Министерство образования и науки Российской Федерации
Ярославский государственный университет им. П. Г. Демидова
Кафедра вычислительных и программных систем

Н. С. Лагутина

Лекции по языку Java

Конспект лекций

Ярославль
ЯрГУ
2017

Оглавление

1. Введение

История Java восходит к 1991 году, когда группа инженеров из компании Sun под руководством Патрика Нотона (Patrick Naughton) и члена Совета директоров (и разностороннего компьютерного волшебника) Джеймса Гослинга (James Gosling) занялась разработкой небольшого языка, который можно было бы использовать для программирования бытовых устройств, например, контроллеров для переключения каналов кабельного телевидения (cable TV switchboxes). Поскольку такие устройства не потребляют много энергии и не имеют больших микросхем памяти, язык должен был быть маленьким и генерировать очень компактные программы. Кроме того, поскольку разные производители могут выбирать разные центральные процессоры (Central Processor Unit— CPU), было важно не завязнуть в какой-то одной архитектуре компьютеров. Проект получил кодовое название "Green". Стремясь изобрести небольшой, компактный и машинонезависимый код, разработчики возродили модель, использованную при реализации первых версий языка Pascal заре эры персональных компьютеров. Никлаус Вирт, создатель языка Pascal, в свое время разработал машинонезависимый язык, генерирующий промежуточный код для некоей гипотетической машины. (Такие гипотетические машины часто называются виртуальными — например, виртуальная машина языка Java, или JVM.) Этот промежуточный код можно выполнять на любой машине, имеющей соответствующий интерпретатор. Инженеры, работавшие над проектом "Green также использовали виртуальную машину, что решило их основную проблему.

Однако большинство сотрудников компании Sun имели опыт работы с операционной системой UNIX, поэтому в основу разрабатываемого ими языка был положен язык C++, а не Pascal. В частности, они сделали язык объектно, а не процедурно-ориентированным. Сначала Гослинг решил назвать его Oak (Дуб). Возможно потому, что он любил смотреть на дуб, растущий прямо под окнами его офиса в компании Sun. Потом сотрудники компании Sun узнали, что слово Oak уже используется в качестве имени ранее созданного языка программирования, и изменили название Java в честь марки кофе Java, которая, в свою очередь, получила наименование одноимённого острова, поэтому на официальной эмблеме языка изображена чашка с парящим кофе. Существует и другая версия происхождения названия языка, связанная с аллюзией на кофе-машину как пример бытового устройства, для программирования которого изначально язык создавался.

В 1992 году в рамках проекта Green была выпущена первая продукция. Это было средство для чрезвычайно интеллектуального дистанционного управления. Оно имело мощность рабочей станции SPARK, помещаясь в коробочке размером 6x4x4 дюйма. К сожалению, ни одна из компаний- производителей электронной техники не заинтересовалась этим изобретением.

Затем группа стала заниматься разработкой устройства для кабельного телевидения, которое могло бы осуществлять новые виды услуг, например, включать видеосистему по требованию. И снова они не получили ни одного контракта. Забавно, что одной из компаний, отказавшихся подписать с ними контракт, руководил Джим Кларк (Jim Clark) --- основатель компании Netscape, впоследствии сделавшей очень много для успеха языка Java.

Весь 1993 год и половину 1994 года продолжались безрезультатные поиски покупателей продукции, разработанной в рамках проекта "Green". Проект "First Person, Inc."был прекращен в 1994 году. Тем временем в рамках Интернет разрасталась сеть World Wide Web. Ключом к этой сети является браузер, превращающий гипертекст в изображение на экране. В 1994

году большинство людей пользовалось браузером Mosaic, некоммерческим Web-браузером, разработанным в суперкомпьютерном центре Университета штата Иллинойс (University of Illinois) в 1993 году.

В своем интервью журналу Sun World Гослинг сказал, что в середине 1994 года разработчики языка поняли: "Браузер должен представлять собой одно из немногих приложений модной клиент-серверной технологии, в которой жизненно важным было бы именно то, что мы сделали: архитектурная независимость, выполнение в реальном времени, надежность, безопасность — вопросы, не являвшиеся чрезвычайно важными для рабочих станций".

Браузер был разработан Патриком Нотоном и Джонатаном Пэйном (Johnatan Payne). Позднее он превратился в современный браузер HotJava. Этот браузер был написан на языке Java, чтобы продемонстрировать всю его мощь. Однако разработчики не забывали о мощных средствах, которые теперь называются апплетами, наделив свой браузер способностью выполнять код внутри Web-страниц. Демонстрация технологии была представлена на выставке Sun World 95 23 мая 1995 года и вызвала всеобщее помешательство на почве языка Java (официальный день рождения технологии Java — 23 мая 1995 г.).

Компания Sun выпустила первую версию языка Java в начале 1996 года. Через несколько месяцев после нее появилась версия Java 1.02. Люди быстро поняли, что версия Java 1.02 не подходит для разработки серьезных приложений. Честно говоря, версия Java 1.02 была еще сырой. Ее преемница, версия Java 1.1, заполнила большинство зияющих провалов, намного улучшив возможность отражения и добавив новую модель событий для программирования графического пользовательского интерфейса. Несмотря на это, она все еще была довольно ограниченной.

Выпуск версии Java 1.2 стал основной новостью конференции JavaOne в 1998 году. В новой версии слабые средства для создания графического пользовательского интерфейса и графических приложений были заменены сложным и масштабным инструментарием. Это был шаг вперед, к реализации лозунга "Write Once, Run Anywhere"("Один раз напиши и везде выполняй"), выдвинутого при разработке предыдущих версий. Графические элементы стали оформлять в виде компонентов — появились JavaBeans, с которыми Java вошла в мир распределенных систем и промежуточного программного обеспечения, тесно связавшись с технологией CORBA. В декабре 1998 года через три дня (!) после выхода в свет название новой версии было изменено на громоздкое словосочетание Java 2 Standart Edition Software Development Kit Version 1.2 (стандартное издание пакета инструментальных средств для разработки программного обеспечения на языке Java 2, версия 1.2). Кроме стандартного издания пакета (Standart Edition) были предложены еще два варианта: Micro Edition для портативных устройств, например, для мобильных телефонов, и промышленная разработка Enterprise Edition для создания серверных приложений. Серверы должны взаимодействовать с базами данных — появились драйверы JDBC (Java DataBase Connection). Взаимодействие оказалось удачным, и многие системы управления базами данных и даже операционные системы включили, Java в свое ядро, например Oracle, Linux, MacOS X, AIX. Версии 1.3 и 1.4 стандартного издания пакета инструментальных средств намного совершеннее первоначального выпуска языка Java 2. Они обладают новыми возможностями и, разумеется, содержат намного меньше ошибок.

Такое быстрое и широкое распространение технологии Java не в последнюю очередь связано с тем, что она использует новый, специально созданный язык программирования, который так и называется — язык Java. Этот язык создан на базе языков Smalltalk, Pascal, C++ и др., вобрав их лучшие, по мнению создателей, черты и отбросив худшие. На этот счет есть разные мнения, но бесспорно, что язык получился удобным для изучения, написанные на нем программы легко читаются и отлаживаются. Язык Java становится языком обучения объектно-ориентированному программированию, так же, как язык Pascal был языком обуче-

ния структурному программированию. Недаром на Java уже написано огромное количество программ, библиотек классов.

В 2009 году состоялась покупка Sun компанией Oracle. С этого момента развитием Java занимается Oracle.

Релиз версии Java 7 состоялся 28 июля 2011 года. В финальную версию Java Standard Edition 7 не были включены все ранее запланированные изменения. Согласно плану развития, включение нововведений разбито на две части: Java Standard Edition 7 (без лямбд, проекта Jigsaw, и части улучшений Coin) и Java Standard Edition 8 (все остальное), намеченный на конец 2012 года. В новой версии, получившей название Java Standard Edition 7 (Java Platform, Standard Edition 7), помимо исправления большого количества ошибок было представлено несколько новшеств. Так, например, в качестве эталонной реализации Java Standard Edition 7 использован не проприетарный пакет JDK, а его открытая реализация OpenJDK, а сам релиз новой версии платформы готовился при тесном сотрудничестве инженеров Oracle с участниками мировой экосистемы Java, комитетом JCP (Java Community Process) и сообществом OpenJDK. Все поставляемые Oracle бинарные файлы эталонной реализации Java Standard Edition 7 собраны на основе кодовой базы OpenJDK, сама эталонная реализация полностью открыта под лицензией GPLv2 с исключениями GNU ClassPath, разрешающими динамическое связывание с проприетарными продуктами. К другим нововведениям относится интеграция набора небольших языковых улучшений Java, развиваемых в рамках проекта Coin, добавлена поддержка языков программирования с динамической типизацией, таких как Ruby, Python и JavaScript, поддержка загрузки классов по URL, обновленный XML-стек, включающий JAXP 1.4, JAXB 2.2a и JAX-WS 2.2 и другие.

Релиз версии Java 8 состоялся 19 марта 2014 года. Список нововведений:

1. Полноценная поддержка лямбда-выражений
2. Дополнительные функции обхода коллекций
3. Ключевое слово default в интерфейсах для поддержки функциональности по умолчанию
4. Динамическая подгрузка методов

Литература для изучения Java:

1. Шилдт Г. Java 8. Полное руководство (9-е издание). 2012.
2. Брюс Э. Философия Java. Библиотека программиста. Санкт-Петербург: Питер. 2014.
3. Уорбэртон Р. Лямбда-выражения в Java 8. Функциональное программирование. ДМК Пресс. 2015

Документация и другие интернет-источники:

1. Описание java8 https://docs.oracle.com/javase/8/docs/api/
2. Описание javaFX https://docs.oracle.com/javase/8/javafx/api/toc.htm
3. Учебные материалы от разработчикав java https://docs.oracle.com/javase/tutorial/
4. Среда разработки netbeans https://netbeans.org/downloads/
5. Установка java https://www.oracle.com/ru/java/index.html

2. Основные особенности Java

2.1. Удобство. Проектировщики Java пытались разработать язык, который могли бы быстро изучить программисты. Также они хотели, чтобы язык был знаком большинству программистов, для простоты перехода. Отсюда, в Java проектировщиками было удалено множество сложных особенностей, которые существовали в C и C++. Особенности, такие как манипуляции указателя, перегрузка оператора и т.д. в Java не существуют. Java не использует goto инструкцию, а также не использует файлы заголовка. Конструкции подобно struct и union были удалены из Java. Поскольку в Java каждая сложная структура данных — это объект, память под такие структуры резервируется в куче (heap) с помощью оператора new. Реальные адреса памяти, выделенные этому объекту, могут изменяться во время работы программы, но вам не нужно об этом беспокоиться, поскольку Java - система с так называемым сборщиком мусора. Сборщик мусора запускается каждый раз, когда система простаивает, либо когда Java не может удовлетворить запрос на выделение памяти.

2.2. Объектно-ориентированность. В Java всё может быть объектом. Так основное внимание уделяется свойствам и методам, которые оперируют данными в нашем приложении и нет концентрации только на процедурах. Свойства и методы вместе описывают состояние и поведение объекта. В Java мы будем наталкиваться на термин метод очень часто, с ним мы будем должны познакомиться. Термин метод используется для функций.

2.3. Устойчивость. Java - язык со строгим контролем типов, так что требуется явное объявление метода. Java проверяет код во время трансляции и во время интерпретации. Таким образом устраняются некоторые типы ошибок при программировании. Java не имеет указателей и соответственно арифметических операций над ними. Все данные массивов и строк проверяются во время выполнения, что исключает возможность выхода за границы дозволенного. Преобразование объектов с одного типа на другой также проверяется во время выполнения.

В традиционных средах программирования, программист должен был вручную распределять память, и в конце программы имел явное количество свободной памяти. Возникали проблемы, когда программист забывал освобождать память. В Java, программист не должен беспокоиться о проблеме, связанной с освобождением памяти. Это делается автоматически.

2.4. Безопасность. Важной особенностью технологии Java является гибкая система безопасности, в рамках которой исполнение программы полностью контролируется виртуальной машиной. Любые операции, которые превышают установленные полномочия программы (например, попытка несанкционированного доступа к данным или соединения с другим компьютером) вызывают немедленное прерывание.

Тип и видимость членов класса при трансляции встраиваются внутрь файла *.class (файла с байт-кодом). Интерпретатор Java пользуется этой информацией в процессе выполнения кода, так что не существует способа получить доступ к закрытым переменным класса извне.

Первый уровень - это безопасность, обеспеченная языком Java. Свойства и методы описываются в классе, и к ним можно обратиться только через интерфейс, обеспеченный классом. Java не позволяет никаких операций с указателями, таким образом запрещает прямой доступ к памяти. Избегается переполнение массивов. Проблемы, связанные с безопасностью и мобильностью, скрыты.

На следующем уровне компилятор, прежде чем приступить к компиляции кода, проверяет безопасность кода и затем следует в соответствии с протоколами, установленными Java.

Третий уровень - это безопасность, обеспеченная интерпретатором. Прежде, чем байт-код будет фактически выполнен, он является полностью укрытым верификатором.

Четвертый уровень заботится о загрузке классов. Загрузчик класса гарантирует, что класс не нарушает ограничения доступа прежде, чем он загружен в систему.

2.5. Кроссплатформенность. Java может использоваться для разработки приложений, которые работают на различных платформах, операционных системах и графических интерфейсов пользователя. Java предназначен также для поддержки сетевых приложений. Таким образом Java широко используется как инструмент разработки в среде подобной Internet, где существуют различные платформы. Программы на Java транслируются в байт-код, выполняемый виртуальной машиной Java (JVM) — программой, обрабатывающей байтовый код и передающей инструкции оборудованию как интерпретатор.

Достоинством подобного способа выполнения программ является полная независимость байт-кода от операционной системы и оборудования, что позволяет выполнять Java-приложения на любом устройстве, для которого существует соответствующая виртуальная машина.

Платформа, независимая на вторичном уровне означает, что откомпилированный двоичный файл может быть выполнен на различных платформах, не перетранслируя код, если они имеют Виртуальную Машину Java, которая функционирует как интерпретатор. Для программ, которые мы записываем в C и C++ или на любом другом языке, компилятор преобразует набор команды в машинный код или команды процессора. Эти команды являются специальными для нашего процессора. В результате, если мы хотим использовать этот код в некоторой другой системе, мы должны найти компилятор для этой системы, и мы должны компилировать код еще раз так, чтобы мы имели машинный код, определенный для этой машины. Когда мы посмотрим на среду развития Java, мы увидим разделение на две части: Компилятор Java и Интерпретатор Java. В отличие от C и C++, компилятор Java преобразует исходный текст в байт-коды, которые являются машинно-независимыми. Байт-коды - это только части команд Java, разрезанные на байты, которые могут быть декодированы любым процессором.

