. Не обязательно оно отработает в процессе выполнения. Опять же, не обязательно, что после преобразования не произойдет проблем с потерей знака и прочим.

Глава 3

Методы и операторы Java

3.1. Создание и вызов методов

Весь код в языке Ява это вызовы методов классов и экземпляров объектов классов. В итоге, для классического процедурного программирования приходится использовать методы классов даже если особого смысла в привязке к классу нет. Для поддержки функционального программирования, указатели на функции имитируются функциональными интерфейсами, но об этом поговорим позже в соответствующей теме.

Синтаксически метод, это определение функции внутри класса или интерфейса. Если в определении функции присутствует модификатор **static** это означает метод класса применимый к классу в целом и не имеющий прямого доступа к экземплярам объектов этого класса.

Методы вызываются через квалификатор точку (dot). Методы класса можно вызывать как от экземпляра класса, так и от самого имени класса. Внутри класса методы можно вызывать и

без уточнения через точку, как обычные функции процедурных языков программирования.

Порядок определения методов не важен. Если метод не возвращает значения он описывается как **void**. Эти принципы иллюстрирует листинг 3.1.

```
1  package javaapplication2;
2  /**
3   * Вызов методов
4   * @author daddym
5   */
6  public class JavaApplication2 {
7      /**
8       * @param args the command line arguments
9       */
10     public static void main(String[] args) {
11         // TODO code application logic here
12         System.out.println();
13         JavaApplication2 instance = new JavaApplication2();
14         instance.mi();
15         JavaApplication2.ms();
16         instance.ms();
17         ms();
18         instance.xmi();
19     }
20     static void ms(){ System.out.println("MS"); }
21     void xmi(){ mi(); }
22     void mi(){ System.out.println("MI"); }
23 }
```

Листинг 3.1: Определение и вызов методов

3.2. Перегрузка и методы с переменным числом аргументов

В классе допускается использование перегрузки (overload), то есть создание методов с теми же именами, но при этом разными и различимыми по типам параметрами или с разным числом параметров. Тип возвращаемого значения при этом может быть каким угодно, он не учитывается при выборе нужного метода.

Классическим примером является функция print, представляющая из себя целый куст разнообразных функций, объединенных концепцией читаемого человеком вывода, но с разными алгоритмами. Попробуем в IDE записать:

```
System.out.println(null);
```

Компилятор не может выбрать какой из двух имеющихся вариантов подходит больше:

```
System.out.println((char[])null);
```

или

```
System.out.println((String)null);
```

Проверьте результат обоих вариантов преобразования.

Ява поддерживает методы с переменным числом аргументов, что потенциально усугубляет возможную путаницу с вызовом нужного метода. Механизм переменного числа аргументов неявно реализован через массив, что демонстрирует очередная программа 3.2.

```
1  package javaapplication5;
2  /**
3   * Переменное число аргументов
4   * @author daddym
5   */
6  public class JavaApplication5 {
7
8      /**
9       * @param args the command line arguments
10      */
11     public static void main(String[] args) {
12         varArg();
13         varArg("один");
14         varArg("один", "два");
15     }
16     static void varArg(String... s){
17         System.out.println(s.getClass().getName());
18         for (String arg:s){
19             System.out.println(arg);
20         }
21     }
22 }
```

Листинг 3.2: Вызов метода с переменным числом аргументов

Синтаксически это три точки подряд после указания типа последнего аргумента. Количество аргументов можно выяснить используя поле length этого параметра.

3.3. Операторы ветвления

Язык Ява унаследовал управление выполнением от C и C++. Простое условное выполнение оператора:

```
if(cond) System.out.println(1);
```

или блока операторов:

```
if(cond) {System.out.println(1); System.out.println(2);}
```

Аналогично с частью else выполняемой если условие false:

```
if(cond) System.out.println(1); else System.out.println(2);
```

Специального оператора elseif в языке не существует, компилятор способен нормально оптимизировать цепочку else if произвольной длинны. Смысл такого оператора - выполнение ровно одного блока для которого выполнится условие соответствующего if или блока else если ни один из if не выполнился а else присутствует.

```
if(cond1) {System.out.println(1);}
else if(cond2) {System.out.println(2);}
else if(cond3) {System.out.println(3);}
else  {System.out.println(4);}
```

Кроме того, стоит напомнить, что ранее рассмотренные операторы && и || могут влиять на последовательность вычисления выражений возвращающих значение boolean. Помимо них есть еще тернарный оператор в выражениях:

```
cond ? true_expr : false_expr
```

который вычисляет условие, после чего вычисляет и возвращает одно из выражений. При истинном условии выражение после знака вопроса, при ложном условии выражение после двоеточия.

Оператор **switch** позволяет переходить по константам-меткам типа целых чисел, String и enum. Можно использовать метку **default**: означающую отсутствие совпадений. Переход по метке приводит к дальнейшему выполнению кода до конца блока switch. Прервать выполнение может встреченный оператор **break**, но это не является требованием.

```
1  package javaapplication6;
2  /**
3   *
4   * @author daddym
5   */
6  public class JavaApplication6 {
7      public enum Day {   Mon, Tue, Wed, Thu, Fri, Sat, Sun, }
8      /**
9       * @param args the command line arguments
10      */
11     public static void main(String[] args) {
12         // TODO code application logic here
13         a:for(Day d:Day.values()){
14             b:switch (d){
15                 case Thu:
16                     break a;
17                 default:
18                     System.out.println(d);
19                     while(true){
20                         break b;
21                     }
22             }
23         }
24     }
25 }
```

Листинг 3.3: Использование switch и enum

3.4. Операторы циклов

В Ява существует четыре оператора цикла, все они в бесконечной нотации приведены на листинге 3.2.

```
1  package javaapplication7;
2  /**
3   *Варианты циклов
4   * @author daddym
5   */
6  public class JavaApplication7 {
7      /**
8       * @param args the command line arguments
9       */
10     public static void main(String[] args) {
11         // TODO code application logic here
12         int ii[]={1,2,3,4,5};
13         a:while(true){
14             do{
15                 for(;;){
16                     for(var jj:ii)
17                         System.out.println(jj);
18                     break a;
19                 }
20             }while(true);
21         }
22     }
23 }
```

Листинг 3.4: Варианты циклов

Оператор **break** прерывает текущий цикл как и для switch, но может быть использован с меткой как в примере чтобы выйти на нужном уровне вложенности. Аналогично используется оператор **continue** пропускающий исполнение тела цикла до конца.

Глава 4

Создание и использование массивов Java

4.1. Одномерные массивы

Одномерные массивы в Яве реализованы классически, как непрерывная область памяти выделенная под данные одного типа. К примеру выражение вида

```
new int[10]
```

создаст в памяти область чисел типа int инициализированную нулями по умолчанию:

int	int	int	int	int	int	int	int	int	int

Также будет заполнено поле **length** для этого массива, которое очевидно будет равно 10.

4.2. Многомерные массивы

Многомерных массивов в Яве на самом деле не существует, по крайней мере в базовом языке. К примеру двухмерный массив, это массив массивов. Вы уже знаете, что массив это ссылочный тип, так что в основе массив ссылок на одномерные массивы. Доступ к их элементам идет по самому правому индексу.

Это и хранение для массива его размера, приводит к необычной возможности для многомерных массивов, они могут быть **jagged**, то есть ступенчатыми. То есть размеры строк многомерного массива могут быть разными.

```
1  package javaapplication8;
2  /**
3   *Массивы Ява
4   * @author daddym
5   */
6  public class JavaApplication8 {
7      /**
8       * @param args the command line arguments
9       */
10     public static void main(String[] args) {
11         // TODO code application logic here
12         int []aa[] = {{1,2,3},{4,5},};
13         for(int[] a:aa){
14             for(int  ii:a)
15               System.out.print(ii+" ");
16             System.out.println();
17         }
18     }
19 }
```

Листинг 4.1: Использование jagged массива

4.3. Работа с массивами и класс Arrays

Работа с массивами в Яве похожа на любой другой язык программирования. Вы указываете в квадратных скобках индекс нужного элемента и получаете содержимое или присваиваете его. Для многомерного массива, квадратные скобки выполняются слева направо, выдавая массивы уменьшающейся мерности.

Индексация элементов, как и во многих других С подобных языках идет с 0, то есть индекс, это не номер элемента, а его смещение от начала. Попытки обращения по отрицательным индексам и индексам за пределами размера отслеживаются при выполнении вызывая ошибку ArrayIndexOutOfBoundsException

Массивы можно смело передавать как параметры для методов и соответственно возвращать из методов. Следует понимать, что хоть параметры всегда передаются по значению, в методы передается значение именно ссылки, так что хоть с массивом в целом ничего не случится, его элементы вполне могут быть изменены внутри метода.