Java-программа представляет собой набор классов, для каждого класса при компиляции создается двоичный файл с расширением class, содержащий кроме собственно байт-кода методов еще и описание структуры класса (описание данных и заголовки методов).

2.6. Области применения Java. Основные области применения Java это большие web приложения, банковские desktop приложения и мобильная разработка для Android. T.e. она сразу попадает в три популярные категории.

Согласно утверждениям с сайта http://www.java.com/ru/about/ :

1. Java используется на 97% корпоративных настольных ПК
2. Java используется на 89% настольных ПК в США
3. 9 млн разработчиков на Java в мире
4. Java используется в 3 млрд мобильных телефонов
5. Java входит в комплект поставки 100% всех проигрывателей дисков Blu-ray
6. Используется 5 млн Java Card
7. Java используется в 125 млн ТВ-устройств

Особое место занимает Java Enterprise Edition, это реализация Java для создания корпоративного (Enterprise) программного обеспечения, многозвенные распределённые системы, приложения масштаба предприятия: различные банковские системы, системы для предприятий планирования ресурсов (ERP-системы), веб-сервисы и т. д.

3. Введение в язык Java

3.1. Пример. Язык Java требует, чтобы весь программный код был заключен внутри поименованных классов. Приведенный ниже текст примера надо записать в файл Variables.java. Обязательно проверьте соответствие прописных букв в имени файла тому же в названии содержащегося в нем класса.

```
package example;
public class Variables {
    public static void main(String args[]) {
        double a = 3;
        double b = 4;
        double c;
        c = Math.sqrt(a * a + b * b);
        System.out.println("c = " + c);
    }
}
```

Во второй строке использовано зарезервированное слово class. Оно говорит транслятору, что мы собираемся описать новый класс. Полное описание класса располагается между открывающей фигурной скобкой в первой строке и парной ей закрывающей фигурной скобкой в последней строке. Фигурные скобки в Java используются точно так же, как в языках C и C++.

Разбивая следующую строку на отдельные лексемы, мы сразу сталкиваемся с ключевым словом public. Это — модификатор доступа, который позволяет программисту управлять видимостью любого метода и любой переменной. В данном случае модификатор доступа public означает, что метод main виден и доступен любому классу. Существуют еще 2 указателя уровня доступа — private и protected.

Следующее ключевое слово — static. С помощью этого слова объявляются методы и переменные класса, используемые для работы с классом в целом. Методы, в объявлении которых использовано ключевое слово static, могут непосредственно работать только с локальными и статическими переменными.

Достаточно часто создаются методы, которые возвращают значение того или иного типа: например, int для целых значений, float - для вещественных или имя класса для типов данных, определенных программистом. В нашем случае нужно просто вывести на экран строку, а возвращать значение из метода main не требуется. Именно поэтому и был использован модификатор void.

Наконец, мы добрались до имени метода main. Все существующие реализации Java-интерпретаторов, получив команду интерпретировать класс, начинают свою работу с вызова метода main. Java-транслятор может оттранслировать класс, в котором нет метода main. А вот Java-интерпретатор запускать классы без метода main не умеет.

Все параметры, которые нужно передать методу, указываются внутри пары круглых скобок в виде списка элементов, разделенных символами ";"(точка с запятой). Каждый элемент списка параметров состоит из разделенных пробелом типа и идентификатора. Даже если у метода нет параметров, после его имени все равно нужно поставить пару круглых скобок. В примере, который мы сейчас обсуждаем, у метода main только один параметр, правда довольно сложного типа.

Элемент String args[] объявляет параметр с именем args, который является массивом объектов — представителей класса String. Обратите внимание на квадратные скобки, стоящие после идентификатора args. Они говорят о том, что мы имеем дело с массивом, а не с одиночным элементом указанного типа.

В конце выполняется метод println объекта out. Объект out объявлен в классе OutputStream и статически инициализируется в классе System.

Рассмотрим общие аспекты синтаксиса языка Java. Программы на Java — это набор пробелов, комментариев, ключевых слов, идентификаторов, литеральных констант, операторов и разделителей.

3.2. Комментарии. Хотя комментарии никак не влияют на исполняемый код программы, при правильном использовании они оказываются весьма существенной частью исходного текста. Существует три разновидности комментариев: комментарии в одной строке, комментарии в нескольких строках и, наконец, комментарии для документирования. Комментарии, занимающие одну строку, начинаются с символов // и заканчиваются в конце строки. Такой стиль комментирования полезен для размещения кратких пояснений к отдельным строкам кода:

```
a = 42; // comment
```

Для более подробных пояснений вы можете воспользоваться комментариями, размещенными на нескольких строках, начав текст комментариев символами /* и закончив символами */ При этом весь текст между этими парами символов будет расценен как комментарий и транслятор его проигнорирует.

```
/*
* Этоткоднесколькозамысловат
* Попробуюобъяснить:
*/
```

Третья, особая форма комментариев, предназначена для сервисной программы javadoc, которая использует компоненты Java-транслятора для автоматической генерации документации по интерфейсам классов. Соглашение, используемое для комментариев этого вида, таково: для того, чтобы разместить перед объявлением открытого (public) класса, метода или переменной документирующий комментарий, нужно начать его с символов /** (косая черта и две звездочки). Заканчивается такой комментарий точно так же, как и обычный комментарий — символами */. Программа javadoc умеет различать в документирующих комментариях некоторые специальные переменные, имена которых начинаются с символа @. Вот пример такого комментария:

```
/**
* Этотклассумеетделатьзамечательныевещи. Советуемвсякому, кто
* захочетнаписатьещеболеесовершенныйкласс, взятьеговкачестве
* базового.
* @see Java. applet. Applet
* ©author Patrick Naughton
* @version 1. 2
*/
class CoolApplet extends Applet { /**
* Уэтогометодадвапараметра:
* @param key — этоимяпараметра.
* @param value — этозначениепараметрасименем key.
*/ void put (String key, Object value) {
```

После начальной комбинации символов /** располагается текст, являющийся главным описанием класса, переменной или метода. Далее можно вставлять различные дескрипторы. Каждый дескриптор @ должен стоять первым в строке. Несколько дескрипторов одного и того же типа необходимо группировать вместе. Встроенные дескрипторы (начинаются с фигурной скобки) можно помещать внутри любого описания.

Утилита javadoc в качестве входных данных принимает файл с исходным кодом программы. Генерирует несколько HTML файлов, содержащих документацию по этой программе. Информация о каждом классе будет содержаться в отдельном HTML файле. Кроме того, создается дерево индексов и иерархии.

3.3. **Зарезервированные ключевые слова**. Зарезервированные ключевые слова — это специальные идентификаторы, которые в языке Java используются для того, чтобы идентифицировать встроенные типы, модификаторы и средства управления выполнением программы. На сегодняшний день в языке Java имеется 59 зарезервированных слов. Эти ключевые слова совместно с синтаксисом операторов и разделителей входят в описание языка Java. Они могут применяться только по назначению, их нельзя использовать в качестве идентификаторов для имен переменных, классов или методов (class if int).

3.4. **Идентификаторы**. Идентификаторы используются для именования классов, методов и переменных. В качестве идентификатора может использоваться любая последовательность строчных и прописных букв, цифр и символов _ (подчеркивание) и $(доллар). Идентификаторы не должны начинаться с цифры, чтобы транслятор не перепутал их с числовыми литеральными константами, которые будут описаны ниже. Java — язык, чувствительный к регистру букв. Это означает, что, к примеру, Value и VALUE — различные идентификаторы. Существует соглашение об именах.

4. Типы данных, литералы, переменные

Типы данных определяют основные возможности любого языка. Кроме того, Java является строго типизированным языком, а потому четкое понимание модели типов данных очень помогает в написании качественных программ.

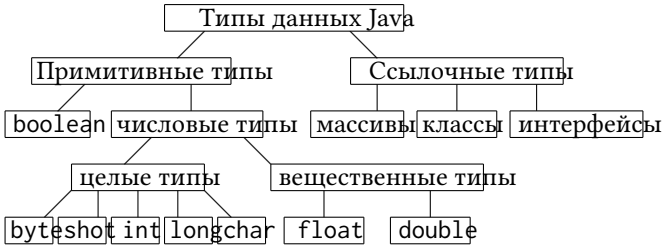

Рис. 4.1. Схема типов данных Java

4.1. Примитивные типы данных. В Java определено 8 примитивных типов данных. К ним относятся четыре целочисленных **byte**, **short**, **int**, **long**, два вещественных **float** и **double**, символьный **char** и логический **boolean** типы данных. Их основные характеристики приведены в табл. 4.1. В отличие от C++ в Java нет типа данных <<указатель>>.

Главная особенность примитивных типов данных в Java состоит в том, что переменные этих типов не являются объектами. Переменные всех остальных типов, включая массивы и переменные строкового типа **String**, являются ссылками на объекты (объектными ссылками). Подробнее они будут рассмотрены далее.

В языке Java диапазоны значений типов не зависят от машины, на которой выполняется программа. Это облегчает страдания программистов, которым необходимо переносить программное обеспечение с одной платформы на другую и даже из одной операционной системы — в другую на одной и той же платформе.

В языке Java понятие беззнаковых чисел отсутствует. Все числовые типы этого языка — знаковые.

Тип byte — это знаковый 8-битовый тип. Его диапазон — от -128 до 127. Он лучше всего подходит для хранения произвольного потока байтов, загружаемого из сети или из файла. Если речь не идет о манипуляциях с битами, использования типа byte, как правило, следует

Тип данных	Размер, байт	Допустимые значения
byte	1	$--128 \div 127$
short	2	$--32768 \div 32767$
int	4	$--2147483648 \div 2147483647$
long	8	$--2^{63} \div 2^{63}--1$
float	8	$\approx 1.4 \cdot 10^{--45} \div 3.4 \cdot 10^{38}$
double	16	$\approx 1.7 \cdot 10^{--324} \div 4.9 \cdot 10^{308}$
char	2	$0 \div 65535$
boolean	---	true, false

Таблица 4.1. Примитивные типы данных в Java

избегать. Для нормальных целых чисел, используемых в качестве счетчиков и в арифметических выражениях, гораздо лучше подходит тип int.

short — это знаковый 16-битовый тип. Его диапазон — от -32768 до 32767. Это, вероятно, наиболее редко используемый в Java тип.

Тип int служит для представления 32-битных целых чисел со знаком. Диапазон допустимых для этого типа значений — от -2147483648 до 2147483647. Чаще всего этот тип данных используется для хранения обычных целых чисел со значениями, достигающими двух миллиардов. Этот тип прекрасно подходит для использования при обработке массивов и для счетчиков.

Тип long предназначен для представления 64-битовых чисел со знаком. Его диапазон допустимых значений достаточно велик даже для таких задач, как подсчет числа атомов во вселенной.

Числа с плавающей точкой, часто называемые в других языках вещественными числами, используются при вычислениях, в которых требуется использование дробной части.

В переменных с обычной, или одинарной точностью, объявляемых с помощью ключевого слова float, для хранения вещественного значения используется 32 бита.

```
float f;
float f2 = 3. 14F;
```

В случае двойной точности, задаваемой с помощью ключевого слова double, для хранения значений используется 64 бита. Все трансцендентные математические функции, такие, как sin, cos, sqrt, возвращают результат типа double.

```
double d;
double pi = 3. 14159265358979323846;
```

Длина char - 2 байта, это непосредственно следует из того, что все символы Java описываются стандартом Unicode. Поскольку в Java для представления символов в строках используется кодировка Unicode, разрядность типа char в этом языке — 16 бит. В нем можно хранить десятки тысяч символов интернационального набора символов Unicode. Диапазон типа char — 0..65536. Unicode — это объединение десятков кодировок символов, он включает в себя латинский, греческий, арабский алфавиты, кириллицу и многие другие наборы символов. Хотя величины типа char и не используются, как целые числа, вы можете оперировать с ними так, как если бы они были целыми. Это дает вам возможность сложить два символа вместе, или инкрементировать значение символьной переменной. В приведенном ниже фрагменте кода мы, располагая базовым символом, прибавляем к нему целое число, чтобы получить символьное представление нужной нам цифры.

```
int three = 3;
char one = '1';
char four = (char) (three+ one);
```

В результате выполнения этого кода в переменную four заносится символьное представление нужной нам цифры — '4'. Обратите внимание — тип переменной one в приведенном выше выражении повышается до типа int, так что перед занесением результата в переменную four приходится использовать оператор явного приведения типа.

Переменная типа boolean имеет два значения: false и true. Они используются для вычисления логических выражений. Преобразования булевских переменных в целочисленные и наоборот невозможны.

4.2. Преобразования типов. Преобразования типов в Java делятся на неявные и явные. Неявные преобразования осуществляются компилятором автоматически. Правила автоматического приведения в Java являются значительно более жёсткими, чем в C++. В отличие от

C++ неявные преобразования в Java осуществляются только в тех случаях, когда возможно гарантировать, что в результате преобразования не произойдёт потеря значимости.

Автоматическое преобразование типов в Java.

```
    class prog_3{
        public static void main(String args[]) {
        int num;
        num=200;
        double rez;
        rez=num;
        System.out.println (rez);
        }
    }
```

Результатом работы программы будет вывод на экран числа 200.0.

Пример выше демонстрирует автоматическое преобразование переменной одного типа к другому. Мы просто присвоили переменной типа "double"значение переменной типа "int". А компилятор автоматически произвел преобразование числа 200 из целого в дробное. Такое автоматическое преобразование переменной возможно не всегда. Должно соблюдаться несколько условий: длина конечного типа должна быть больше длины исходного типа, оба типа должны быть совместимыми.

Следующая строка, являющаяся корректной с точки зрения языка C++, в Java вызовет ошибку компиляции:

```
int x = 2.5; // попытканеявногопреобразованияот double кint
```

поскольку преобразование от **double** к **int** может привести к потере значимости.

Говоря про длину типа, следует учитывать, что результирующий тип должен быть больше, чем исходный. Например, тип "double"больше типа "int"(диапазон его значений больше), поэтому данные типа "int"легко могут быть преобразованы к типу "double". А вот обратное невозможно, поскольку тип "double"имеет значительно больший диапазон чем тип "int".