При декларации массива, квадратные скобки могут располагаться и до, и после имени массива. Кроме того массив может быть и вовсе анонимным, если используется как параметр или возвращаемое значение.

```
return new int[]{1,2,3,4,5};
```

Массивы поддерживают интерфейс **Cloneable** и следовательно имеют метод **clone**, создающий копию. Копия создается только для крайнего левого измерения. Так что для одномерного массива примитивных типов это будет **deep** копия, а в остальных случаях **shallow**.

В **System** есть утилита копирования, позволяющая в том числе безопасно копировать элементы в одном и том же массиве:

```
System.arraycopy(     src, srcPos, dest, destPos, length     )
```

В библиотечном классе **java.util.Arrays** имеется большой набор функций для обработки массивов, инициализация, сортировка, поиск, все то же копирование, сравнение.

Глава 5

Работа со строками в Java

5.1. Основные методы класса String

Со строками, объектами типа String, в Яве мы начинаем знакомство со строчного литерала, то есть последовательности символов в двойных кавычках. В этой последовательности могут попадаться эскейпы в смысле языка C, то есть обратная косая черта и спецсимвол или кодировка utf-16.

Все это перекодируется в utf-16 и хранится в специальном строчном пуле констант. Так что если в программе попадаются одинаковые строки в JVM не создаются копии. Литералы строк, как и сами строки, **immutable**, то есть не изменяемые по реализации, даже если это кажется не так.

Хоть с какой то точки зрения строка это массив символов char, на самом деле это совсем не так. Интерфейса типичного для массивов у строк нет и любимый цикл for с двоеточием к ним напрямую неприменим. Как впрочем и квадратные скобочки для доступа к отдельным элементам и поле length.

Впрочем мы не будем грустить и найдем замену среди методов объекта:

- **char** charAt(**int** index)- Символ на заданной позиции, index от 0 до length() -1;
- **int** length()- та самая длина строки;
- **char** [] toCharArray()- массив символов, создается копированием;
- **boolean** isEmpty()- проверка не пуста ли строка;
- **boolean** isBlank()- и не состоит ли из одних пробелов в смысле Java;
- **boolean** equals (Object anObject)- посимвольное сравнение. Обычное для строк не работает, проверяя идентичность;
- **boolean** equalsIgnoreCase (String anotherString)- то же но с игнорирование регистра буков;
- **int** compareTo (String anotherString)- Лексикографическое сравнение. Возвращает при разных символах на позиции k:
  this.charAt(k) - anotherString.charAt(k)
  или если строка кончилась, а разных символов не нашлось:
  this.length() - anotherString.length();
- **int** compareToIgnoreCase(String str)- то же но с игнорирование регистра буков;
- **byte** [] getBytes(String charsetName)- преобразование строки в массив байт заданной кодировки. Кодировка может быть: “UTF-8”, “UTF-16”, и.т.п.

Среди методов самого класса тоже есть интересное, это набор конструкторов строк позволяющих в частности создавать их из массивов байтов с заданной кодировкой и утилиты String.valueOf() возвращающие строчное представление для примитивных типов и объектов.

5.2. Сложение и преобразование строк

Разумеется имеется богатый набор операций по обработке строк. В первую очередь это сложение, все остальные типы складываясь со строкой вызывают свой метод toString() и конкатенацию получившегося строчного представления в одну большую строку. это эквивалентентно вызову метода для получившихся строк concat(String str).

Методы класса String для работы со строками:

- **static** String join(CharSequence delimiter, CharSequence... elements) - тоже складывает строки вместе добавляя между ними заданный разделитель;
- **static** String format(String format, Object... args) - форматирование строки, мощный инструмент описание которого имеется в классе java.util.Formatter.

Строки также можно преобразовывать их собственными методами, часть из которых приведена далее:

- String strip() - удаление пробелов в начале и конце строки;
- String strip() - удаление всего меньше или равного ‘\u0020’ в начале и конце строки;
- String substring(**int** beginIndex, **int** endIndex) - извлечение подстроки;
- String toLowerCase() - преобразование в нижний регистр;
- String toUpperCase() -преобразование в верхний регистр;
- String toString() - угадайте что это?

5.3. Классы динамических строк

Все ранее помянутые методы манипуляции строками неэффективны при массовом применении. Строки не изменяются, так что видимые изменения приводят к их постоянному пересозданию. Если алгоритм требует изменяемых строк, следует использовать соответствующие классы. Это изменяемсые строки StringBuilder и StringBuffer то же, дополнительно поддерживающий синхронизацию потоков выполнения. В String они преобразуются конструктором строк, из String своими конструкторами.

Кроме методов строк они поддерживают методы для модификации:

- StringBuilder append(*) - добавление всякого разного в конец билдера;
- **int** capacity() - текущий размер буфера;
- **void** ensureCapacity(**int** minimumCapacity) - настройка размера буфера;
- StringBuilder delete(**int** start, **int** end) - удаление символов;
- StringBuilder deleteCharAt(**int** index) - удаление символа;
- StringBuilder insert(**int** offset, *) - вставка всякого разного в позицию билдера;
- StringBuilder replace(**int** start, **int** end, String str) - замена;
- **void** setCharAt(**int** index, **char** ch) - замена чара;
- **void** setLength(**int** newLength) - установка длинны;
- **void** trimToSize() - сброс буфера;
- StringBuilder reverse() - извращенная замена порядка символов.

5.4. Ввод данных с клавиатуры и класс Scanner

По умолчанию переменная System.in направлена на консоль, что позволяет просто зачитывать символы из потока. Прочитанные строки далее можно преобразовать в простые типы данных при помощи метода **static** valueOf(String s) для соответствующего упаковочного класса. Но это не наш путь, воспользуемся классом **java.util.Scanner**. Он позволяет обрабатывать ввод по мере чтения. При этом создаваться может и для текстовых файлов.

```
1  package javaapplication9;
2  import java.util.Scanner;
3  /**
4   *
5   * @author daddym
6   */
7  public class JavaApplication9 {
8      /**
9       * @param args the command line arguments
10      */
11     public static void main(String[] args) {
12         Scanner scanner = new Scanner(System.in);
13         while (scanner.hasNext())
14             if (scanner.hasNextInt()){
15                 System.out.println("Integer " +scanner.nextInt() );
16             } else if (scanner.hasNextBoolean() ){
17                 System.out.println("Boolean " +scanner.nextBoolean() );
18                 break;
19             }else
20                 scanner.next();
21         scanner.close();
22     }
23 }
```

Листинг 5.1: Использование класса Scanner

5.5. Регулярные выражения

Сами про себе регулярные выражения, это отдельная тема освещенная и в теории обработки данных, и в реализации конкретных библиотек. Для нашего курса мы рассмотрим их применение в Яве, а не язык сопоставления. Начинается все с методов String:

- **boolean** matches(String regex) - проверка на совпадение с регулярным выражением regex;
- String replaceAll(String regex, String replacement) - замена всех совпавших;
- String replaceFirst(String regex, String replacement) - замена первого совпадения;
- String [] split(String regex) - разрез по совпадениям, с пустой строкой посимвольно. Не очень удачная обработка дублирующихся разделителей в начале и середине текста, образуются пустые строки.

Если требуется полное использование библиотеки regex, классы для нее находятся в java.util.regex пример чего приведен в листинге 5.2

```
1  package javaapplication14;
2  import java.util.regex.Matcher;
3  import java.util.regex.Pattern;
4  /**
5   * Серьезные реулярные выражения
6   * @author daddym
7   */
8  public class JavaApplication14 {
9      /**
10      * @param args the command line arguments
11      */
12     public static void main(String[] args) {
13         // TODO code application logic here
14         Pattern pattern = Pattern.compile("[0-9]{3}");
15         Matcher matcher = pattern.matcher("Это строка 123 и число 321");
16         while (matcher.find()) {
17             System.out.println("I found( "+matcher.group()+" )starting at "+
18             matcher.start()+" and ending at "+matcher.end());
19         }
20     }
21 }
```

Листинг 5.2: Использование java.util.regex

Глава 6

Разработка классов на Java

6.1. Обзор основных концепций ООП

Объектно ориентированное программирование, одна из методологий, для повышения производительности труда программистов. Ява, достаточно хорошо, за исключением своих примитивных типов, поддерживает этот подход в самом языке и поставляемых библиотеках.

В основе концепции ООП лежит идея что разнообразные **объекты(objects)** создаваемые в программе во время выполнения являются **экземплярами (instances)**, **классов (classes)**. Это дальнейшее развитие идеи типизации данных, позволяющему пользователю создавать свои классы-типы и объекты к ним принадлежащие. Класс выступает как шаблон для создания своих объектов, описывающий данные и функции для их обработки.

Для хорошей поддержки ООП требуются еще некоторые механизмы, которые в Яве присутствуют и выполнены достаточно изящно.

Это **Наследование (Inheritance)** позволяющее создавать между классами отношения вида **Родитель-Ребенок (Parent-child)** или другими словами **Суперкласс-Подкласс (Superclass-Subclass)**. При этом функционал и данные родителя переписываются и дополняются в ребенке позволяя повторно использовать уже разработанное ранее. В Яве для классов не используется множественное наследование, у каждого из них, только один родитель.

Любой объект является объектом класса своих родителей, до самого верха цепочки наследования. То есть объект Тузик, не только класса Собака, но и класса Млекопитающее, и Жывотное, и Живое, и Объект. Если у нас конечно выстроена такая иерархия наследования классов

Следующая концепция **Полиморфизм (Polymorphism)**, использование одного и того же имени функции, для вызова разных функций, выполняющихся по своему для разных объектов. К примеру toString для разных объектов устроена по своему обеспечивая читаемую выдачу.

Последняя важнейшая особенность ООП, **Инкапсуляция (Encapsulation)**, группировка данных и обрабатывающего их кода внутри класса-экземпляра и механизмы ограничения доступа из остальной программы, кроме явно разрешенного.

Можно еще помянуть **Абстракцию(Abstraction)**, описание функций и классов, без предоставления реализации на данном уровне. Это позволяет удобно выносить конкретную реализацию на более позднее время, в подклассы, при этом сохраняя концептуальную целостность объектов верхних уровней.

6.2. Объявление класса

В минимальном варианте объявления ключевое слово, имя класса и блок в фигурных скобках определяют класс:

```
class ClassName {}
```

Можно указать родителя нашего класса , словом очень хорошо описывающим суть наследования:

```
class ChildName  extends ParentName{}
```

Если родитель не указан то предполагается всехный родитель, класс Object.

iПо соглашению, имена классов это существительные начинающиеся с большой буквы, если слов несколько, они разделяются большими буквами. Системные классы описаны именно так.

6.3. Члены класса и создание объектов класса

Все переменные объявляемые непосредственно внутри блока класса, являются полями экземпляра класса. Они создаются при создании экземпляра класса и исчезают при его уничтожении.

Инициализация может происходить по умолчанию, при помощи присваивания в декларации выполняемых при создании полей:

```
class CName {long startTime = System.nanoTime();}
```

Либо блоками кода которые выполняются при создании экземпляра **(Instance Initializer block)**:

```
class CName {long startTime = -1; {startTime = System.nanoTime();}}
```

Эти блоки собираются вместе и выполняются когда уже отработал родительский конструктор, но еще не начал выполнятся собственный. Более подробно см. стр. 46.

В отличии от прочих методов, конструктор не должен иметь оператора **return** и явного описания возвращаемого типа. И то и другое подразумевается, поскольку изначально конструктор вызывается после выделения памяти, для полей его экземпляра объекта. Конструкторы вызываются в **new** и при создании деток. Более подробно см. стр. 46.

Для ссылок на собственный конструктор может быть удобен вызов **this()**. Это же слово **this** можно использовать для уточнения работы с переменными экземпляра. Для ссылок на родительские методы служит соответственно ссылка **super** и вызов **super()** для родительского конструктора.

6.4. Модификаторы доступа

Переменные методы и классы могут иметь модификаторы доступа, а могут не иметь, что тоже является указанием на их доступность. В таблице 6.1 систематизированы возможные варианты:

Модификатор	Класс	Пакет	Подклассы вне пакет	Прочие
private	+	-	-	-
	+	+	-	-
protected	+	+	+	-
public	+	+	+	+

Таблица 6.1: Таблица правил доступа

Есть достаточно логичные ограничения на использование модификаторов доступа:

- Хотя бы один конструктор в классе не должен быть **private**. Либо должна быть статическая функция, генератор экземпляров. Либо можно использовать статический блок, см. стр. 42. Иначе нет способа создать класс и подклассы. Компилятор это не ловит;
- Классы верхнего уровня не должны быть **private** поскольку это не имеет смысла, они не находятся внутри класса.
- В файле может быть ровно один класс верхнего уровня **public** и его имя должно совпадать с именем файла.
- Метод **static void main** обязан быть **public** иначе JVM его не стартует, а вот его класс не обязан.

6.5. Модификаторы final & static

Модикатор **final** уже поминался ранее для переменных означая, что больше одного присваивания таким переменным запрещено. Это семантика сохраняется и для методов, запрещено переопределять (override) методы описанные **final** в деточке.

Если же класс обзаводится таким модификатором, как большинство классов стандартной библиотеки, то от него нельзя наследовать.

Модикатор **static** применяемый для переменных объявляемых непосредственно внутри блока класса, являются полями самого класса. Они создаются при загрузке класса в память JVM. Эти поля общие и доступные для всех экземпляров класса. Так обычно задаются константы или глобальные настройки.

Модикатор **static** применяемый для методов класса, описывает методы класса. Для них недоступно **this** и поля экземпляра, поскольку в этом контексте определенного экземпляра не существует.

Модикатор **static** применяемый для блока является инициализатором однократно выполняющимся при загрузке класса в память JVM.

```
class CName {static long sTime = -1; static {sTime = System.nanoTime();}}
```

6.6. Использование пакетов, директив импорта и переменной среды CLASSPATH

Даже стандартные библиотеки языка Ява огромны. Помимо очевидных плюсов, у этого есть и минусы. Невозможно загружать все, следует указывать какие библиотеки, в терминологии Явы **package** использовать. IDE по умолчанию предполагает что вы разрабатываете пакет и создает нужную структуру директорий, а также прописывает внутри ваших файлов принадлежность к пакету:

```
package javaapplication12;
```

Если же мы используем пакеты, свои или системные, то должны их импортировать, либо указывая конкретный объект, либо звездочку для всех объектов (on-demand). Первый вариант правильней, второй проще.

```
import java.util.Scanner;
```

```
import java.util.*;
```

Далее можно использовать импортированные объекты как будто они определенны в файле. Если возникает конфликт одинаковых имен, то их можно полностью квалифицировать. К примеру работая с библиотекой сканирования изображений, к родному сканеру ввода Явы можно обращаться как к **java.util.Scanner**.

Есть и еще один вариант использования декларации импорта, статический:

```
import static java.lang.System.*;
```

Он позволяет использовать статические функции и переменные без имени их класса. Сделано это, в основном для имитации использования функций при обращении к библиотечным методам.

6.7. Модули Java, как единица развёртывания и безопасности

Со временем пакеты перестали удовлетворять требованиям к удобной организации крупных приложений. В дополнение к ним в языке появился механизм более высокого уровня - модули. Стандартные библиотеки точно также были переделаны с использованием этого механизма.

Модуль состоит из группы пакетов. Также модуль включает список всех пакетов, которые входят в модуль, и список всех модулей, от которых зависит данный модуль. Дополнительно он может включать другие файлы ресурсов. Модули обеспечивают лучшую гранулярность кода и проверку зависимостей на этапе запуска.

Технически модуль это папка с именем модуля к примеру demomod, в которой создается файл **module-info.java**. Внутри этого файла код описывающий модуль:

```
module demomod { }
```

Внутри этой папке создается структура директорий соответствующая имени пакета. А в самом **module-info.java** возможно добавятся требования:

```
module demomod { requires aaa.bbb;     }
```

или появятся экспортируемые наружу пакеты.

```
module demomod { exports my.demo.lib; }
```

Поддержка в текущей версии IDE несколько некорректна. Файлы создаются и компилируются, но не выполняются. Для выполнения следует запустить:

```
java -p ...\build\modules  -m demomod/newpackage.NewMainClass
```

или при настройке **jlink** в IDE

```
jbin\java -m demomod/newpackage.NewMainClass
```

Глава 7

Наследование и полиморфизм

7.1. Наследование как механизм повторного использования кода

При использовании наследования, уже существующие родительские поля и методы для класса и экземпляра появляются в новом классе. В крупных прикладных библиотеках вам зачастую предлагается просто переопределить несколько методов, чтобы воспользоваться предлагаемым сервисом, не пытаясь разобраться как оно устроено внутри.

Это же принцип можно применять буквально, с каждым новым наследованием добавляя очередной фрагмент функционала к вашему приложению.

7.2. Конструктор при наследовании

Код программы 7.1 иллюстрирует вызовы конструкторов и инициализацию. Код строк 12-15 выполняется после инициализации поля startTime, но в начале конструктора на строке 10.

```
1  package javaapplication11;
2  /**
3   *
4   * @author daddym
5   */
6  class P{
7     P(){System.out.println("P");}
8  }
9  class C extends P{
10    C(){ System.out.println("C " + startTime);};
11    long startTime = -1;
12    {
13       System.out.println("I" + startTime);
14       startTime = System.nanoTime();
15    }
16 }
17 public class JavaApplication11 {
18    /**
19     * @param args the command line arguments
20     */
21    public static void main(String[] args) {
22       // TODO code application logic here
23       C cc= new C();
24    }
25 }
```

Листинг 7.1: Вызовы конструкторов при наследовании

Родительский конструктор без параметров вызывается по умолчанию, перед отработкой собственного. Если хочется изменить этот функционал, вызовите первым оператором в своем конструкторе конструктор родителя **super()** с нужными параметрами. Этот вызов должен быть первым в коде детского конструктора.

```
ChildName() {super(123); moreInit(456);}
```

7.3. Преобразование типов и операция instanceof

В общем случае с любыми объектами мы можем работать используя хоть ссылки на Object корневой класс в иерархии классов Явы. Это так называемый **upcasting**, использование ссылок более высокого уровня иерархии классов. Напомню, что компьютерные деревья растут вниз, так что приближаясь к корню в котором находится Object, мы поднимаемся вверх. Все конечно здорово, но нам нужно иногда обращаться к объекту как к объекту определенного класса.