Второе условие, при котором возможно автоматическое преобразование типов – совместимость типов. Здесь все достаточно просто. Любые числовые типы, будь то целые числа или дробные, совместимы между собой. Но, например, тип char и тип boolean – не совместимы.

Приведём более формальные правила неявного преобразования типов. Автоматическое преобразование типов при присваивании производится, если выполнены следующие условия:

- требуется приведение целочисленного типа или типа **char** к какому-либо числовому типу или вещественного к вещественному;
- тип, к которому требуется приведение, имеет больший размер, чем исходный.

Автоматическое преобразование типов в выражениях осуществляется

- для аргументов операции конкатенации строк
- если один из аргументов бинарной операции имеет тип **double**, второй приводится к типу **double**;
- иначе: если один из аргументов имеет тип **float**, второй приводится к типу **float**;
- иначе: если один из аргументов имеет тип **long**, второй приводится к типу **long**;
- иначе оба аргумента приводятся к типу **int**.

Здесь следует отметить последний пункт. В некоторых случаях его применение может приводить к не вполне естественным результатам. Например, следующий фрагмент является ошибочным с точки зрения языка:

```
short a = 5;
short b = a + 1; // ошибка: требуется short, а не int
```

Ошибка связана с тем, что при добавлении 1 к значению переменной **a** произошло преобразование результата к типу **int**. Так как преобразование от типа **int** к **short** может приводить к потере значимости, то оно должно поризводиться явно:

```
short b = (short)(a + 1);
```

Причиной такого <<странного>> поведения является тот факт, что типы данных **byte** и **short** создавались не для выполнения арифметических операций, а для обмена с внешними устройствами. Для выполнения арифметических операций следует использовать типы **int** и **long**.

Явное преобразование (приведение) типов в Java.

```
class prog_3{
    public static void main(String args[]) {
    int num;
    num=600;
    short rez;
    rez=num;
    System.out.println (rez);
    }
}
```

Эта программа работать не будет.

Необходимо переменную большего типа явно преобразовать к меньшему. Например, в программе выше, нужно изменить строку 6 на такую:

```
rez=(short)num;
```

Переменную "num"мы принудительно преобразовали к типу "short указав его в круглых скобках перед названием переменной.

При приведении типов следует помнить, что такая процедура опасна тем, что может привести к потере значений. В нашем примере, в качестве данных, мы использовали число "600". Оно прекрасно вписывается в диапазон чисел и типа "int и типа "short". Но, давайте для примера возьмем число "60000". Это число физически не поместится в переменную типа "short". В результате получим: -5536 Явное преобразование совместимых типов осуществляется по следующим правилам:

- сужение целочисленных типов производится путём усечения старших битов;
- усечение вещественных типов при присвоении их целым производится путём отбрасывания дробной части и применения предыдущего пункта.

Синтаксис явного преобразования типов совпадает с синтаксисом преобразования типов в C (см. пример выше).

Наконец, некоторые преобразования типов не являются допустимыми. В число таких преобразований входят: любые преобразования к типу **boolean**, преобразования классов к примитивным типам и примитивных типов к классам (с некоторыми исключениями), и т. д. Попытка выполнения таких преобразований будет приводить к ошибкам на этапе компиляции.

4.3. Литералы. Литералы позволяют задать в программе значения для числовых, символьных и строковых выражений, а также null-литералов.

Целочисленные литералы могут задаваться в десятичном, восьмеричном и шестнадцатеричном виде. Литералы в десятичном виде записываются непосредственно, для остальных

используются префиксы, аналогичные префиксам в языке C: <<0>> — для восьмеричных и <<0x>> — для шестнадцатеричных литералов. Например, целочисленный литерал 47 может быть представлен как 47, 057 или 0x2F. Все целочисленные литералы автоматически приводятся к минимально возможному типу, но не выше **int**. Литералы типа **long** должны иметь суффикс <<L>> или <<l>> (например, 47L).

Литералы с плавающей точкой могут представляться в обычном либо в научном формате. Литералы типа **float** должны иметь суффикс <<F>> или <<f>> (например, 3.5F), а литералы типа **double** — суффикс <<D>> или <<d>>, который может опускаться (3.5D или просто 3.5). Числа с плавающей точкой, не имеющие суффикса F (например 3.402), всегда рассматриваются как числа типа double.

В языке Java есть три специальных числа плавающей точкой: положительная бесконечность; отрицательная бесконечность; не число.

Они используются для обозначения переполнения и ошибок. Например, результат деления положительного числа на 0 равен положительной бесконечности. Вычисление 0/0 или извлечение квадратного корня из отрицательного числа равно NaN.

Для их обозначения введены константы Double.POSITIVE_INFINITY, Double.NEGATIVE_INFINITY и Double.NaN (а также соответствующие константы типа float).

Однако на практике они редко используются. В частности, для того, чтобы убедиться, что некий результат равен константе Double.NaN, нельзя выполнить проверку

```
if (x == Double.NAN) // Тождественноложное; условие.
```

Все величины, "не являющиеся числами считаются разными. Однако можно вызывать метод Double.isNaN():

```
if (Double.IsNaN(x)) // Проверка, являетсялихне" числом".
```

Символьные литералы ограничиваются одинарными кавычками. Также возможно представление в шестнадцатеричном виде в Unicode. В последнем случае литерал записывается в виде последовательности четырёх шестнадцатеричных цифр с префиксом \u. Например, литерал 'ю' может быть представлен в виде '$\backslash u044E$'.

Строковые литералы ограничиваются двойными кавычками ("abcd").

Например,'H' — это символ. Он отличается от "H строки, состоящей из единственного символа. Во-вторых, тип char обозначает символы, представленные в формате Unicode. Вы можете не знать этот формат и не беспокоиться о нем, если не разрабатываете приложения, в которых нужна поддержка языков, помимо английского.

Поскольку формат Unicode был разработан для обработки практически всех символов всех письменных языков, существующих в мире, он является 2-байтовым кодом. В нем допускается 65536 символов, из которых обычно используется около 35000. Формат Unicode намного богаче набора ASCII, представляющего собой 1-байтовый код, состоящий из 128 символов, или широко используемого расширения ISO 8859-1, с помощью которого можно закодировать 256 символов.

Этот набор символов (который программисты часто называют множеством символов представляет собой подмножество формата Unicode. Точнее, эти символы являются первыми 256 кодировки Unicode.

Таким образом, код символов 'a', ' 1', ' [' и 'a' в формате Unicode не превышает 256. Коды символов в формате Unicode лежат в диапазоне от 0 до 65535, однако обычно они выражаются как шестнадцатеричные величины от '\u0000' до '\uFFFF' (в то время как в формате ISO 8859-1 их диапазон ограничивается числами ' \u0000 ' и '\uOOFF').

Префикс \u означает, что символ записан в формате Unicode, а следующие за ним четыре шестнадцатеричные цифры задают сам символ. Например, \u2122 — это символ торговой

марки. Более подробную информацию о формате Unicode вы можете найти на vveb-странице http : //www.unicode.org.

Определены два логических литерала **true** и **false**, *не равные* целым литералам 1 и 0 соответственно.

4.4. Переменные. Переменная — это основной элемент хранения информации в Java-программе. Переменная характеризуется комбинацией идентификатора, типа и области действия. В зависимости от того, где вы объявили переменную, она может быть локальной, например, для кода внутри цикла for, либо это может быть переменная экземпляра класса, доступная всем методам данного класса. Локальные области действия объявляются с помощью фигурных скобок.

Основная форма объявления переменной такова:

тип идентификатор [= значение] [, идентификатор [= значение 7...];

Тип — это либо один из встроенных типов, то есть, byte, short, int, long, char, float, double, boolean, либо имя класса или интерфейса.

```
double a = 3, b = 4, c;
```

Объявление переменных в Java аналогично объявлению переменных в C++. Любая переменная может быть инициализирована в момент описания любым допустимым выражением. Область действия и время жизни локальной переменной ограничивается блоком, в котором эта переменная описана.

В отличие от C++ не допускается объявление во внутреннем блоке переменной с тем же именем, что и переменная во внешнем блоке (кроме случая, когда переменная во внешнем блоке является аргументом метода).

Переменные, объявленные со спецификатором **final**, являются неизменяемыми (константами). Например:

```
final double PI = 3.141592653589793116;
```

4.5. Массивы. Массив – это группа однотипных переменных, доступ к которым осуществляется по имени массива и индексу переменной в массиве. Массивы бывают 2-х видов: одномерные и многомерные.

Массивы в Java являются объектами, поэтому на них распространяются все правила работы с объектами Однако, поскольку массивы являются одними из наиболее широко применяемых структур данных, они будут рассмотрены здесь.

Для объявления одномерного массива используется синтаксис, аналогичный описанию переменных, но к имени переменной либо к имени типа данных добавляются пустые квадратные скобки. Например:

```
int a[]; // первыйвариант
int[] b; // второйвариант
```

При этом сам массив *не создаётся*. Для создания массива следует использовать операцию **new**:

```
a = new int[50];
```

Таким образом, процесс создания массива происходит в 2 этапа: Создается сам массив. Резервируется место в памяти под этот массив. Массивы в Java всегда определяются динамически, хотя ссылки на них задаются статически.

Возможно одновременное объявление и выделение памяти:

```
int a[] = new int[50];
```

Возможна также инициализация массива при его создании:

```
int a[] = { 20, 50, 166, 72, 0, −53 };
```

Для обращения к элементам массива используется операция **[]**. Элементы массива нумеруются с нуля. Попытка обращения к несуществующему элементу массива приводит к ошибке времени выполнения. Массивы поддерживают получение информации о своих размерах посредством экземплярной переменной **length**. Например a.length равна 6. Последний элемент массива a можно записать так: a [a.length - 1].

Рассмотрим пример:

```
public class ExampleArray {

    public static void main(String args[]) {

        int a[] = {20, 50, 166, 72, 0, −53};
        double aMin = a[0], aMax = a[0];
        for (int i = 1; i < a.length; i++) {
            if (a[i] < aMin) {
                aMin = a[i];
            }
            if (a[i] > aMax) {
                aMax = a[i];
            }
        }
        double range = aMax − aMin;
        System.out.println(range);
    }
}
```

Здесь вычисляется диапазон значений массива.

Многомерные массивы представляют собой массивы массивов. Они могут создаваться аналогично одномерным:

```
int a[][] = new int[5][2];
```

Возможно создание непрямоугольных массивов. Способ работы с такими массивами иллюстрируется следующим примером.

```
 char c[][];
```

Затем определяем внешний массив:

```
c = new char[3][];
```

Становится ясно, что c — массив, состоящий из трех элементов-массивов. Теперь определяем его элементы-массивы:

```
c[0] = new char[2];
c[1] = new char[4];
c[2] = new char[3];
```

После этих определений переменная c.length равна 3, c[0].length равна 2, c[1].length равна 4 и c[2].length равна 3. Наконец, задаем начальные значения c[0][0] = 'a', c[0][1] = 'r', c[1][0] = 'r',c[1][1] = 'a',c[1][2] = 'y' и т.д.

```
 class PascalTriangle{
   public static void main(String[] args) {
        final int LINES = 10;
        int[][] p = new int[LINES][];
```

1.	()	[]	.	
2.	++	−−	~	!
3.	*	/	%	
4.	+	−		
5.	<<	>>	>>>	
6.	<	>	<=	>=
7.	==	!=		
8.	&			
9.	^			
10.	|			
11.	&&			
12.	||			
13.	?:			
14.	=	*operator=*		

Таблица 4.2. Приоритеты операций

```
            p[0] = new int[1];
            System.out.println(p[0][0] = 1);
            p[1] = new int[2];
            p[1][0] = p[1][1] = 1;
            System.out.println(p[1][0] + " " + p[1][1]);
            for (int i = 2; i < LINES; i++) {
                p[i] = new int[i + 1];
                System.out.print((p[i][0] = 1) + " ");
                for (int j = 1; j < i; j++) {
                   System.out.print((p[i][j] = p[i − 1][j − 1] + p[i − 1][j]) + " ");
                }
                System.out.println(p[i][i] = 1);
            }
        }
    }
```

4.6. Общая характеристика операций. Большинство операций Java аналогичны операциям C++, хотя некоторые из них функционируют чуть иначе. Поскольку в Java нет указателей, естественным представляется отсутствие операций взятия адреса (&), разыменования (*) и косвенной адресации (-->). Операция . (<<точка>>) в Java используется для обращения к полям и методам объектов через объектные ссылки (более подробно см. п. 5.4).

В последующих пунктах описаны основные операции Java. Приоритеты операций приведены в табл. 4.2.

4.7. Арифметические операции. К арифметическим относятся следующие операции: унарные операции сохранения и изменения знака (+ и --); унарные операции инкремента и декремента (++ и ----); бинарные операции сложения, вычитания, умножения, деления и нахождения остатка от деления (+, --, *, /, %), а также операции с присваиванием (+=, --=, *=, /=, %=).

Арифметические операции применяются к числовым типам и имеют результат также числового типа. Операция деления для целочисленных типов даёт результат целочисленного типа (неполное частное), для вещественных — вещественного.

Если при выполнении операций в целочисленной арифметике происходит переполнение, то один или несколько старших битов результата могут быть усечены (при этом, поскольку старший бит отвечает за знак числа, возможно изменение знака) без генерации со-

общения об ошибке. Единственной операцией в целочисленной арифметике, приводящей к ошибке, является операция деления на 0.

При выполнении операций в вещественной арифметике ошибка не возникает никогда: при переполнениях и делении на ноль результат получает одно из специальных значений NaN (<<не число>>), Infinity (<<плюс бесконечность>>), --Infinity (<<минус бесконечность>>). При необходимости их можно получить, обратившись к константам **NaN**, **POSITIVE_INFINITY**, **NEGATIVE_INFINITY** соответственно в классах **Float** и **Double**. Например, результат вычисления выражения --3.0f/0.0f равен **Float**.**NEGATIVE_INFINITY**, а результат вычисления выражения 0.0/0.0 равен **Double**.**NaN**.

Подумайте: одинаковы ли следующие выражения

```
byte b = 1;
b = b + 10; // Ошибка!
b += 10; // Правильно!
```

4.8. Операции сравнения. К операциям сравнения относятся операции ==, **!**=, <, >, <=, >=. Единственным их отличием от соответствующих операций C++ является то, что их результат всегда имеет тип **boolean**.