Как это делается безопасно? Объекты ссылочных типов хранят свой класс. Так что мы можем полезть в глубины и проверить его, а можем поступить проще, используя оператор **instanceof** и **downcasting**, то есть преобразование к конкретному детскому типу.

```
if(p instanceof A){ A a=(A)p; a.m(); }
```

Реального преобразование не производится конечно, просто вызывается корректный метод имеющийся в данном случае. При этом p не обязательно указывает на объект именно данного класса, возможно, это объект производного класса.

Чем еще **instanceof** очень удобен на практике, тем что спокойно обрабатывает **null**, возвращая **false** . В качестве переменной может выступать ссылка на объект реализующий интерфейс, но это тема следующей главы.

iЧем еще интересен **upcasting** и **downcasting**, одинаково называемые поля в экземплярах и классах которые прячутся при наследовании, становятся доступны при соответствующем выборе типа ссылки.

7.4. Виртуальные методы и позднее связывание

В языке Ява все методы являются виртуальными. То есть если я где нибудь в детке переопределяю метод, то все обращения к этому методу в методах родительских классов переопределяться тоже. Опять же, при работе не с примитивными типами их класс хранится вместе с ними и поэтому можно выбрать корректные методы, см. 7.2.

```
1  package javaapplication12;
2  /**
3   *Пример позднего связывания и виртуальных методов
4   * @author daddym
5   */
6  class P{
7      void print(){System.out.println("P");}
8      P(){print();}
9  }
10 class C extends P{
11     void print(){System.out.println("C");}
12 }
13 public class JavaApplication12 {
14     /**
15      * @param args the command line arguments
16      */
17     public static void main(String[] args) {
18         P p= new C();
19         p.print();
20     }
21 }
```

Листинг 7.2: Виртуальные методы и позднее связывание

7.5. Абстрактные классы и методы

В языке Ява есть специальный модификатор **abstract** для классов и методов, который не разрешает создавать объекты именно этого класса, только детки, в которых недостающее будет определено. При этом вы можете использовать переменные такого типа, как **upcasting** ссылки, на экземпляры реализующие заданный функционал.

Соответственно на практике абстрактный класс может быть использован в процессе пошаговой детализации задачи. Может быть использован для обозримого высокоуровневого описания реализованного в большой библиотеке. Может быть использован для принуждения пользователя библиотеки написать свои драйвера для реализации настраиваемых частей.

Глава 8

Интерфейсы и аннотации

8.1. Концепция интерфейсов

Древовидное наследование реализованное в Яве, не слишком удобно для некоторых применений. Поэтому в язык добавили концепцию интерфейсов реализующих указания на реализацию конкретного набора функций.

Это перекликается с идеей утиной (duck) типизации. То есть, если нечто плавает и крякает считаем это уткой. Сами по себе интерфейсы Ява как и классы могут наследоваться, причем у них допустимо множественное наследование. Кроме того класс может реализовывать несколько интерфейсов.

8.2. Объявление интерфейса

Интерфейс декларируется очень просто:

```
interface Printable{ void print(); }
```

Внутри блока ноль и более деклараций методов которые должен реализовать в будущем класс предоставляющий этот интерфейс.

В чем смысл интерфейса без функций, так называемого **marker** или **tagged**?

```
public interface Serializable{ }
```

Всего навсего, возможность пометить классы, для тех или иных целей.

Наследование интерфейсов объединяет их, с синтаксисом подобным тому что у классов :

```
interface Showable extends Printable{ void show(); }
```

Только после слова **extends** может идти список через запятую при множественном наследовании.

8.3. Реализация интерфейса

Если мы хотим реализовать интерфейс в нашем классе то должны в определении класса использовать слово **implements** и далее список реализуемых интерфейсов:

```
class C implements Serializable,Printable{ void print(){}; }
```

Если не все необходимые для реализации интерфейсов функции присутствуют, класс становится абстрактным.

8.4. Статические методы, методы по умолчанию в интерфейсах и приватные методы

Как и в классах в интерфейсах можно определить статические методы.

```
interface Some{ static void help(who){System.out.println("help me " + who);}; }
```

Они вызываются только от имени интерфейса.

```
Some.help("somebody" )
```

От экземпляра реализующего интерфейс или его класса эти методы недоступны.

Возможна реализация методов по умолчанию, с ключевым словом **default** . Если они не определены в классе реализующем интерфейс, то берется реализация по умолчанию:

```
interface Printable {default void print(){System.out.println("Undefined !!!");}}
```

Также внутри интерфейса можно определить константы:

```
interface Printable { void print(); int A4=4; int A3=3; }
```

Это удобно для группировки констант

Некоторые методы интерфейса могут быть описаны **private**, они полностью определяются и доступны только внутри.

8.5. Использование и создание аннотаций

В стандартном языке существуют следующие аннотации:

- @Override - указание на желание переписать метод;
- @Deprecated - указание на устаревший метод;
- @SuppressWarnings - подавление предупреждений;
- @SafeVarargs - безопасное использование переменного числа аргументов;
- @FunctionalInterface - указание на желание определить функциональный интерфейс, применимо понятно к интерфейсам.

При создании аннотаций тоже используются аннотации:

- @Retention(…) - область хранения данных анотации RetentionPolicy.SOU RetentionPolicy.CLASS, RetentionPolicy.RUNTIME;
- @Documented - обрабатывается javadoc;
- @Target(…) - выбор элементов для которых применима аннотация;
- @Inherited - наследуется ли подклассами;
- @Repeatable(…) - может применяться несколько раз сохраняя массив данных.

Ну а собственно сами аннотации создаются при помощи подобного кода 8.1 и используются как встроенные 8.2 .

```
1  package annotationsdemo;
2  import java.lang.annotation.*;
3  /**
4   *Определение аннотаций
5   * @author daddym
6   */
7  @Retention(RetentionPolicy.RUNTIME)
8  @Target(ElementType.METHOD)
9  @interface Tested{
10    boolean value() default false;
11 }
12
13 @Retention(RetentionPolicy.SOURCE)
14 @Documented
15 @interface Todo{}
```

Листинг 8.1: Определение анотаций

```
1  package annotationsdemo;
2  /**
3   *Применение аннотаций
4   * @author daddym
5   */
6  public class AnnotationUse {
7     @Tested public void m1(){System.out.println("Method ");}
8     @Tested(value=true)public void m2(){}
9     @Tested(true)public static void m3(){}
10
11    /**
12     *aaaa
13     */
14    @Todo public void m4(){}
15 }
```

Листинг 8.2: Использование анотаций

Глава 9

Пакет java.lang

9.1. Класс Object и переопределение его методов

Класс Object находится на самом верху дерева классов. Он определен в пакете **java.lang** который включается в любой файл кода по умолчанию. Все остальные классы являются его подклассами, детьми. Попытка получить его суперкласс вернет null.

9.2. Метаданные и рефлексия

Метаданные (Metadata), это дополнительные данные привязанные к коду и напрямую не влияющие на выполнение программы. К примеру аннотации могут выполнять такую роль, поскольку могут храниться вместе с объектами программы.

Рефлексия (Reflection), в терминологии Явы, это изменение изучение и изменение кода программы в процессе выполнения.

Рефлексия в первую очередь базируется на **java.lang.Class** который фактически является **Метаклассом (Metaclass)** языка. То есть все классы являются экземплярами этого класса.

Получить объект типа Class для заданного класса можно следующим образом:

- java.lang.Class.forName("..."); статический метод загружающий класс по полному имени;
- метод Object.getClass() позволяющий получить класс для любого не примитивного типа;
- поле .class позволяющее определить класс даже для примитивов.