4.9. Логические операции. Логические операции применяются к аргументам типа **boolean** и дают результат типа **boolean**. К логическим относятся: унарная операция логического <<не>> (**!**), бинарные операции логического <<и>> (**&&**, **&**), <<или>> (|, ||) и <<исключающего или>> (^), а также аналогичные операции с присваиванием (**&**=, |=, ^=).

Операции **&&** и || вычисляют значение своего второго аргумента только в том случае, если результат невозможно определить по значению первого. В отличие от них операции **&** и | всегда вычисляют значения обоих своих аргументов. Например:

```
int x = 10, y = 10;
if(x == 10 || ++y == 10)
    System.out.println(y);
```

печатает <<10>>, так как второй аргумент операции || не вычисляется. Если заменить || на |, результатом работы будет <<11>>.

4.10. Условная операция. В Java имеется аналогичная C++ условная операция **?:**. Её синтаксис:

```
  условие
? выражение1 : выражение2
```

Если *условие* истинно, то результатом этой операции является *выражение1*, иначе — *выражение2*. При этом выражение, не являющееся результатом операции, не вычисляется. Существенно, что в Java условие должно иметь тип **boolean** (см. также п. 4.12).

4.11. Операция присваивания и оператор-выражение. Все операции языка Java (как и во всех других языках) имеют возвращаемое значение. Однако, кроме этого, они (как и в C, C++, но не в других языках) могут также изменять свои операнды. К таким операциям относятся операции **++**, **----**, **=** и все операции с присваиванием. По этой причине в языках C, C++ и Java нет *оператора присваивания*, но есть *операция присваивания* и *оператор-выражение*.

Операция присваивания — это бинарная операция, которая присваивает значение своего второго аргумента первому. Первый аргумент операции присваивания должен быть *lvalue*, т. е. <<допускающим присваивание>>. Результат операции присваивания равен значению её второго аргумента. Например, следующая операция

```
x = y = 10
```

осуществляет присваивание переменной **y** значения 10, после чего результат этой операции (т. е. 10) присваивается переменной **x**.

Оператор-выражение имеет синтаксис:

```
expression;
```

Таким образом, добавление точки с запятой к любому выражению превращает его в оператор, а возможность соединять в одном выражении несколько операций позволяет одним оператором изменить сразу несколько переменных, что невозможно при использовании оператора присваивания.

В отличие от C++, в Java есть дополнительное требование, чтобы последняя операция, входящая в оператор-выражение, изменяла свои операнды, т. е. требование отсутствия бесполезного кода (code having no effect) в программе. Например, оператор

```
x + 1;
```

является допустимым в C++ и недопустимым в Java, поскольку он не изменяет значения никаких переменных.

4.12. Операторы управления потоком. К операторам управления потоком относятся: условный оператор **if**, оператор выбора **switch**, операторы циклов **while**, **do---while** и **for**, операторы досрочного выхода из цикла **break** и перехода к следующей итерации **continue**, а также оператор выхода из функции **return**. Синтаксис этих операторов аналогичен C++ с небольшими модификациями.

Все управляющие операторы можно разделить на 3 категории:

Операторы выбора, которые позволяют программе выполняться тем или иным образом в зависимости от различных условий. Операторы цикла позволяют повторять выполнение одного или нескольких операторов. Операторы перехода позволяют пропускать выполнение одного или нескольких операторов и переходить сразу к нужному оператору.

Операторы выбора.

В языке Java существует всего 2 оператора выбора: оператор if и оператор switch. Они позволяют управлять выполнением той или иной программы в зависимости от некоторых условий.

Оператор if.

Оператор if позволяет направить выполнение программы по одному из 2-х возможных путей в зависимости от какого-либо условия. Конструкция с этим оператором имеет вид:

```
if условие() оператор_1;
else оператор_2;
```

Все операторы, выполняющие проверку условий, требуют, чтобы условие имело тип **boolean**. Это означает, что, в отличие от C++, конструкции if(n) и if(n != 0) не эквивалентны, и первая из них является синтаксически неверной, если переменная **n** имеет тип, отличный от **boolean**.

оператор выбора switch. Он позволяет направить поток программы по одному из возможных путей в зависимости от значения управляющего оператора. Использование данного оператора представляется более удобным по сравнению с оператором if в тех случаях, когда необходимо сделать выбор из более чем 2-х вариантов.

Оператор switch имеет следующий синтаксис:

```
switch управляющее(выражение_) {
  case значение1:оператор
   ;
   break;
```

```
  case значение2:оператор
   ;
   break;
  case значениеN:оператор
   ;
   break;
  default:оператор
   ; }
```

Управляющее выражение в операторе **switch** должно быть одного из целочисленных типов.

"Значение*"должно быть уникальным в пределах одного блока оператора "switch а его тип совместим с типом "Управляющего выражения". По ходу программы "Управляющее выражение"сравнивается со "значением*"в каждом блоке оператора "case". Если будет обнаружено совпадение, то выполниться последовательность "операторов находящаяся в данном блоке "case". Если совпадений не будет найдено, то выполниться оператор(ы), который находится в блоке "default". Использование блока "default"не обязательно. Если его не будет, то не найдя совпадений, программа не выполнит ни одного оператора. В каждом блоке "case"есть оператор "break который прерывает выполнение всего блока "switch"после обнаружения совпадения. Если этого оператора не будет, то программа продолжит выполнять операторы, которые находятся после блока, в котором найдено совпадение.

```
public class TimeYear {
    public static void main(String[] args) {
        int timeOfYear = 2;
        switch (timeOfYear) {
            case 1:
                System.out.println("Зима");
                break;
            case 2:
                System.out.println("Весна");
                break;
            case 3:
                System.out.println("Лето");
                break;
            case 4:
                System.out.println("Осень");
                break;
            default:
                System.out.println("Соответствия ненайдено!");
        }
    }
}
```

Циклы в Java представляют собой специальные конструкции, которые позволяют выполнять один или несколько операторов многократно до достижения условия завершения цикла. В Java существуют следующие операторы цикла: for, while, do-while.

Цикл while.

Оператор while наиболее часто используется для создания в программе циклов. Он позволяет выполнять какой-то блок операторов до тех пор, пока условие цикла является истинным. Как только условие становиться ложным, программа начинает работать с той строки кода, которая следует непосредственно за циклом.

Цикл while имеет следующую структуру:

```
while условие() {блокоператоров

 }
```

```
public class JavaApplication1 {
    public static void main(String[] args) {
        int d = 10;
        int c = 20;
        while (++d < --c);
        System.out.println("Среднее значениех2— переменных=" + d);
    }
}
```

Цикл do-while.

Мы знаем, что при использовании цикла while тело цикла может не выполниться ни раз, если условие цикла изначально ложно. Однако, иногда возникают ситуации, когда нам необходимо выполнить цикл хотя бы один раз, даже если условие изначально ложно. Для этих целей в языке Java предусмотрен цикл do-while. Этот цикл, сначала выполняет операторы цикла, а уже затем проверяет условие, и если оно ложно, то операторы больше не будут выполняться. Таким образом, в любом случае тело цикла будет выполнено хотя бы один раз.

Цикл do-while имеет следующий синтаксис:

```
do {блокоператоров
        ;
      } while условие();
```

Пример:

```
public class JavaApplication1 {

    public static void main(String[] args) {
        int d = 10;
        do {
            System.out.println("число " + d);
            ++d;
        } while (d < 10);
    }
}
```

Цикл for имеет 2 разновидности:

for – традиционная форма. for-each – новая форма.

В операторе **for**, так же как и в C++, возможно объявление переменных. Например

```
for(int i = 0; i < N; ++i)
```

Объявленные таким образом переменные действительны лишь внутри цикла. Можно использовать несколько выражений в заголовке оператора **for**, разделяя их запятыми:

```
for(i = 0, j = 0; i + j < 20; ++i, ++j)
```

Версия for-each имеет следующий синтаксис:

```
for (type iterVariable: коллекция) {блокоператоров
      ;
}
```

В этой конструкции "тип"– это тип данных; "итерационная переменная"– переменная, которая поочередно получает значения из "коллекции". "Коллекция"представляет собой набор из однотипных элементов, такой как массив. "Тип"и тип элементов в коллекции должен совпадать. В цикле for могут применяться различные виды коллекций. Мы будем использовать в качестве "коллекции"массивы, а в следующих уроках рассмотрим более сложные типы коллекций.

сумма всех элементов массива.

```
public class Summa {
    public static void main(String[] args) {
        int numb[] = {1, 2, 3, 4, 5};
        int summa = 0;
        for (int i = 0; i < 5; i++) {
            summa += numb[i];
        }
        System.out.println("Сумма=" + summa);
    }
}
```

Теперь посмотрим, как ту же задачу можно решить с помощью for-each.

```
public class JavaApplication1 {
    public static void main(String[] args) {
        int numb[] = {1, 2, 3, 4, 5};
        int summa = 0;
        for (int i : numb) {
            summa += i;
        }
        System.out.println("Сумма=" + summa);
    }
}
```

Основное отличие здесь в том, что мы не указываем начальное и конечное значение цикла. Здесь все делается автоматически. Цикл будет продолжаться до тех пор, пока не будут перебраны все элементы массива. Следует также сказать, что в цикле for-each нельзя изменить переменную "i т.к. она связана только с исходным массивом.

```
for(int i : numb){
    summa+=i;
    i=20;
}
```

Мы изменяем переменную "i присваивая ей значение "20". Но на программе это никак не отражается.

Операторы перехода

Java не содержит оператора **goto**. Вместо него существуют расширения операторов **break** и **continue**, позволяющие осуществить выход из нескольких вложенных циклов. Они мало распространены и потому здесь не рассматриваются. Их описание можно найти, например, в книге

Важно отметить, что наличие в программе недоступного кода (например, кода, следующего за оператором **return**) запрещено в Java и приводит к ошибке компиляции.

5. Классы и объекты

5.1. Основные понятия объектно-ориентированного программирования. Язык программирования Java поддерживает парадигму *объектно-ориентированного программирования* (ООП). Центральным понятием ООП является понятие класса. *Класс* — это тип данных, определяемый пользователем. Класс может содержать *поля* (переменные) и *методы* (функции). Переменные типа <<класс>> называются *объектами* (или *экземплярами класса*). Каждый объект имеет свой собственный набор экземпляров полей, определенных в классе. Содержимое полей в любой момент времени определяет *состояние* этого объекта. Методы существуют в одном экземпляре для каждого класса, однако вызов любого (нестатического) метода может осуществляться только с указанием объекта класса. Как правило, такой метод обрабатывает поля именно того экземпляра класса, на котором он вызывается (изменяет его состояние). Тем самым, методы класса определяют *поведение* всех объектов этого класса. Очень важно отметить, что *ни поля, ни методы класса не существуют сами по себе вне объектов этого класса.*

Понятие класса можно рассматривать как развитие понятия структуры языка C, при котором функции, обрабатывающие данные, размещённые в структурах, помещаются внутри этих структур. При этом средства объектно-ориентированных языков программирования позволяют запретить доступ к данным объектов извне и осуществлять обработку данных только посредством методов. В этом случае говорят, что *методы класса предоставляют интерфейс к его данным.*

Предыдущее изложение описывает понятия класса и объекта с технической стороны, с точки зрения средств языка программирования. Однако можно дать и другую интерпретацию. Характерной особенностью объектно-ориентированного программирования по сравнению с ранними парадигмами (структурное программирование) является то, что программист получает возможность не подбирать языковые средства для выражения механизмов обработки данных предметной области, а *моделировать* предметную область внутри программы. Объектам в программе в этом случае соответствуют (в большинстве случаев) объекты предметной области, а классам — универсальные понятия предметной области, характеризующие свойства и поведение однотипных объектов. Такой подход к программированию, как правило, является более эффективным (особенно при разработке больших программных комплексов, а также при коллективной разработке), поскольку программист получает возможность описывать взаимодействие объектов на языке предметной области, а не на языке, отражающем внутренние детали работы вычислительной машины.

5.2. Объявление класса. Для объявления класса используется следующий синтаксис:

```
[ спецификдоступа _ ] class имякласса _
{
  [ спецификдоступа _ ] типданных _ имяполя _ [ = начальнзнач _ ];
    . . .
  [ спецификдоступа _ ] типданных _ имяметода _список(аргументов _)
    {телометода
        _
    }
    . . .
}
```

Порядок объявления полей и методов в классе может быть произвольным. Возможные спецификаторы доступа описаны в п. 5.9. В отличие от C++ объявление класса и его реализация размещаются в одном и том же файле.

Пример

```
public class Point {
    private double x, y;
    public Point() {
        x = y = 0;
    }
    public Point(double abscissa, double ordinate) {
        x = abscissa;
        y = ordinate;
    }
    /*public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }*/
    public Point(double value) {
        x = y =value;
    }
    public double getX() {
        return x;
    }
    public double getY() {
        return y;
    }
    public void setX(double abscissa) {
        x = abscissa;
    }
    public void setY(double ordinate) {
        y = ordinate;
    }
    public void printPoint() {
        System.out.println("("+x+","+y+")");
    }
    public double distance(Point p) {
        return Math.sqrt((x-p.x)*(x-p.x)+(y-p.y)*(y-p.y));
    }
}
```

5.2.1. Конструктор

При создании объекта (в момент его определения и только в этот момент) обязательно вызывается специальный метод инициализирующий объект. Это конструктор. Компилятор может его найти среди остальных методов, т.к. его имя совпадает с именем класса. Конструктор не имеет возвращаемого значения, но в отличие от остальных функции слово `void` перед ним не пишется. Конструктор может иметь аргументы, можно определять несколько конструкторов, отличающихся друг от друга набором и количеством аргументов. Это называется перегрузкой конструктора.

Если конструктор не имеет параметров, то его называют конструктором по умолчанию. Если конструктор класса не объявлен явно, то создаётся пустой конструктор, не имеющий аргументов (конструктор по умолчанию).

Аргументы по умолчанию не поддерживаются. Если такая возможность требуется, то следует явно реализовать методы со всеми необходимыми наборами аргументов.

Часто чтобы избежать дублирования кода конструктор вызывают из конструктора, используя слово this (в этом случае нельзя вызвать два конструктора подряд)

```
    public Point(double value) {
        this(value, value);
    }
```

Если в классе определено два метода с одним и тем же именем, то такие методы называются *перегруженными*. Выбор необходимого перегруженного метода осуществляется компилятором на основании количества и типов аргументов, передаваемых методу. Не допускается определение двух методов с одинаковыми сигнатурами (сигнатурой называется совокупность имени метода и типов его аргументов), так как в этом случае компилятор не сможет определить, какой из методов следует вызывать. Нельзя создавать методы отличающиеся только типом возвращаемого значения.