Получив объект типа класс с ним уже можно делать что угодно. К примеру получить аннотации, созданные ранее, см. листинг 9.1. Или создать новый объект этого класса вызовом newInstance(), как альтернатива оператору **new**. А при необходимости можно даже вызвать метод объявленный как **private**.

```
1  package annotationsdemo;
2  import java.lang.reflect.Method;
3  /**
4   * Генерация отчета по аннотациям
5   * @author daddym
6   */
7  public class AnnotationsDemo {
8      /**
9       * @param args the command line arguments
10      */
11     public static void main(String[] args) throws ClassNotFoundException {
12         // TODO code application logic here
13         Class cl = Class.forName("annotationsdemo.AnnotationUse");
14         Method[] methods = cl.getDeclaredMethods();
15         for(Method md: methods)
16             if(md.isAnnotationPresent(Tested.class))
17                 System.out.println("Method " + md.getName() + " test status "
18                         + md.getAnnotation(Tested.class).value());
19             else
20                 System.out.println("Method " + md.getName() + " has no test annotation");
21     }
22 }
```

Листинг 9.1: Зачитывание примененных аннотаций

9.3. Классы System и Math

Класс **java.lang.System** реализует базовые системные функции.

Класс **java.lang.Math** реализует не менее базовую математику.

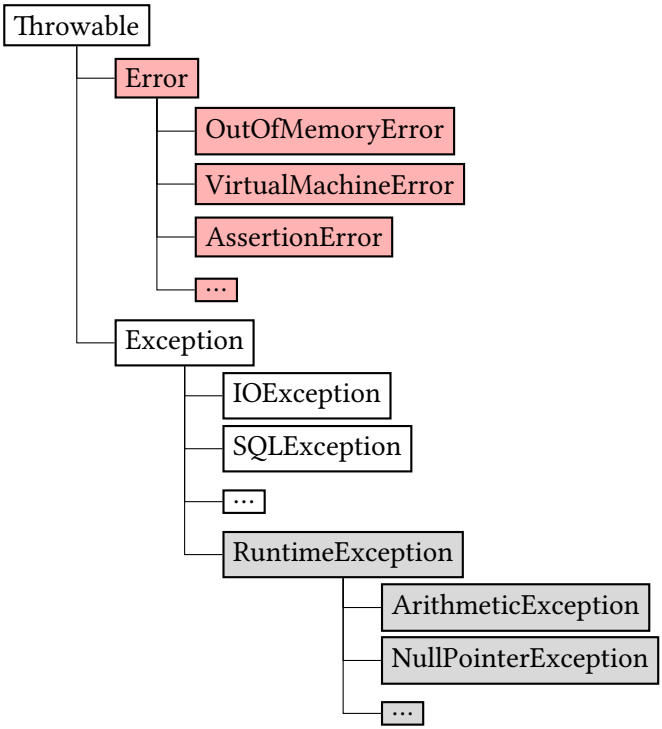
Глава 10

Обработка ошибок в Java

10.1. Концепция исключений в Java

Исключения (Exeptions), - прерывание нормального выполнения программы по той или иной причине. В качестве представления в языке, для возможной обработки, используется объект класса из соответствующей иерархии. Как раз в рамках наследования они делятся на:

- ошибки, безусловно прекращающие дальнейшую работу, объекты класса **Error** и его подклассов;
- **Непроверяемые (Unchecked)**, - не проверяемые компилятором на наличие обработчиков, объекты класса **RuntimeException** и его подклассов;
- **Проверяемые (Checked)**, остальные (я знаю что перевод дурацкий).



Дерево классов на рисунке изображено не полностью, но и в таком виде понятен принцип группировки.

10.2. Использование операторов try, catch и finally

```
1  package javaapplication17;
2  /**
3   * Обработка исключений
4   *
5   * @author daddym
6   */
7  public class JavaApplication17 {
8      /**
9       * @param args the command line arguments
10      */
11     public static void main(String[] args) {
12         try {
13             try {
14                 double ff = -10. / 0;
15                 System.out.println(ff + " " + (int) ff);
16                 int ii = 10 / 0;
17             } catch (ArithmeticException e) {
18                 System.out.println(e);
19                 throw new Throwable();
20             } finally {
21                 System.out.println("finally");
22             }
23         } catch (Throwable e) {
24             System.out.println(e);
25         }
26     }
27 }
```

Листинг 10.1: Синтаксис обработки исключений

Листинг 10.1 демонстрирует синтаксис языка для обработки исключений. Блок в фигурных скобках после **try** это код для которого будут перехватываться исключения. В конце этого блока опционально может быть один или несколько блоков **catch** и опционально один блок **finaly**.

В круглых скобках после **catch** описано конкретное исключение, которое ловится эти блоком. Все подклассы этого исключения также ловятся этим блоком. Поэтому если блоков **catch** несколько, то они должны идти в порядке генерализации.

После перехвата исключения блоком **catch** оно удаляется. В один момент времени может быть активно только одно исключение. Если есть необходимость передать перехваченное исключение выше по уровню вложенных обработчиков, или создать нужное используется оператор **throw** с параметром объектом класса исключение.

Если присутствует блок блок **finaly**, код в нем выполняется всегда, не зависимо от способа покидания блока **try**, за исключение падения JVM или вызова **exit()**.

10.3. Проверяемые и непроверяемые исключения

По факту различие заключается лишь в том, что компилятор найдя место возможного возбуждения checked и исключения требует либо использовать блок **try**, либо поместить в объявлении методе указание **throws** после конца списка параметров и закрывающей скобочки.

10.4. Создание своих классов исключений

Собственные классы исключений создаются как подклассы существующей иерархии исключений. Их обработка ничем не отличается от системных классов.

10.5. Оператор try для освобождения ресурсов

В операторе **try** могут быть опциональные круглые скобки в которые помещают имена переменных или определения переменных с инициализацией.

```
try(){;}
```

Это гарантирует, что по завершению блока try любым способом, для этих объекты вызовется метод **close()**.

Объекты обязаны реализовывать интерфейс java.lang.AutoCloseable, что включает интерфейс java.io.Closeable.

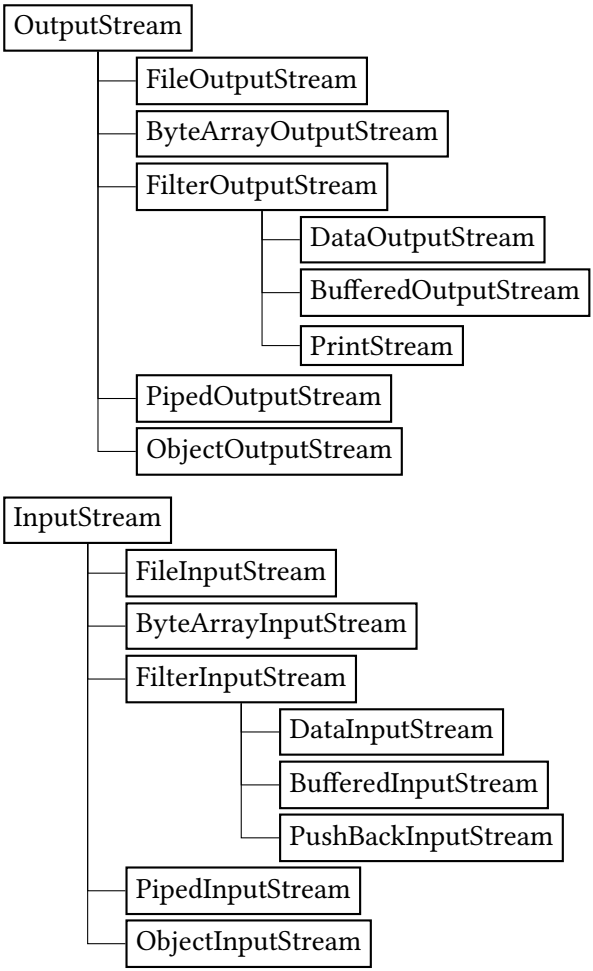
Глава 11

Потоки данных в Java

11.1. Обзор классов потоков

Потоки (Streams), объекты реализующие концепция последовательной обработки данных. К примеру печать на экран или ввод с клавиатуры может описываться в таких терминах. Выделяются потоки для ввода, принимающие данные в программу и для вывода, выдающие данные во внешний мир. От конкретных реализаций потоки обычно отвязаны, как правило один и тот же механизм может работать и по сети, и с файлами на диске, и даже с массивом в собственной программе.

11.2. Работа с байтовыми потоками



```
1  package ioserver;
2  import java.io.DataInputStream;
3  import java.net.ServerSocket;
4  import java.net.Socket;
5  /**
6   * Сервер
7   *
8   * @author daddym
9   */
10 public class IOServer {
11     /**
12      * @param args the command line arguments
13      */
14     public static void main(String[] args) {
15         try(ServerSocket ss = new ServerSocket(6666);) {
16             String str;
17             do{
18                 Socket cs = ss.accept();//establishes connection
19                 DataInputStream dis = new DataInputStream(cs.getInputStream());
20                 str = (String) dis.readUTF();
21                 System.out.println("get= " + str);
22                 cs.close();
23             }while(!"quit".equalsIgnoreCase(str));
24         } catch (Exception ex) {
25             System.out.println(ex);
26         }
27     }
28 }
```

Листинг 11.1: Чтение байтов на сервере

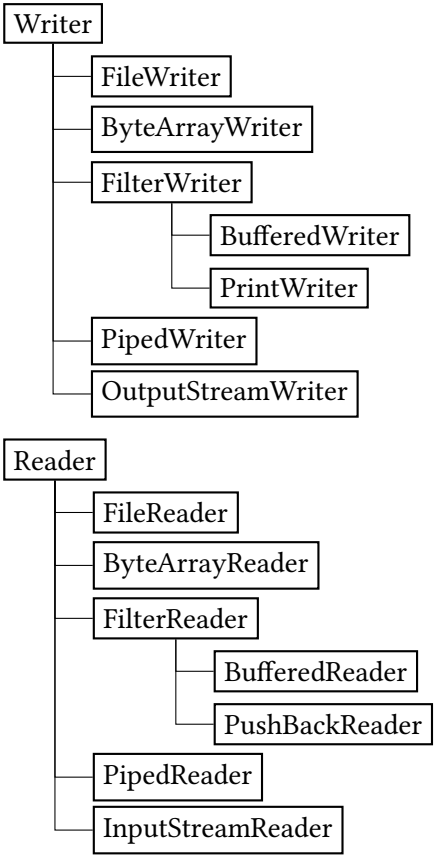
Программа 11.2 читает данные по сети, а ?? пишет их. Наглядно иллюстрируется тезис, что потокам в общем все равно, как выглядит обмен на более низком уровне. Заодно продемонстрировано преимущество кодировки UTF-8.

```
1  package ioclienr;
2  import java.io.DataOutputStream;
3  import java.io.IOException;
4  import java.net.Socket;
5  import java.util.Scanner;
6  /**
7   * Клиент
8   * @author daddym
9   */
10 public class IOClient {
11
12     /**
13      * @param args the command line arguments
14      */
15     public static void main(String[] args) {
16         try(Socket s=new Socket("localhost",6666);
17             DataOutputStream dout=new DataOutputStream(
18                     s.getOutputStream());
19             Scanner scanner = new Scanner(System.in);
20         ){
21             dout.writeUTF(scanner.next() );
22             dout.flush();
23         }  catch (IOException ex) {
24             System.out.println(ex);
25         }
26     }
27 }
```

Листинг 11.2: Запись байтов на клиенте

11.3. Работа с потоками символов

При работе с потоками символов есть возможность в самом потоке производить перекодировку к нужному формату символьных данных. Иерархия классов подобна таковой для байтов.



Самые полезные нам классы, это InputStreamReader и OutputStreamWriter. Они при создании позволяют указывать кодировку байтового потока на котором создаются.

11.4. Использование класса java.io.File

Это базовый набор операций с файлами как демонстрирует программа 11.3 :

```
1  package jdir;
2  import java.io.File;
3  import java.io.IOException;
4  import static java.lang.System.*;
5  /**
6   * Обработка списка файлов
7   *
8   * @author daddym
9   */
10 public class Jdir {
11     /**
12      * @param args the command line arguments
13      */
14     public static void main(String[] args) {
15         try {
16             File dir = new File((args.length > 0) ? args[0] : ".");
17             out.println("For " + dir.getCanonicalFile().getPath());
18             for (File file : dir.listFiles())
19                 out.println( file.getCanonicalFile().getName()
20                         + " Can Read: " + file.canRead()
21                         + " Can Write: " + file.canWrite()
22                         + "Is Hidden: " + file.isHidden() + " Length: "
23                         + file.length() + " bytes");
24         } catch (IOException ex) {
25             out.println(ex);
26         }
27     }
28 }
```

Листинг 11.3: Чтение списка файлов в каталоге

11.5. Сжатие файлов

В состав стандартной Явы входит большой набор библиотек для работы с разными вариантами компрессии. Но нам пожалуй интересней всего будет посмотреть как прочитать ресурсы из jar файла приложения. Разместим в IDE файл с парами “ключ=значение” там же, где и соответствующий файл java. При компиляции и создании jar, все скопируется, куда нужно. Это настройки по умолчанию.

```
1  package javaapplication22;
2  import java.io.BufferedReader;
3  import java.io.InputStreamReader;
4  import java.util.Properties;
5  /**
6   *
7   * @author daddym
8   */
9  public class JavaApplication22 {
10     /**
11      * @param args the command line arguments
12      */
13     public static void main(String[] args) {
14         Properties props = new Properties();
15         try (var input = new BufferedReader(new InputStreamReader(
16                 JavaApplication22.class.getResourceAsStream("props.ini")));){
17             props.load(input);
18         } catch (Exception ex) {System.out.println( ex );}
19         System.out.println(
20             props.getProperty("english") +" "+ props.getProperty("russian"));
21     }
22 }
```

Листинг 11.4: Чтение текстового файла из jar

11.6. Сериализация объектов в Java

Сериализация и десериализация в Яве, это выгрузка и загрузка объектов между памятью JVM и внешним хранилищем.

Для простейшего примера нужно просто определить у класса интерфейс маркер Serializable что приводит к простой загрузке и выгрузке из памяти. Пример такой программы приведен на листинге 11.5.

Можно определить какие то поля нашего класса со словом **transient**, что обеспечит отсутствие восстановления при десериализации. У поле остается значение по умолчанию.

Есть более продвинутый вариант такого интерфейса, Externalizable. Он позволяет самостоятельно вызывать нужную инициализацию при загрузке и выгрузке.

Для этих операций используются стандартные потоки типа ObjectOutputStream и ObjectInputStream.

Потенциально это небезопасная операция, поэтому при загрузке данных таким способом следует предпринимать необходимые меры предосторожности. По факту код полученный таким способом может получить управление и захватить власть.

```
1  package javaapplication23;
2  import java.io.*;
3  class MyC implements Serializable{
4      MyC(){System.out.println("Constructor here " + this.hashCode());}
5      String name;}
6  /**
7   * Сохранение восстановление
8   * @author daddym
9   */
10 public class JavaApplication23 {
11    /**
12     * @param args the command line arguments
13     */
14    public static void main(String[] args) {
15       MyC m1 = new MyC();
16       m1.name="m1";
17       MyC m2 = new MyC();
18       m2.name="m2";
19       try (var oos = new ObjectOutputStream(new FileOutputStream(
20             new File("person.ser")));) {
21          oos.writeObject(m1); oos.writeObject(m2);
22       } catch(Exception ex){System.out.println(ex);}
23       m1 = null; m2 = null;
24       //Read again
25        try (var ois = new ObjectInputStream(new FileInputStream(
26             new File("person.ser")));) {
27          m2 = (MyC)ois.readObject();
28          m1 = (MyC)ois.readObject();
29          System.out.println(m1.name + m2.name);
30       } catch(Exception ex){System.out.println(ex);}
31    }
32
33 }
```

Листинг 11.5: Сохранение и восстановление объектов

Глава 12

Работа с файловой системой в NIO 2

12.1. Использование интерфейса Path

Интерфейс java.nio.Path это более продвинутый вариант объекта java.io.File. В обоих случаях хранимый в объекте такого типа путь к файлу не обязан быть валидным. Между собой они взаимодействуют прекрасно:

```
Path path = file.toPath();

File file = path.toFile();
```

По сути Path отличается более корректным названием класса и большим числом методов.

Объекты Path можно сравнивать друг с другом, разбирать на компоненты и преобразовывать из абсолютных в относительные и наоборот.

Получат Path весьма нетривиально.

```
Path path = FileSystems.getDefault().getPath("");
```

12.2. Работа с атрибутами файлов

Эта библиотека обеспечивает полный доступ к атрибутом файлов в зависимости от активных файловых систем, листинг 12.1 показывает это.

Узнав какие атрибуты поддерживаются, можно запросить или изменить их. К примеру базовые атрибуты вроде типа, размера и времен, зачитываются так:

```
BasicFileAttributes bfa = Files.readAttributes(path, BasicFileAttributes.class);
```

```
1  package javaapplication24;
2  import java.nio.file.*;
3  import java.nio.file.attribute.*;
4  /**
5   *
6   * @author daddym
7   */
8  public class JavaApplication24 {
9      /**
10      * @param args the command line arguments
11      */
12     public static void main(String[] args) {
13         Path path = Paths.get("C:");
14         try {
15             FileStore fs = Files.getFileStore(path);
16             printDetails(fs, AclFileAttributeView.class);
17             printDetails(fs, BasicFileAttributeView.class);
18             printDetails(fs, DosFileAttributeView.class);
19             printDetails(fs, FileOwnerAttributeView.class);
20             printDetails(fs, PosixFileAttributeView.class);
21             printDetails(fs, UserDefinedFileAttributeView.class);
22         } catch (Exception ex) {
23             ex.printStackTrace();
24         }
25     }
26     public static void printDetails(FileStore fs,
27             Class<? extends FileAttributeView> attribClass) {
28         boolean supported = fs.supportsFileAttributeView(attribClass);
29         System.out.format("%s is  supported: %s%n", attribClass.getSimpleName(),
30                 supported);
31     }
32 }
```

Листинг 12.1: Проверка поддерживаемых атрибутов

12.3. Основные возможности класса Files

Класс **Files** это просто библиотека функций для работы с файлами. Помимо всего прочего она предоставляет удобный инструмент для открытия потоков.