5.3. Статические поля и методы. Статические поля и методы существуют в единственном экземпляре для каждого класса, т. е. являются разделяемыми для всех экземпляров этого класса.

Для объявления статических полей и методов используется спецификатор **static**:

```
static int static_field = 10;
```

Статические поля инициализируются в момент загрузки класса либо значением, указанным в объявлении (см. пример п. **??**), либо в специальном **static**-блоке.

Статические методы не могут (без явного указания объекта) обращаться к нестатическим полям и нестатическим методам класса.

Вызов статических полей и методов может осуществляться без указания экземпляра класса. При таком обращении вместо имени экземпляра используется имя самого класса. Например:

```
double x = Math.sqrt(2.0);
```

Метод **main()**, который является точкой входа в Java-программу, всегда должен объявляться статическим.

Определим в классе **Point** статический метод:

```
    public static double distance(Point a, Point b) {
        return Math.sqrt((a.x−b.x)*(a.x−b.x)+(a.y−b.y)*(a.y−b.y));
    }
```

Рассмотрим примеры его использования:

```
public class PointApp {

    public static void main(String[] args) {
        // TODO code application logic here
        Point a = new Point();
        Point b = new Point(0,5);
        System.out.println("("+a.getX()+","+a.getY()+")");
        b.printPoint();
        System.out.println(a.distance(a,b));
```

```
        double d = Point.distance(a, b);
        System.out.println(d);
    }
}
```

5.4. Создание объектов. В Java все объекты создаются в динамической памяти виртуальной машины. Переменных-объектов в том смысле, в каком они есть в C++, в Java не существует, а переменные, типом которых (по синтаксису) является некоторый класс, на самом деле являются не объектами, а *ссылками* на эти объекты (объектными ссылками). Например

```
Point myPoint;
```

Здесь объявляется объектная ссылка типа **Point** с именем **myPoint** и *не создаётся* сам объект (в C++ аналогом данного объявления было бы объявление указателя: Point* myPoint). Собственно создание объекта осуществляется с помощью операции **new**:

```
myPoint = new Point();
```

Здесь создаётся объект типа **Point** (при создании вызывается конструктор без аргументов), и ссылка на него записывается в переменную **myPoint**. Отметим, что в Java нет никакой возможности работать с объектами непосредственно как в C++ Возможно одновременное объявление объектной ссылки и её инициализация:

```
Point myPoint = new Point();
```

```
public class PointApp {

    public static void main(String[] args) {
        // TODO code application logic here
        Point a = new Point();
        Point b = new Point(0,5);
        System.out.println("("+a.getX()+","+a.getY()+")");
        b.printPoint();
        System.out.println(a.distance(b));
    }
}
```

5.5. Обращение к полям и методам. Когда объект создан, доступ к его полям и методам осуществляется посредством операции . (точка):

```
myPoint.someMethod();
```

При обращении к полю или методу объекта из метода, вызванного на этом же объекте, имя объекта может опускаться либо заменяться ключевым словом **this**. Ключевое слово **this** используется также для вызова конструктора класса из другого конструктора (см. пример выше). В последнем случае подобный вызов должен быть первым оператором конструктора.

5.6. Удаление объектов. Операция удаления объектов в Java не предусмотрена. Её заменяет механизм автоматической сборки мусора. Для обеспечения работы этого механизма JVM (виртуальная машина Java) отслеживает количество объектных ссылок на каждый созданный объект и, как только на объект не остаётся ни одной ссылки, он становится кандидатом на удаление. Само удаление объектов осуществляется периодически и только в случае, когда объём памяти, требующей освобождения, становится существенным.

Автоматическая сборка мусора позволяет обходить значительное количество проблем, связанных с утечками памяти. К сожалению, в некоторых случаях такие проблемы всё же возникают и в Java-приложениях.

Деструкторы в Java не предусмотрены. Так как освобождение памяти осуществляет JVM, то необходимость в них возникает не слишком часто. Если они всё же необходимы, следует определять собственные методы и вызывать их явно. Примером здесь может служить метод **close()** поточных классов Java, осуществляющий закрытие потока ввода/вывода.

5.7. **Особенности использования объектных ссылок**. Объектная ссылка ведёт себя как указатель в C, поэтому присвоение одной объектной ссылки другой *не приводит* к копированию объекта. Для примитивных типов присвоение приводит к копированию соответствующего значения в переменную (содержимое a копируется в b).

```
int a, b;
b = a;
```

При работе с ссылочными типами аналогичное присвоение приводит к копированию ссылок на объект, а не самого объекта.

```
Point a, b;
a = new Point();
b = new Point();
b = a;
```

Теперь ссылки a и b указывают на одим и тот же объект.

Аналогично передача примитивных типов методам осуществляется *по значению*, а при передаче ссылочных типов по ссылке. Например результат выполнения программы

```
public class PointApp {
    public static void f (Point p) {
        p.setX(10);
        p.setY(−10);
    }
    public static void main(String[] args) {
        // TODO code application logic here
        Point a = new Point();
        a.printPoint();
        f(a);
        a.printPoint();
    }
}
```

будет следующим:

```
(0.0,0.0)
(10.0,−10.0)
```

Рассмотрим типичную ошибку при использовании объектных ссылок. Пусть требуется реализовать метод, выполняющий обмен двух объектов класса **Point**. Поскольку объекты передаются по ссылке, естественной может показаться следующая реализация

```
static void swap(Point c1, Point c2)
{
    // Даннаяреализацияневерна!
    Point s = c1;
    c1 = c2;
    c2 = s;
}
```

Однако данная реализация является неверной, поскольку она производит обмен *копий ссылок* на объекты, подлежащие обмену. Поскольку копии являются локальными переменными, то они уничтожаются при выходе из метода.

Верная реализация (реализованная в виде метода) должна обменивать *содержимое* объектов, а не ссылки на них:

```
static void swap(Point c1, Point c2)
{
    double s;
    s = c1.x; c1.x = c2.x; c2.x = s;
    s = c1.y; c1.y = c2.y; c2.y = s;
}
```

Наконец, отметим, что с точки зрения объектно-ориентированного программирования предпочтительным является решение данной задачи, оформленное в виде метода *самого класса* **Point**. Пример такой реализации:

```
void swap(Point c)
{
    double s;
    s = x; x = c.x; c.x = s;
    s = y; y = c.y; c.y = s;
}
```

Здесь происходит обмен содержимого объекта, на котором вызван метод **swap()** с другим объектом, переданным этому методу в качестве аргумента. Отметим, что такая реализация не провоцирует программиста использовать описанный выше неверный подход.

Логическая операция сравнения тоже работает со ссылками:

```
Point a = new Point();
        Point b = new Point();
        if (a == b)
            System.out.println("Equals");
        else
            System.out.println("No equals");
```

результат:

```
No equals
```

Можно переопределить специальный метод.

```
    public boolean equals (Point p) {
        if (x!=p.x)
            return false;
        if (y!=p.y)
            return false;
        return true;
    }
```

и вызвать его

```
if (a.equals(b) == true)
            System.out.println("Equals");
        else
            System.out.println("No equals");
```

5.8. Пакеты и структура модуля трансляции. Все идентификаторы, которые мы до сих пор использовали в наших примерах, располагались в одном и том же пространстве имен (name space). Это означает, что нам во избежание конфликтных ситуаций приходилось заботиться о том, чтобы у каждого класса было свое уникальное имя. Пакет (package) — это некий контейнер, который используется для того, чтобы изолировать имена классов. Например, вы можете создать класс List, заключить его в пакет и не думать после этого о возможных конфликтах, которые могли бы возникнуть если бы кто-нибудь еще создал класс с именем List.

Пакеты являются средством группировки типов (классов и интерфейсов) в Java-программе. Каждый пакет может содержать один или несколько классов и интерфейсов, а также других пакетов. Распределение классов по пакетам позволяет решать проблему конфликтов имён, а также более эффективно осуществлять управление доступом.

Название пакета должно соответствовать названию веб-сайта организации.

Для обращения к элементам пакета используется полный путь, состоящий из набора подпакетов, разделённых точками (например, **java.util.HashSet**). Для обращения к классам того же самого пакета можно использовать просто их имена, опуская имя пакета.

Модулем трансляции в Java является файл. У каждого файла .java есть 4 одинаковых внутренних части, из которых мы до сих пор в наших примерах использовали только одну. Ниже приведена общая форма исходного файла Java.

одиночный оператор package (необязателен) любое количество операторов import (необязательны) одиночное объявление открытого (public) класса любое количество закрытых (private) классов пакета (необязательны)

Каждый модуль трансляции может начинаться утверждением **package**, определяющим пакет, которому принадлежат классы, описанные в данном модуле. Например,

```
package mypackage.mysubpackage;
```

Если **package**-утверждение отсутствует, то считается, что класс принадлежит пакету по умолчанию, который не имеет имени, не может иметь подпакетов и не является видимым из других пакетов.

Оператор import

После оператора package, но до любого определения классов в исходном Java-файле, может присутствовать список операторов import. Пакеты являются хорошим механизмом для отделения классов друг от друга, поэтому все встроенные в Java классы хранятся в пакетах. Общая форма оператора import такова:

import пакет1 [.пакет2].(имякласса|*);

Здесь пакет1 — имя пакета верхнего уровня, пакет2 — это необязательное имя пакета, вложенного в первый пакет и отделенное точкой. И, наконец, после указания пути в иерархии пакетов, указывается либо имя класса, либо метасимвол звездочка. Звездочка означает, что, если Java-транслятору потребуется какой-либо класс, для которого пакет не указан явно, он должен просмотреть все содержимое пакета со звездочкой вместо имени класса. В приведенном ниже фрагменте кода показаны обе формы использования оператора import :

Все встроенные в Java классы, которые входят в комплект поставки, хранятся в пакете с именем java. Базовые функции языка хранятся во вложенном пакете java.lang. Весь этот пакет автоматически импортируется транслятором во все программы. Это эквивалентно размещению в начале каждой программы оператора

import java.lang.*;

Модуль трансляции может содержать одно или несколько **import**-утверждений, позволяющих опускать имена пакетов при обращении к классам. Например, если в модуле трансляции присутствует строка

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

то ко всем классам пакета **java.util** (но не к классам его подпакетов!) можно будет обращаться по коротким именам (**HashSet** вместо **java.util.HashSet** и т. д.). Возможно и импортирование единичного класса пакета, например:

```
import java.util.HashSet;
```

Но использовать без нужды форму записи оператора import с использованием звездочки не рекомендуется, т.к. это может значительно увеличить время трансляции кода (на скорость работы и размер программы это не влияет).

Модуль трансляции обязательно должен содержать исходный текст основного класса (или интерфейса) модуля. Имя этого класса должно совпадать с именем файла модуля трансляции. Этот класс является единственным классом модуля трансляции, который может быть объявлен открытым (**public**, см. п. 5.9). Модуль трансляции может содержать также один или несколько вспомогательных классов.

Файлы исходных текстов, а также файлы оттранслированных классов должны располагаться в файловой системе в соответствии с иерархией пакетов, которым они принадлежат. Например, исходный текст класса **java.util.HashSet** должен размещаться в файле java.util.HashSet подкаталога util каталога java.

Поиск файлов, содержащих байт-код классов, осуществляется в подкаталогах, соответствующих местоположению классов в иерархии пакетов, внутри каталогов, перечисленных в переменной окружения **CLASSPATH**.

Например, для того чтобы JVM нашла байт-код класса **java.util.HashSet**, он должен находиться в файле с именем HashSet.class в подкаталоге java/util какого-либо каталога переменной окружения **CLASSPATH**. Как правило, туда включается текущий каталог, а также каталог, содержащий стандартные библиотеки Java. Файлы исходных текстов обычно также располагаются в файловой системе в соответствии с иерархией пакетов.

Например, если эта переменная имеет значение <</opt/jdk/lib:.>>, JVM будет искать байт-код класса **java.util.HashSet** в файле `/opt/jdk/lib/java/util/HashSet.class`, а затем в файле HashSet.class подкаталога java/util текущего каталога.

5.9. Управление доступом и инкапсуляция. Управление доступом используется для того, чтобы, с одной стороны, предотвратить несанкционированное изменение данных, содержащихся в объектах, в результате неправильного или злонамеренного использования, а с другой стороны, чтобы отделить интерфейс класса от его реализации (сокрытие реализации). Программист, использующий класс, видит только его интерфейс (поля и методы с открытым доступом) и может ничего не знать о внутреннем устройстве класса. Последнее позволяет изменять реализацию класса прозрачно для программиста. Например, можно заменить реализацию более эффективной, полностью перестроив внутреннее устройство класса, причём программный код, использующий класс, будет без изменений работать с новой версией класса.

Размещение полей и методов внутри классов совместно с сокрытием реализации называется *инкапсуляцией*. Инкапсуляция является характерной чертой объектно-ориентированного программирования.

Управление доступом осуществляется путём установки определённого спецификатора доступа в описании классов, полей и методов. В Java существует два спецификатора доступа для классов и четыре — для полей и методов. Все они вместе с областями видимости соответствующих элементов программы перечислены в табл. 5.1, 5.2. Уровни доступа обозначаются своими спецификаторами, кроме уровня доступа <<по умолчанию>>, которому никакого специального спецификатора не соответствует.

Область видимости		private	по умолч.	protected	public
тот же пакет	тот же класс	+	+	+	+
	другой класс	--	+	+	+
другой пакет	субкласс	--	--	+	+
	не субкласс	--	--	--	+

Таблица 5.1. Уровни доступа полей и методов

Область видимости	по умолч.	public
тот же пакет	+	+
другой пакет	--	+

Таблица 5.2. Уровни доступа классов и интерфейсов

Приведём некоторые рекомендации по использованию управления доступом в Java-программах.

есть несколько правил, которые помогут вам разобраться. Элемент, объявленный public, доступен из любого места. Все, что объявлено private, доступно только внутри класса, и нигде больше. Если у элемента вообще не указан модификатор уровня доступа, то такой элемент будет виден из подклассов и классов того же пакета. Именно такой уровень доступа используется в языке Java по умолчанию. Если же вы хотите, чтобы элемент был доступен извне пакета, но только подклассам того класса, которому он принадлежит, вам нужно объявить такой элемент protected. И наконец, если вы хотите, чтобы элемент был доступен только подклассам, причем независимо от того, находятся ли они в данном пакете или нет — используйте комбинацию private protected.