12.4. Обход дерева каталогов

```
1  package javaapplication25;
2  import java.io.IOException;
3  import java.nio.file.*;
4  import java.nio.file.attribute.*;
5  /**
6   * обход файловой системы
7   * @author daddym
8   */
9  public class JavaApplication25 extends SimpleFileVisitor<Path> {
10     @Override
11     public FileVisitResult visitFile(Path file, BasicFileAttributes attrs)
12             throws IOException {
13         System.out.println(file);
14         return FileVisitResult.CONTINUE; }
15     @Override
16     public FileVisitResult preVisitDirectory(Path dir, BasicFileAttributes attrs)
17             throws IOException {
18         System.out.println(dir);
19         return FileVisitResult.CONTINUE; }
20     /**
21      * @param args the command line arguments
22      * @throws java.io.IOException
23      */
24     public static void main(String[] args) throws IOException {
25         Files.walkFileTree(Paths.get("C:"), new JavaApplication25());
26     }
27 }
```

Листинг 12.2: Проверка поддерживаемых атрибутов

Обход дерева каталогов в этой библиотеке делается через создание класса, листинг 12.2.

12.5. Мониторинг изменений ФС

```
1  package javaapplication26;
2  import java.nio.file.*;
3  import static java.nio.file.StandardWatchEventKinds.*;
4  import java.nio.file.WatchEvent.Kind;
5  /**
6   * Мониторнг файловой системы
7   * @author daddym
8   */
9  public class JavaApplication26 {
10   public static void main(String[] args) {
11     try ( WatchService ws = FileSystems.getDefault().newWatchService()) {
12       Path dirToWatch = Paths.get("C:\\alg");
13       dirToWatch.register(ws, ENTRY_CREATE, ENTRY_MODIFY, ENTRY_DELETE);
14       while (true) {
15         WatchKey key = ws.take();
16         for (WatchEvent<?> event : key.pollEvents()) {
17           Kind<?> eventKind = event.kind();
18           if (eventKind == OVERFLOW) {
19             System.out.println("Избыток событий");
20             continue;
21           }
22           WatchEvent<Path> currEvent = (WatchEvent<Path>) event;
23           Path dirEntry = currEvent.context();
24           System.out.println(eventKind + "  случилось с  " + dirEntry);
25         }
26         boolean isKeyValid = key.reset();
27         if (!isKeyValid) {
28           System.out.println("Все с " + dirToWatch);
29           break;
30         }
31       }
32     } catch (Exception e) { System.out.println(e);}
33   }
34 }
```

Листинг 12.3: Мониторинг изменений в файловой системе

Глава 13

Пакет java.util

13.1. Форматирование данных

В Яве основа форматного вывода данных это **java.util.Formatter**. В описании этого класса рассказано о форматных символах, подобных классическому синтаксису C для printf. Впрочем в Яве есть **printf** для **PrintStream** и такой же **format**. Для типа String используется именно **format**.

13.2. Работа с датой и временем

В современных программах следует использовать только **java.time** . В этом пакете определяются следующие классы:

- **Instant** - таймштамп, подход удобный для сравнения очередности событий;
- **LocalDate** - просто текущая дата;
- **LocalTime** - просто текущее время;
- **LocalDateTime** -дата и время;
- **ZonedDateTime** - дата и время с таймзоной.

Для них имеется достаточно богатый набор операций и дополнительные классы типа периодов времени.

13.3. Класс Locale и глобализация кода

К счастью локаль ru и ru_RU в современные версии Явы входит. Что можно активно использовать при выборе локалезависимых параметров.

```
Locale.setDefault(new Locale("ru_RU"));
```

Русский в итоге неплохо поддерживается:

```
String m = Month.FEBRUARY.getDisplayName(TextStyle.FULL , new Locale("ru"));
```

даже в разных падежах:

```
String s = Month.FEBRUARY.getDisplayName(TextStyle.FULL_STANDALONE , new Locale("ru"));
```

13.4. Локализация и класс ResourceBundle

В частности локаль полезна для выбора файлов properties, имена которых так и задаются с суффиксом из локали через подчеркивание. В данном примере 13.1 это mybundle_en_US.properties и mybundle_ru_RU.properties. Они содержат пары ключ значение через знак равно. Располагаются эти файлы в корне соответствующих директорий. То есть как бы в пакете по умолчанию.

```
1  package javaapplication27;
2  import java.util.Locale;
3  import java.util.ResourceBundle;
4  /**
5   * Локализация
6   *
7   * @author daddym
8   */
9  public class JavaApplication27 {
10     /**
11      * @param args the command line arguments
12      */
13     public static void main(String[] arg) {
14         ResourceBundle bundle = ResourceBundle.getBundle("mybundle", Locale.US);
15         System.out.println("Message in " + Locale.US + ":" + bundle.getString("hi"));
16         Locale.setDefault(new Locale("ru", "RU"));
17         bundle = ResourceBundle.getBundle("mybundle");
18         System.out.println("Message in " + Locale.getDefault() + ":" + bundle.getString("hi"));
19     }
20 }
```

Листинг 13.1: Локализация сообщений

13.5. Генерация псевдослучайных чисел

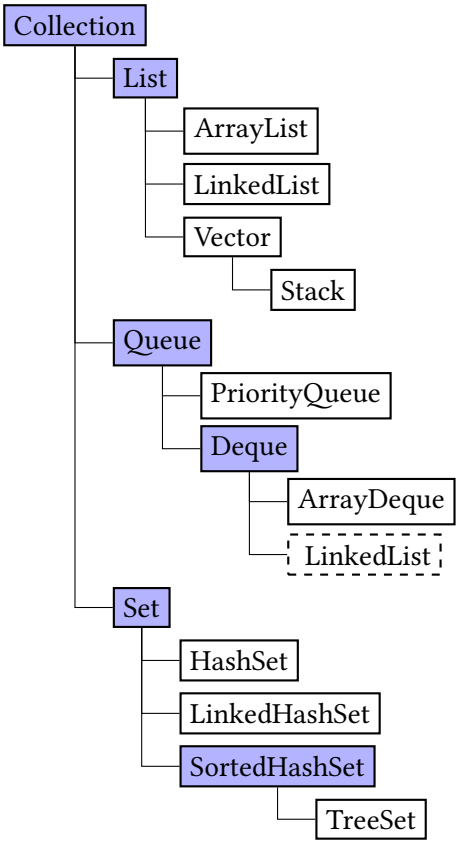
java.util.Random - класс позволяющий организовать генератор случайных, точнее конечно псевдослучайных чисел.

Глава 14

Коллекции в Java

14.1. Иерархия классов коллекций

Набор коллекций давно входит в стандартную библиотеку и достаточно удобно и эффективно реализован. Следует иметь в виду что эти типы данных небезопасные, с точки зрения использования нескольких нитей выполнения. Цветом на схеме помечены интерфейсы.


Collection
List
ArrayList
LinkedList
Vector
Stack
Queue
PriorityQueue
Deque
ArrayDeque
LinkedList
Set
HashSet
LinkedHashSet
SortedHashSet
TreeSet

14.2. Параметризация типов данных (Generics)

Кроме подстановки значения фактических параметров вместо формальных, при вызове методов, на чем и базируется традиционный вызов функций, удобно иметь в языке подстановку типов параметров, переменных и прочего, где в языке может иметься тип. Она особо актуально для языков со строгой типизацией. Иначе в классах вроде коллекций пришлось бы использовать ссылки на Object и лишаться преимуществ этой самой сильной типизации.

Называется это Синтаксически это использование шаблона в угловых скобках для типа <T> для задания конкретного типа T для объектов с параметризацией. По факту аналог макроподстановки заменяющий имя в угловых скобках, на задаваемый при вызове тип. Остается строгая типизация. Не нужен кастинг типов в явном виде. Проверка типов выполняется еще на этапе компиляции. Принято использовать следующие имена параметров:

- T - Type;
- E - Element;
- K - Key;
- N - Number;
- V - Value;
- ? super V - родительский класс V;
- ? extends V - класс производный от V.

Для параметризации класса угловые скобки указываются после его имени, как в примере 14.1 :

14.3. Работа с параметризованными данными

```
1  package javaapplication30;
2  class  MyVar<T>{
3     private T field;
4     T get(){ return field;}
5     T set(T field){ this.field = field; return field;}
6  }
7  /**
8   *
9   * @author daddym
10  */
11 public class JavaApplication30 {
12    /**
13     * @param args the command line arguments
14     */
15    public static void main(String[] args) {
16       //MyVar<> ii = new MyVar<>();
17       MyVar jj = new MyVar<Integer>();
18       jj.set(10);
19       System.out.println(" " + jj.get() );
20    }
21 }
```

Листинг 14.1: Пример параметризации

14.4. Обзор возможностей коллекций

```
1  package javaapplication31;
2  import java.util.*;
3
4  /**
5   *
6   * @author daddym
7   */
8  public class JavaApplication31 {
9
10     /**
11      * @param args the command line arguments
12      */
13     public static void main(String[] args) {
14         ArrayList<String> list = new ArrayList<String>();//Creating arraylist
15         list.add("Раз");
16         list.add("Два");
17         list.add("Три");
18 //Traversing list through Iterator
19         for(Iterator itr = list.iterator();itr.hasNext();)
20             System.out.println(itr.next());
21         for(var name:list)
22             System.out.println(name);
23         list.forEach((name) ->
24             {System.out.println(name); });
25     }
26 }
```

Листинг 14.2: Варианты циклов по коллекциям

Глава 15

Вложенные классы в Java

15.1. Внутренние классы

Внутренние (Internal) классы , это не статические вложенные. Они интересны, тем что имеют доступ ко всем элементам объемлющего класса, даже если они декларированы как private. Их в свою очередь можно разделить на

- **Члены (Member)** классы, определенные в блоке объемлющего класса;
- **Анонимные (Anonimous)** классы, без явного имени;
- **Локальные (Local)** классы, определенные в методах.

15.2. Вложенные классы

Рассмотрим пример статического вложенного класса 15.1. Он имеет доступ к приватным переменным, впрочем как и к нему имеет доступ объемлющий класс. Вызов его статических методов не требуют его создания.

```
1  package outer;
2  /**
3   * Вложенный класс
4   * @author daddym
5   */
6  public class Outer {
7      private int currentVal =1;
8      private static int defaultVal = 100;
9      private static class Nested{
10         static void sayStatic(){System.out.println("S Default " + defaultVal);}
11         void sayInstance(){System.out.println("I Default " + defaultVal);}
12     }
13     /**
14      * @param args the command line arguments
15      */
16     public static void main(String[] args) {
17         // TODO code application logic here
18         Outer.Nested.sayStatic();
19         Outer.Nested inst = new Outer.Nested();
20         inst.sayInstance();
21         inst.sayStatic();
22     }
23 }
```

Листинг 15.1: Вложенный класс

15.3. Анонимные классы

Анонимные классы полезнейший механизм внутренних. Это возможность создать указатель на объект определенного класса, не создавая явно сам класс. В качестве шаблона выступает имя существующего класса или интерфейса. Как демонстрирует 15.1 к имени справа дописывается блок, внутри которого вы набираете код, как будто это действительно тело класса наследующего выбранный.

```
1  /*
2   * To change this license header, choose License Headers in Project Properties.
3   * To change this template file, choose Tools | Templates
4   * and open the template in the editor.
5   */
6  package anonimus;
7
8  /**
9   *
10  * @author daddym
11  */
12 public class Anonimus {
13     static Anonimus a3 =  new Anonimus(){
14             void say(){System.out.println("Anonimus 3 " + this.getClass().getName());}
15         };
16     void say(){System.out.println("Anonimus 1 " + this.getClass().getName());}
17     /**
18      * @param args the command line arguments
19      */
20     public static void main(String[] args) {
21         Anonimus a1 = new Anonimus();
22         Anonimus a2 = new Anonimus(){
23             void say(){System.out.println("Anonimus 2 " + this.getClass().getName());}
24         };
25         a1.say();
26         a2.say();
27         a3.say();
28     }
29
30 }
```

Листинг 15.2: Анонимный класс

15.4. Перечисления в Java

Перечисление (enum) в Яве это специальный класс который неявно наследует javaEnum. В нем могут быть методы, включая main. Нельзя лишь создавать экземпляры этого класса. Значения констант по умолчанию задаются в круглых скобках. Точнее по умолчанию компилятор создает такой код который вы можете переопределить:

```
1 enum Season?{
2   WINTER(10),SUMMER(20);
3
4   private int value;
5
6   Season?(int value){
7     this.value=value;
8   }
9 }
```

Листинг 15.3: Инициализация перечисления

Глава 16

Лямбда-выражения

16.1. Синтаксис лямбда-выражений

Язык Ява не слишком хорошо отображает функциональное программирование, поскольку все функции в нем, изначально методы. Но с версии 8 появилась поддержка анонимной функции (лямбда):

```
() -> {}
```

В скобках параметры, если параметр один, скобки можно опустить, потом стрелка вправо и скобки содержащие код. Лямбда-выражение возвращает ссылку на определенную в ней функцию. Ее можно вызвать, присвоить или передать как параметр.

16.2. Ссылки на методы

Поскольку в версии Явы 8 появилась работа с функциями, добавились возможности получения ссылки на методы класса и его экземпляров. Для статического метода:

```
ContainingClass::staticMethodName
```

Для метода экземпляра:

```
containingObject::instanceMethodName
```

Для конструктора:

```
ClassName::new
```

16.3. Функциональные интерфейсы

Раз появились ссылки на функции, значит потребовались переменные позволяющие их хранить. В это роли были выбраны функциональные интерфейсы. Это просто интерфейс описывающий ровно оду функцию. В переменной такого типа хранится ссылка на описываемую в нем функцию. Можно ее вызвать вызвать после присваивания, по имени метода интерфейса..

```
1  package javaapplication35;
2  interface Fi{
3      C create(String s);
4  }
5  class C{
6      C(String s){
7          say(s);
8      }
9      void say(String s){System.out.println(s);}
10 }
11 /**
12  *
13  * @author daddym
14  */
15 public class JavaApplication35 {
16
17     /**
18      * @param args the command line arguments
19      */
20     public static void main(String[] args) {
21         Fi fi = C::new;
22         C cc = fi.create("Hello world!");
23         cc.say("Превед медвед!");
24     }
25 }
```

Листинг 16.1: Использование ФИ

Глава 17

Паттерны проектирования

17.1. Нововведения Java

В 11 Яве более 90 изменений по сравнению с десятой, но большинство вряд ли коснутся вас напрямую. Основное:

- можно написать аннотацию к параметру лямбда функции используя слово var как его тип;
- включен http клиент java.net.http, удивительно:
- изменения java.io, в частности null потоки;
- больше методов для String;
- java.nio.file.Files - возможность читать и писаит файлы за одну операцию, здравствуй Питон.

17.2. Обзор паттернов

Паттерны программирования, это хорошие решения для частных н при этом довольно типичных задач. Действительно, шаблоны хорошего стиля, собранные в книжке Design Patterns — Elements of Reusable Object-Oriented Software , но ею далеко не ограничивающиеся.

Как и любой другой прием программирования, паттерны можно применять в любом языке, с тем или иным уровнем удобства. Ява достаточно хорошо поддерживает многие из них, кое что прямо в языке, как например концепцию Интерфейсов и Тэговых интерфейсов.

-

17.3. Паттерн одиночка

Он же Singleton, реализации концепции создания одного и только одного экземпляра некоего объекта. Часто применяется для задач инициализации.

```
1  public class Singleton {
2    private static Singleton instance;
3    private Singleton () {};
4
5    public static Singleton getInstance() {
6      if (instance == null) {
7        instance = new Singleton();
8      }
9      return instance;
10   }
11 }
```

Листинг 17.1: Singleton

17.4. Паттерн композиция

Он же Composite , объединение объектов в древовидную структуру для представления иерархии от частного к целому. Позволяет клиентам обращаться к отдельным объектам и к группам объектов одинаково.

```
1  class Data{
2     Data parent;
3     private List<Data> childs = new <Data>LinkedList();
4     public void remove(){
5        if (null != parent  ){
6           parent.remove(this);
7           parent = null;
8        }
9     }
10    public void remove(Data ch){
11       childs.remove(ch);
12    }
13    Data root(){
14       if (null != parent)
15          return parent.root();
16       else
17          return this;
18    }
19 }
```

Листинг 17.2: Composite

17.5. Паттерн наблюдатель

Он же Observer, прозрачное отслеживание событий в коде. Позволяет объекту получать оповещения об изменении состояния других объектов и тем самым наблюдать за ними.

```
1  //Интерфейс наблюдателя
2  interface Observer {
3     void update (float temperature, float humidity, int pressure);
4  }
5  //Интерфейс наблюдаемого
6  interface Observable {
7     void registerObserver(Observer o);
8     void removeObserver(Observer o);
9     void notifyObservers(); //Вызывается наблюдаемым при ищменениях
10 }
```

Листинг 17.3: Observer

17.6. Метаданные и рефлексия

Метаданные (Metadata),

Список листингов

Список иллюстраций

Список таблиц