Главное правило управления доступом: *при проектировании для всех элементов программы следует задавать максимально ограниченный уровень доступа, позволяющий этим элементам правильно функционировать.* Чем меньше классы знают друг о друге, тем больше шансов они имеют быть использованными повторно.

Для полей классов используется доступ **private**. Для чтения и изменения их значений извне используются специальные методы (setter'ы и getter'ы), имеющие спецификатор доступа **public**. Например,

```
class MyClass
{
    private int value;
    public void setValue(int value) { this.value = value; }
    public int getValue() { return value; }
}
```

Данный подход является достаточно гибким. Например, всегда можно сделать некое поле полем только для чтения, удалив или просто не реализовав соответствующий setter, или ограничить возможные значения для поля, вставив соответствующую проверку.

Методы класса, имеющие спецификатор доступа **public**, образуют *интерфейс к данным* этого класса. Только через эти методы внешние классы смогут обращаться к данным, инкапсулированным классом, и только они определяют возможные способы обработки этих данных.

При наличии наследования (см. п. 6.1) спецификаторы доступа элементов **private** и <<по умолчанию>> часто заменяются спецификатором **protected**, чтобы позволить наследникам класса иметь доступ к соответствующим элементам классов. Также возможно применение доступа по умолчанию, если вся иерархия наследников находится в том же пакете, что и наследуемый класс.

Доступ по умолчанию может использоваться для создания дружественных классов, которые могут обращаться к закрытым полям и методам друг друга. Такие дружественные классы размещаются в одном пакете. Возможности управления доступом в этом случае гибче, чем в случае дружественных классов в C++. Однако злоупотреблять этой возможностью не рекомендуется, поскольку дружественные классы оказываются сильнее связанными друг с другом, чем обычные, поэтому изменения в одном классе могут потребовать серьёзных изменений в других.

```
    public void clone(Point p) {
        x=p.x;
        y=p.y;
    }
```

```
package pointapp;
public class Rectangle {
    private Point a, b;
    public Rectangle () {
        a = new Point(0,1);
        b = new Point(1,0);
    }
    public Rectangle (double x1, double y1, double x2, double y2) {
        a = new Point(x1,y1);
        b = new Point(x2,y2);
    }
    public Rectangle (Point v1, Point v2) {
        a = new Point();
        a.clone(v1);
        b = new Point();
        b.clone(v2);
    }
    public double square () {
        return Math.abs(a.getX() − b.getX()) * Math.abs(a.getY() − b.getY());
    }
}
```

```
Point a = new Point();
Point b = new Point(1,1);
Rectangle r = new Rectangle(−1, 1, 3, 4);
Rectangle r2 = new Rectangle(a,b);
System.out.println(r2.square());
```

6. Наследование и полиморфизм

6.1. Наследование. *Наследование* — это механизм построения новых классов на основе уже существующих. При этом наследники класса (*субклассы*) получают свойства и функциональность класса-родителя (*суперкласса*) и имеют возможность изменять и расширять их. Механизм наследования используется для создания более частных, специализированных классов по сравнению с суперклассом.

Для того чтобы объявить класс наследником некоторого другого класса, используется ключевое слово **extends** с последующим именем наследуемого класса, добавляемое в объявление класса:

[специфдоступа_] **class** имясубкласса_ **extends** имясуперкласса_

В Java допускается наследование только от одного класса. Субкласс наследует все поля и методы суперкласса, однако те из них, которые объявлены со спецификаторами доступа **private** и по умолчанию (последние, только если субкласс находится вне пакета, содержащего суперкласс), оказываются недоступными для субкласса.

```
 class Base {
     private int a;
     protected double b;
     private double product() {
         return a*b;
     }
     public Base() {
         a=0;
         b=1;
     }
     public int getA() {
         return a;
     }
     public void setA(int a) {
         this.a = a;
     }
     public void print() {
         System.out.println(product());
     }
}
class Son extends Base {
     private char c;
     public Son() {
         //a=2;
         b=2;
         c='a';
     }
     public double sum() {
     //return a+b;
     return getA()+b;
     }
```

```
}
public class SonApp {
    public static void main(String[] args) {
        Base x = new Base();
        x.print();
        //double p = x.product();
        Son y = new Son();
        y.print();
    }
}
```

Можно сказать, что в объекте класса-наследника содержится (упакован) объект базового класса.

6.2. Наследование методов. Внутри класса-наследника можно обращаться ко всем элементам базового класса, кроме части private. Объект класса-наследника может вызвать методы только из части public, как своей так и предка.

Конструкторы не наследуются. При создании объекта сначала автоматически вызывается конструктор базового класса, затем свой, если уровней наследования несколько, конструкторы всех предков вызываются попорядку, начиная со старшего. Можно описать явный вызов конструктора класса-предка внутри конструктора класса-наследника (для этого первым оператором конструктора должен быть вызов конструктора суперкласса через **super**), неявно происходит вызов конструктора по умолчанию. Вызов конструкторов родительских классов происходит сверху вниз по дереву наследования.

Если в базовом классе есть только конструктор с параметрами, то в классе-наследнике его придется вызывать явно.

```
class Base {
    private int a;
    public Base(int a) {
        this.a=a;
    }
}
class Son extends Base {
    public Son() {
        super(0);
    }
}
```

Если мы перегружаем метод, то есть создаем метод, у которого имя совпадает с именем метода базового класса, но аргументы различные, то все варианты метода можно использовать в наследнике (в отличие от C++).

```
class Base {
    private int a;
    public Base() {
        a=0;
    }
    public void method() {
        System.out.println("base");
    }
    public void method(int b) {
        System.out.println(b);
    }
```

```
}
class Son extends Base {
    public void method(double c) {
        System.out.println("son "+c);
    }
}
public class SonApp {
    public static void main(String[] args) {
        Base x = new Base();
        x.method();
        x.method(5);
        Son y = new Son();
        y.method();
        y.method(10);
        y.method(0.5);
    }
}
run:
base
5
base
10
son 0.5
```

Субкласс может переопределять поля и методы суперкласса, если сигнатура метода полностью совпадает с сигнатурой метода базового класса (для поля - совпадение имен). В этом случае доступ к родительским полям и методам закрывается и может осуществляться только из методов субкласса посредством ключевого слова **super**, синтаксис которого совпадает с синтаксисом явного обращения к элементам текущего экземпляра через **this**.

```
class Base {
    private int a;
    public Base() {
        a=0;
    }
    public void method(int b) {
        System.out.println(b);
    }
}
class Son extends Base {
    public void method(int c) {
        System.out.println("son "+c);
    }
    public void example () {
        System.out.print("example ");
        super.method(20);
    }
}
public class SonApp {
    public static void main(String[] args) {
        Son y = new Son();
        y.method(10);
        y.example();
```

```
        }
    }
    run:
    son 10
    example 20
```

Если опустить в методе example () слово super

```
class Base {
    private int a;
    public Base() {
        a=0;
    }
    public void method(int b) {
        System.out.println(b);
    }
}
class Son extends Base {
    public void method(int c) {
        System.out.println("son "+c);
    }
    public void example () {
        System.out.print("example ");
        method(20);
    }
}
public class SonApp {
    public static void main(String[] args) {
        Son y = new Son();
        y.method(10);
        y.example();
    }
}
run:
son 10
example son 20
```

В Java SE5 появилось слово @Override, которое пишут перед переопределяемым методом. Если по ошибке метод будет перегруженным, компилятор выдаст ошибку.

```
    @Override
    public void method(int c) {
        System.out.println("son "+c);
    }
```

При переопределении полей и методов возможно изменение спецификатора доступа на более широкий (**private** на любой другой; по умолчанию на любой другой, кроме **private** и т. д.).

Если в описании класса суперкласс для него не определён, то по умолчанию производится наследование от библиотечного класса **Object**. Этот класс представляет собой корень иерархии наследования в Java и содержит методы, необходимые для корректного функционирования JVM, а также методы, выполняющие такие распространённые операции, как преобразование к строке, сравнение объектов на равенство и т. д.

Стандартная операция Java == осуществляет *сравнение ссылок на объекты*. Однако, как правило, требуется сравнение объектов не по ссылкам, а *по их содержимому*. Для этого предназначен метод

```
boolean equals(Object obj)
```

определённый в классе **Object** и потому содержащийся во всех классах Java. Реализация этого метода по умолчанию функционирует так же, как и операция ==, т. е. сравнивает объектные ссылки. Программист должен переопределить этот метод так, чтобы он осуществлял сравнение требуемым образом. Например, для класса **Point** (<<точка плоскости>>), содержащего две целочисленные координаты **x** и **y**, соответствующий метод можно реализовать следующим образом:

```
public boolean equals(Object obj)
{
    if(!(obj instanceof Point))
        return false;
    Point c = (Point)obj;
    return x == c.x && y == c.y;
}
```

Если в программе используются стандартные классы-контейнеры (см. п. 10.1), обычно требуется переопределить не только метод **equals()**, то также метод

```
int hashCode()
```

который возвращает хэш-код объекта. Хэш-код объекта — это целое число, которое вычисляется на основании состояния объекта (содержимого его полей). Равным объектам (т. е. объектам, для которых метод **equals()** возвращает значение **true**) должен соответствовать один и тот же хэш-код. Для неравных объектов хэш-коды не обязаны быть неравными, однако к этому следует стремиться. Кроме того, следует учитывать, что чем более равномерным является распределение хэш-кодов объектов, тем выше будет производительность встроенных коллекций, их использующих (множеств и ассоциативных массивов).

Для рассмотренного выше класса **Point** можно определить метод **hashCode()** следующим образом:

```
int hashCode()
{
    return x + y;
}
```

6.3. Ограничение и форсирование наследования. Если метод объявлен со спецификатором **final**, то его переопределение запрещается. Если же со спецификатором **final** объявлен класс, то запрещается наследование от такого класса.

Отдельные методы класса могут быть объявлены абстрактными. В этом случае они объявляются со спецификатором **abstract** и не имеют тела:

```
abstract void abstractMethod();
```

Такие методы должны обязательно переопределяться в субклассах. Если класс содержит хотя бы один абстрактный метод, то он также должен быть объявлен абстрактным. В этом случае запрещается создание объектов данного класса. Чтобы иметь возможность создавать экземпляры субклассов такого класса, следует реализовать в них все абстрактные методы суперкласса. Если субкласс реализует не все абстрактные методы суперкласса, он также должен объявляться абстрактным.

6.4. Полиморфизм. *Полиморфизм* --- это механизм, который позволяет обращаться к объектам, унаследованным от одного класса, как к объектам этого класса. При таком обращении к объектам субклассов они реагируют способом, определённым в соответствующем субклассе, а не в родительском классе. В этом случае методы, определённые в суперклассе, образуют *единый интерфейс доступа к данным всех субклассов*. Полиморфизм является одним из самых мощных средств объектно-ориентированного программирования, поскольку он позволяет единообразно обрабатывать наборы объектов, принадлежащих различным классам, причём выбор реализации той или иной операции осуществляется автоматически на основании типа данных объекта.

Реализация полиморфизма в Java основана на следующих правилах.

1. Ссылка на объект класса может быть приведена к ссылке на объект любого из классов, находящихся выше данного в иерархии наследования. Это преобразование является расширяющим и может производиться автоматически. Обратное преобразование также может быть произведено, но оно выполняется всегда явно. При попытке привести объектную ссылку к недопустимому типу возникает ошибка. Для проверки допустимости преобразования может быть использована операция **instanceof**:

```
if(myPoint instanceof MyClass)
    ((MyClass)myPoint).doSomethingSpecificForMyClass();
```

2. Если некоторые из методов суперкласса были переопределены в субклассе, то *при обращении к этим методам даже посредством ссылки на объект суперкласса будет происходить вызов именно переопределённых методов*. Это отличается от реализации полиморфизма в C++, где обращение к переопределённым методам в описанной ситуации будет происходить лишь в том случае, когда соответствующие методы были объявлены виртуальными.

Рассмотрим пример. Пусть есть базовый класс <<геометрическая фигура>>. Он содержит метод **area()**, предназначенный для вычисления площади фигуры. Каждый из субклассов класса <<фигура>> переопределяет этот метод таким образом, чтобы вычислялась площадь соответствующей фигуры.

Ссылки на создаваемые экземпляры классов конкретных фигур сохраняются в массиве ссылок, имеющих тип <<фигура>>, а затем единообразно обрабатываются: для каждой фигуры выводится её площадь. При этом никаких проверок типов фигур не требуется, для каждой фигуры автоматически вызывается нужный метод.

```
abstract class Figure
{
      abstract public double area();
}
class Rectangle extends Figure
{
    private double a, b;
    public Rectangle(double a, double b) { this.a = a; this.b = b; }
    public double area() { return a * b; }
}
class Circle extends Figure
{
    private double r;
    public Circle(double r) { this.r = r; }
    public double area() { return Math.PI * r * r; }
}
class Triangle extends Figure
```

```
{
    private double a, b, c;
    public Triangle(double a, double b, double c)
    {
        this.a = a; this.b = b; this.c = c;
    }
    public double area()
    {
        double p = (a + b + c) / 2;
        return Math.sqrt(p * (p − a) * (p − b) * (p − c));
    }
}
public class FiguresPolymorphism
{
    public static void main(String args[])
    {
        Figure f[] = new Figure[3]; // массивобъектныхссылок
        f[0] = new Rectangle(2, 3);
        f[1] = new Triangle(3, 4, 5);
        f[2] = new Circle(1);
        for(int i = 0; i < f.length; ++i)
            System.out.println(f[i].area());
    }
}
```

Сопоставление вызова метода с телом метода называется связыванием. Если связывание осуществляется на этапе компиляции оно называется ранним или статическим связыванием. Если связывание осуществляется в момент выполнения программы, то поздним или динамическим.

В Java все методы связываются динамически, кроме закрытых.

6.5. Интерфейсы. В Java не допускается наследование одного класса от нескольких. Однако существует ограниченный аналог множественного наследования в виде *множественной реализации одним классом нескольких интерфейсов*. Функционально интерфейсы близки к абстрактным классам с определёнными ограничениями (допускаются только абстрактные методы и только статические поля), однако любой класс может *реализовывать* (аналог наследования) несколько интерфейсов.

Для объявления интерфейса используется следующий синтаксис:

```
interface имяинтерфейса_ [ extends списокимёнинтерфейсовпредков___ ]
```

Все методы интерфейса неявно получают спецификатор **abstract public**, а все переменные интерфейса --- спецификатор **final static public** и должны быть сразу же инициализированы. Использование других спецификаторов доступа в интерфейсах не допускается.

Любой класс может реализовать любое количество интерфейсов. Для объявления класса, реализующего один или несколько интерфейсов, используется следующий синтаксис:

```
[ специфдоступа_ ] class имякласса_ [ extends имясуперкласса_ ]
    implements списокименинтерфейсов__
```

Элементы списка имён интерфейсов разделяются запятыми. Если класс объявлен как реализующий некоторые интерфейсы, он должен либо реализовать все методы, объявленные в этих интерфейсах, либо быть абстрактным.

Подобно возможности приведения ссылки на объект класса к ссылке на объект его суперкласса, существует возможность приведения такой ссылки к ссылке на любой из интерфейсов, которые реализует класс. Такие ссылки ведут себя абсолютно аналогично ссылкам на объекты абстрактных классов. Например:

```
package interfeiceapp;
public interface MyArray {
    int Get(int i);//возвращает элементпоиндексу
    int Add(int value);//кладет элементвконецмассивавозвращаетегоиндекс
    int Size();//возвращает размермассива
}
public class Array implements MyArray {
    int[] array = new int[100];
    int size = 0;//количество использованныхэлементов
    @Override
    public int Get(int i) {
        return array[i];
    }
    @Override
    public int Add(int value) {
       array[size] = value;
        size++;
        if(size==array.length){//если массивзакончился
            int[] temparray = new int[size * 2];
            for(int i = 0; i < size;i++){
                temparray[i] = array[i];
            }
            array = temparray;
        }
        return size−1;
    }
    @Override
    public int Size() {
        return size;
    }
}
public class LinkedList implements MyArray {
    private class Node {
        public int value;
        public Node next;
    }
    int size = 0;
    Node firstNode = new Node();
    Node lastNode = firstNode;
    @Override
    public int Get(int i) {
        Node tempNode = firstNode;
        for (int j = 0; j < i; j++) {
            tempNode = tempNode.next;
        }
        return tempNode.value;
    }
```

```
    @Override
    public int Add(int value) {
        lastNode.value = value;
        lastNode.next = new Node();
        lastNode = lastNode.next;
        size++;
        return size − 1;
    }
    @Override
    public int Size() {
        return size;
    }
}
```

вместо конкретных классов, мы в программе будем указывать лишь интерфейсы и с легкостью менять реализации:

```
public class InterfeiceApp {
    public static void main(String[] args) {
        MyArray superArray = new Array();
        superArray = new LinkedList();superArray = new Array();
    }
}
public class InterfeiceApp {
    public void method(MyArray A) {...}
    public static void main(String[] args) {
        MyArray superArray = new Array();
        superArray = new LinkedList();
        method(superArray);
        superArray = new Array();
        method(superArray);
    }
}
```

6.6. Рекомендации по использованию наследования и полиморфизма. При разработке иерархии классов следует строить дерево наследования в соответствии с принципом: *субкласс является более частным, специализированным классом по отношению к суперклассу*. Несоблюдение этого принципа приводит к серьёзным ошибкам проектирования, проявляющимся в сильной связности классов, неочевидности взаимоотношений между ними и затруднении проектирования на последующих этапах.

Несмотря на очевидное сходство абстрактных классов и интерфейсов, их назначение, вообще говоря, является несколько различным. Абстрактный класс представляет собой абстрактную сущность, сущность, детали которой не до конца описаны в силу её общности. В отличие от класса интерфейс --- это не сущность, а, скорее, набор потенциальных возможностей, которые могут быть реализованы различными классами, не имеющими между собой ничего общего кроме этих возможностей.

Наследование поддерживает полиморфизм <<генетически>> общих методов. Полиморфизм же методов, имеющий другую природу, должен быть реализован посредством интерфейсов. Примером может служить интерфейс **Comparable<T>**. Этот интерфейс может быть реализован любым классом, допускающим упорядочение своих экземпляров. В то же время предположение о том, что все объекты, допускающие сравнение, должны быть субклассами одного класса выглядит несостоятельным.

7. Обработка исключений

7.1. Концепция исключений. Исключительной называется ситуация, из-за которой нормальное продолжение работы метода или даже всей программы невозможно. При этом обработка ее как обычной ошибки невозможна (например, исключительная ситуация может возникнуть в конструкторе, если невозможно правильно создать объект). В таком случае генерируются специальные объекты-исключения.

Исключения представляют собой объектно-ориентированный механизм обработки исключительных ситуаций при выполнении программы.

Исключение --- объект, который создается (оператором new) и *выбрасывается* программой при возникновении исключительной ситуации. Это может происходить автоматически. Исключения могут возбуждаться и «вручную» для того, чтобы сообщить о некоторых нештатных ситуациях. Обычно такой объект инкапсулирует описание этой ситуации. Выброшенное исключение *ловит* некоторый обработчик --- программный блок, осуществляющий обработку исключительной ситуации. После выполнения обработчика управление передаётся на оператор, непосредственно следующий за обработчиком. Таким образом осуществляется переход из одной части программы в другую с выполнением некоторых специальных действий, заключённых в обработчике исключительной ситуации.

В идеале исключения позволяют разобраться с проблемой и восстановить работоспособность программы, в крайнем случае остановить выполнение и сообщить о возникших трудностях.

Объекты-исключения автоматически создаются исполняющей средой Java в результате возникновения определенных исключительных состояний. Например, программа содержит выражение, при вычислении которого возникает деление на нуль.

```
public class JavaApplication1 {
    public static void main(String[] args) {
        int d = 0;
        int a = 42 / d;
    }
}
run:
Exception in thread "main" java.lang.ArithmeticException: / by zero
at javaapplication1.JavaApplication1.main(JavaApplication1.java:15)
...
```

7.2. Выбрасывание и обработка исключений. Все объекты-исключения в Java должны принадлежать классам, унаследованным от класса **Throwable**, имеющего два субкласса: **Exception** и **Error**. Исключения типа **Error** выбрасываются JVM в случае возникновения серьёзных ошибок. Как правило, программист не должен ни выбрасывать, ни обрабатывать такие исключения. Все остальные исключения являются экземплярами класса **Exception** и его субклассов. Кроме того, имеется особая группа исключений, представленная классом **RuntimeException** и его субклассами.

Для того чтобы выбросить исключение, используется следующий синтаксис:

```
throw объектисключение−;
```

Обычно объект-исключение создаётся непосредственно в операторе **throw**, например:

```
throw new NullPointerException();
```

Для задания блока программного кода, который требуется защитить от исключений, используется ключевое слово try. Сразу же после try-блока помещаются блоки catch, задающие тип исключения которое вы хотите обрабатывать. Оператор, выбрасывающий исключение, должен находиться внутри специального блока **try**, непосредственно за которым располагаются обработчики исключений:

```
try
{операторывыбрасывающиеисключения
    ——
}
catch( Type1 id1)
{телообработчикаисключения
    ——
}
catch( Type2 id2)
{телообработчикаисключения
    ——
}
. . .
[ finally
  {операторы

  } ]
```

Блок catch напоминает маленький метод с единственным аргументом. Аргумент используется не всегда.

Если некоторый оператор блока **try** выбрасывает исключение, то все дальнейшие операторы этого блока не выполняются и начинается поиск обработчика, в заголовке которого указан тип, совпадающий с типом выброшенного объекта-исключения, либо тип, наследником которого (не обязательно непосредственным) он является. Поиск осуществляется по следующим правилам:

- если оператор, выбросивший исключение, не находится в **try**-блоке, то происходит выход из метода и продолжение поиска обработчика в вызывающем методе (это может повторяться несколько раз, пока один из операторов, вызывающих метод, не окажется в блоке **try**);
- обработчик ищется среди **catch**-блоков, непосредственно следующих за данным **try**-блоком;
- если подходящего обработчика не находится, поиск продолжается среди **catch**-блоков, относящихся к включающему **try**-блоку и т. д. (при этом также может происходить выход из одного или нескольких методов);
- если подходящего обработчика не находится вообще, исключение ловит обработчик по умолчанию, который печатает описание исключения, трассу стека и завершает работу программы.

Если подходящий обработчик найден, то он выполнятся, причём объект-исключение передаётся обработчику в качестве аргумента. По окончании выполнения обработчика управление передаётся на оператор, непосредственно следующий за всеми блоками **catch**, находящимися после блока, обработавшего исключение. Выполняется только один блок catch.

Поскольку класс **Exception** является суперклассом всех пользовательских исключений, для того чтобы обработать все выброшенные исключения независимо от их типов, можно использовать конструкцию:

```
catch(Exception e)
```

Обработчики исключений-субклассов должны находиться до обработчиков исключений-суперклассов, так как иначе все такие исключения будут обрабатываться обработчиком исключений-суперклассов и возникнет недоступный код.

Иногда требуется гарантировать, что определенный участок кода будет выполняться независимо от того, какие исключения были возбуждены и перехвачены. Для создания такого участка кода используется ключевое слово finally. Даже в тех случаях, когда в методе нет соответствующего возбужденному исключению раздела catch, блок finally будет выполнен до того, как управление перейдет к операторам, следующим за разделом try. У каждого раздела try должен быть по крайней мере или один раздел catch или блок finally. Блок finally очень удобен для закрытия файлов и освобождения любых других ресурсов, захваченных для временного использования в начале выполнения метода.

При наличии блока **finally** операторы этого блока выполняются независимо от того, было ли исключение выброшено или нет:

- если исключение произошло и было обработано, блок **finally** выполняется после обработчика исключений;
- если обработчик не был найден, то блок **finally** выполняется перед поиском обработчика во включающем блоке;
- если выход из **try**-блока был выполнен посредством оператора **return**, то блок **finally** выполняется перед выходом из метода.

Целью большинства хорошо сконструированных catch-разделов должна быть обработка возникшей исключительной ситуации и приведение переменных программы в некоторое разумное состояние — такое, чтобы программу можно было продолжить так, будто никакой ошибки и не было.

7.3. **Стандартные исключения**. Иерархия исключений в Java представлена следующим образом: родительский класс для всех Throwable. От него унаследовано 2 класса: Exception и Error. От класса Exception унаследован еще RuntimeException. Error – критические ошибки, который могут возникнуть в системе (например, StackOverflowError). Как правило обрабатывает их система. Если они возникают, то приложение закрывается, так как при данной ситуации работа не может быть продолжена.

Полный набор стандартных исключений можно увидеть в документации:
Java Platform, Standard Edition 7 API Specification
http://docs.oracle.com/javase/7/docs/api/

У всех стандартных исключений есть два конструктора: по умолчанию (без аргументов) и со строковым аргументом. Использование исключений:

```
public class ExceptionMethods {
    public static void main(String[] args) {
        try {
            throw new Exception("Мое исключение");
        } catch(Exception e) {
            System.out.println(e.getMessage());
            System.out.println(e);
        }
    }
}
run:Моеисключение

java.lang.Exception: Моеисключение
```

Exception – это проверенные исключения. Это значит, что если метод бросает исключение, которое унаследовано от Exception (напр. IOException), то этот метод должен быть обязательно заключен в блок try-catch. Сам метод, который бросает исключение, должен в сигнатуре содержать конструкцию throws. Проверенные (checked) исключения означают, что исключение можно было предвидеть и, соответственно, оно должно быть обработано, работа приложения должна быть продолжена.

Существует важная особенность, касающаяся исключений, являющихся экземплярами класса **Exception** и его субклассов, кроме класса **RuntimeException** и его субклассов. Если такое исключение выбрасывается из метода, то тип его должен быть обязательно указан в утверждении **throws** в заголовке этого метода

```
[ спецификдоступа_ ] типданных_ имяметода_список(аргументов_)
[ throws списокимёнклассовисключенийвыбрасываемыхметодом_____ ]
```

а также всех методов, которые вызывают такие методы, но не обрабатывают соответствующие исключения. В противном случае компилятор выдаст сообщение об ошибке. Это сделано для того, чтобы привлечь внимание программиста к исключительным ситуациям, которые требуют обязательной обработки. В стандартной библиотеке к таким исключениям относятся: ошибки ввода/вывода (**IOException**), ошибки обращения к базе данных (**SQLException**) и некоторые другие.

Пример такого исключения — это попытка создать новый файл, который уже существует (IOException). В данному случае, работа приложения должна быть продолжена и пользователь должен получить уведомление, по какой причине файл не может быть создан.

```
 try {
     File.createTempFile("prefix", "");
} catch (IOException e) {
     // Handle IOException
}
public static File createTempFile(String prefix, String suffix) throws
IOException {
...
}
```

В данном примере можно увидеть, что метод createTempFile может выбрасывать IOException, когда файл не может быть создан. И это исключение должно быть обработано соответственно. Если попытаться вызвать этот метод вне блока try-catch, то компилятор выдаст ошибку и будет предложено 2 варианта исправления: окружить метод блоком try-catch или метод, внутри которого вызывается File.createTempFile, должен выбрасывать исключение IOException (чтобы передать его на верхний уровень для обработки).

RuntimeException – это непроверенные исключения. Они возникают во время выполнения приложения. К таким исключениям относится, например, NullPointerException. Они не требуют обязательного заключения в блок try-catch. Когда RuntimeException возникает, это свидетельствует о ошибке, допущенной программистом (неинициализированный объект, выход за пределы массива и т.д.). Поэтому данное исключение не нужно обрабатывать, а нужно исправлять ошибку в коде, чтобы исключение вновь не возникало.

При выбрасывании исключений, являющихся экземплярами класса **RuntimeException** и его субклассов, никакой специальной декларации не требуется. В библиотеке к таким исключениям относятся: исключение при выполнении недопустимых математических операций (**ArithmeticException**), попытка обращения к несуществующему элементу массива (**IndexOutOfBoundsException**), к полю или методу несуществующего объекта (**NullPointerException**) и другие. Такие ошибки никогда не должны возникать в правильной

программе, в то же время требование обязательной их обработки привело бы к загромождению исходного текста (например, к необходимости размещения каждой операции деления внутри блока **try** во избежание ошибки деления на ноль).

7.4. **Создание пользовательских исключений.** Невозможно предусмотреть все ошибочные ситуации, поэтому при необходимости программист может создать собственный класс-исключение, являющийся наследником одого из стандартных, наиболее подходящих для этого случая.

```
class SimpleException extends Exception {}
public class UserException {
    public void f() throws SimpleException {
        throw new SimpleException();
    }
    public static void main(String[] args) {
        UserException obj = new JavaApplication1();
        try {
            obj.f();
        } catch(SimpleException e) {
            System.out.println("Исключение перехвачено");
        }
    }
}
```

Отметим, что <<пробрасывание>> исключений через несколько методов является типичным при их обработке — исключение обрабатывается в том контексте, где это становится возможным, при этом стек вызовов <<разворачивается>> без участия программиста. Эта особенность во многом и обусловливает большую практическую ценность исключений.

Механизм исключений позволяет отделить основной код от блка, где обрабатываются ошибки. Кроме того код сокращается, так как несколько одинаковых ошибок обрабатываются одним блоком.

7.5. **Обёртки примитивных типов**. Как уже отмечалось, переменные примитивных типов в Java не являются объектами. В некоторых случаях, однако, требуется их представление как объектов (например, для хранения в контейнерах, см. п. 10.1). Для этого используются так называемые *обёртки примитивных типов* — классы, инкапсулирующие примитивные типы.

Для каждого из восьми примитивных типов определён соответствующий ему класс-обёртка. При создании экземпляров таких типов обёртываемый объект передаётся в конструктор соответствующего класса:

```
Byte(byte value)
Short(short value)
Integer(int value)
Long(long value)
Float(float value)
Double(double value)
Character(char value)
Boolean(boolean value)
```

Кроме того, все классы-обёртки, кроме **Character**, имеют конструкторы с аргументом типа **String**. Они осуществляют преобразование строки к соответствующему примитивному типу и обёртывают полученный результат.

Все классы-обёртки переопределяют методы **equals()**, **toString()**, а также реализуют интерфейс **Comparable<T>**.

Для выполнения обратного преобразования от объектов-обёрток к примитивным типам используются методы

```
byte byteValue()
short shortValue()
int intValue()
...
```

вызываемые на соответствующих объектах. Начиная с версии Java 1.5, большая часть таких преобразований производится автоматически. Все классы-обёртки являются неизменяемыми (immutable).

Классы-обёртки содержат также статические методы

```
static byte parseByte(String s)
static short parseShort(String s)
static int parseInt(String s)
...
```

которые могут использоваться для преобразования строки к примитивным типам. В случае, если преобразование осуществить не удаётся, выбрасывается исключение типа **NumberFormatException**.

8. Ввод/вывод

8.1. Потоки ввода/вывода. Ввод/вывод в Java осуществляется посредством потоковых классов. Понятие потока позволяет абстрагироваться от конкретного физического устройства. Классы можно разделить 1) потоки для вывода, потоки для ввода; 2) байтовые, символьные, форматные. Байтовые потоки предназначены для передачи бинарных данных, а символьные - текстовых. Для представления информации символьные потоки используют Unicode. Потоковая библиотека Java разработана таким образом, что ввод/вывод данных производится практически одинаково, вне зависимости от источников и приёмников передаваемых данных. Эта особенность обусловлена тем, что все потоковые классы унаследованы от одних и тех же абстрактных классов, определяющих методы, посредством которых и происходит взаимодействие пользовательских приложений с потоками.

8.2. Класс File. Класс File работает не с потоками, а непосредственно с файлами. Он может представлять имя определённого файла, а также имена группы файлов, находящихся в каталоге. Если класс представляет каталог, то его метод list() возвращает массив строк с именами всех файлов. Класс File позволяет получить информацию о файле: права доступа, время и дата создания, путь к каталогу. А также осуществлять навигацию по иерархиям подкаталогов.

Чтобы работать с этим классом надо подключить java.io.File

Для создания объектов класса File можно использовать один из следующих конструкторов.

1. File(File dir, String name) - указывается объекта класса File (каталог) и имя файла,
2. File(String path) - указывается путь к файлу без указания имени файла,
3. File(String dirPath, Sring name) - указывается путь к файлу и имя файла,
4. File(URI uri) - указывается объект URI, описывающий файл.

У класса очень много методов, перечислим некоторые.

1. boolean canRead() - доступно для чтения
2. boolean canWrite() - доступно для записи
3. boolean delete() - удаляет файл. Также можно удалить пустой каталог
4. boolean exists() - файл существует или нет
5. String getAbsolutePath() - абсолютный путь файла, начиная с корня системы
6. String getName() - возвращает имя файла
7. String getParent() - возвращает имя родительского каталога
8. String getPath() - путь
9. boolean isFile() - проверяет, что объект является файлом, а не каталогом
10. boolean isDirectory - проверяет, что объект является каталогом
11. long lastModified() - дата последнего изменения, возвращает количество миллисекунд, прошедшее с 1 января 1970 года,
12. boolean renameTo(File newPath) - переименовывает файл. В параметре указывается имя нового файла. Если переименование прошло неудачно, то возвращается false.

```
import java.io.File;
import java.util.Date;

 File f = new File("data.txt");
 Date myDate = new Date(f.lastModified());
```

```
 System.out.println(myDate.toString());
run:
Wed Nov 19 19:45:19 MSK 2014

import java.io.File;
import java.util.Date;
import java.text.SimpleDateFormat;

 File f = new File("data.txt");
 Date myDate = new Date(f.lastModified());
 SimpleDateFormat formatDate = new SimpleDateFormat("dd−MM−yyyy");
 System.out.println(formatDate.format(myDate));
run:
19−11−2014

 System.out.println(new SimpleDateFormat("dd−MM−yyyy HH−mm−ss").
 format( new Date(new File("data.txt").lastModified())));
run:
19−11−2014 19−45−19
```

Абстрактными суперклассами байтовых потоков являются классы **InputStream** и **OutputStream**, символьных - **Reader** и **Writer**. Основные методы этих классов перечислены в табл. 8.1, 8.2.

В случае ошибок ввода-вывода методы потоковых классов выбрасывают исключение класса **IOException** или его подклассов (например, **FileNotFoundException**). Классы этих исключений не являются подклассами класса **RuntimeException** и потому нуждаются в обязательной обработке.

Средства потокового ввода/вывода размещаются в пакете **java.io**.

void write(**int** a) **throws** IOException	Запись одного байта/символа в поток
void write(♢ a) **throws** IOException	Запись содержимого из массива в поток
void write(♢[] a, **int** offset, **int** len) **throws** IOException	Запись содержимого части массива в поток
void flush() **throws** IOException	Сброс буферов вывода, связанных с потоком
void close() **throws** IOException	Закрытие потока

Таблица 8.1. Основные методы абстрактных классов **OutputStream** и **Writer**. Вместо символа ♢ подставляется **int** для байтовых и **char** для символьных потоков

Для чтения и записи используются наследники этих классов, например

```
BufferedInputStream(InputStream in)
BufferedOutputStream(OutputStream out)
BufferedReader(Reader in)
BufferedWriter(Writer out)
```

Буферизованные потоки являются обёртками для других потоков. Они обеспечивают повышение эффективности ввода/вывода за счёт буферизации. Функциональность буферизованных потоков обеспечивается классами **BufferedInputStream**, **BufferedOutputStream** (входной и выходной байтовые потоки с буферизацией), **BufferedReader** и **BufferedWriter** (аналогичные символьные потоки). Для создания объектов этих классов используются конструкторы, принимающие в качестве аргумента ссылку на обёртываемый поток:

Кроме того, входной символьный поток с буферизацией добавляет метод

int read() **throws** IOException	Чтение одного байта/символа из потока (в случае достижения конца файла возвращает --1)
int read(♢[] a) **throws** IOException	Заполнение массива содержимым, считанным из потока (возвращает фактическое количество считанных байтов/символов)
int read(♢[] a, **int** offset, **int** len) **throws** IOException	Заполнение части массива содержимым, считанным из потока
boolean ready() **throws** IOException	Проверка наличия данных в потоке (если метод возвращает **false**, то следующий вызов метода **read()** будет ожидать данных и завершится только при их получении). *Метод определяется только интерфейсом **Reader**!*
long skip(**long** n) **throws** IOException	Пропуск заданного количества байтов/символов в потоке
void close() **throws** IOException	Закрытие потока

Таблица 8.2. Основные методы абстрактных классов **InputStream** и **Reader**. Вместо символа ♢ подставляется **int** для байтовых и **char** для символьных потоков

```
String readLine() throws IOException
```

осуществляющий чтение одной строки из входного символьного потока. В случае достижения конца файла этот метод возвращает значение **null**.

8.3. Байтовые потоки, связанные с файлами. Байтовые потоки, связанные с файлами, используются для работы с бинарными файлами. Для этих потоков ввод/вывод данных осуществляется непосредственно без выполнения каких-либо преобразований над содержимым. Функциональность байтовых потоков, связанных с файлами, обеспечивается классами **FileInputStream** и **FileOutputStream**. Для создания объектов используются следующие конструкторы:

```
FileInputStream(String fileName) throws FileNotFoundException
FileOutputStream(String fileName) throws FileNotFoundException
FileOutputStream(String fileName, boolean append)
        throws FileNotFoundException
```

Все конструкторы пытаются открыть файл, имя которого им передаётся в качестве аргумента. Если при создании выходного потока соответствующий файл уже существовал, он будет усечён до нулевой длины. Однако, если использовать третий из числа перечисленных выше конструкторов со значением аргумента **append** равным **true**, файл будет открыт в режиме дозаписи в конец.

8.4. Символьные потоки-обёртки для байтовых потоков. Поскольку вся символьная информация представляется в Java в Unicode, в то время как внешние данные зачастую представлены в других кодировках, то для эффективного взаимодействии Java с внешним миром требуется преобразование кодировок. Эту операцию выполняют символьные потоки-обёртки. Они могут присоединяться к любым байтовым потокам (<<обёртывают>> его) и позволяют обращаться с полученным потоком как с символьным, выполняя все необходимые преобразования.

Рассматриваемым потокам соответствуют классы **InputStreamReader** (преобразует данные, считываемые из байтового потока в Unicode) и **OutputStreamWriter** (преобразует символьные данные, выводимые в байтовый поток из Unicode). Для создания объектов данных классов используются следующие конструкторы:

```
InputStreamReader(InputStream in)
InputStreamReader(InputStream in, String charSet)
OutputStreamWriter(OutputStream out)
OutputStreamWriter(OutputStream out, String charSet)
```

Требуемая кодировка определяется аргументом **charSet**. Если она не указана, используется кодировка системной локали.

8.5. Символьные потоки, связанные с файлами. Символьные потоки, связанные с файлами, предназначены для работы с текстовыми файлами. Фактически они представляют собой комбинацию байтового потока и символьного потока-обёртки, осуществляющего преобразование. При этом всегда используется кодировка системной локали.

Данным потокам соответствуют классы **FileReader** и **FileWriter** со следующими конструкторами:

```
FileReader(String fileName) throws FileNotFoundException
FileWriter(String fileName) throws FileNotFoundException
FileWriter(String fileName, boolean append) throws FileNotFoundException
```

Используются они полностью аналогично конструкторам байтовых потоков, связанных с файлами.

8.6. Консольный ввод/вывод. Существует класс System. Его поля:

```
static PrintStream  err// The "standard" error output stream.
static PrintStream  out// The "standard" output stream.
static InputStream  in// The "standard" input stream.
```

Для вывода в них обычно используются методы **print()** и **println()**, например:

```
System.err.println("Это строкавыводитсявстандартныйпотокошибки");
```

Для ввода в Java создан объект **System.in**, однако, он является не символьным, а байтовым потоком и потому для ввода данных из него требуется преобразование в Unicode. Последнее можно осуществить, использовав поток-обёртку **InputStreamReader**:

```
Reader in = new InputStreamReader(System.in);
```

Подробнее:

```
        try{
        System.out.print("Введите символ: ");
        Reader in = new InputStreamReader(System.in);
        System.out.println((char)in.read());
        } catch(IOException e)
        {
            System.out.println("Ошибка вводавывода/");
        }
```

Поскольку обычно ввод из стандартного входного потока осуществляется построчно, а не посимвольно, то часто используется ещё одна обёртка типа **BufferedReader**:

```
Reader in = new BufferedReader(new InputStreamReader(System.in));
```

После этого можно использовать методы **int read()** и **String readLine()** созданного потока:

```
String s = in.readLine();
```

Подробнее:

```
try {
                Reader in = new BufferedReader(new InputStreamReader(System.in));
                System.out.print("Введите символ: ");
                System.out.println("Вы ввели: " + (char) in.read());
            } catch (IOException e) {
                System.out.println("Ошибка вводавывода/");
            }
```

```
try {
        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        System.out.print("Введите строку: ");
        System.out.println("Вы ввели: " + in.readLine());
    } catch (IOException e) {
            System.out.println("Ошибка вводавывода/");
    }
```

8.7. Класс Scaner. Для ввода данных используется класс Scanner из библиотеки пакетов Java. Этот класс надо импортировать в той программе, где он будет использоваться.

Для работы с потоком ввода необходимо создать объект класса Scanner, при создании указав, с каким потоком ввода он будет связан. Стандартный поток ввода (клавиатура) в Java представлен объектом — System.in. А стандартный поток вывода (дисплей) — уже знакомым вам объектом System.out.

```
import java.util.Scanner; // импортируемкласс
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in); // создаёмобъектклассаScanner
        System.out.print("Введите целоечисло: ");
        if(sc.hasNextInt()) { // возвращаетистиннуеслиспотокавводаможносчитатьцелоечисло

            int i = sc.nextInt(); // считываетцелоечислоспотокавводаисохраняемвпеременную

            // double i = sc.nextDouble();
            //
            System.out.println(i*2);
        } else {
            System.out.println("Вы ввелинецелоечисло");
        }
    }
}
```

Метод hasNextDouble(), применённый объекту класса Scanner, проверяет, можно ли считать с потока ввода вещественное число типа double, а метод nextDouble() — считывает его. Имеется также метод nextLine(), позволяющий считывать целую последовательность символов, т.е. строку, а, значит, полученное через этот метод значение нужно сохранять в объекте класса String. В следующем примере создаётся два таких объекта, потом в них поочерёдно записывается ввод пользователя, а далее на экран выводится одна строка, полученная объединением введённых последовательностей символов.

```
import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
```

```
        String s1, s2;
        s1 = sc.nextLine();
        s2 = sc.nextLine();
        //sc.next();
        System.out.println(s1 + s2);
    }
}
```

Существует и метод hasNext(), проверяющий остались ли в потоке ввода какие-то символы.

8.8. Ввод-вывод данных в файл. Чтение строк:

```
public static void main(String[] args) {
        try {
            BufferedReader in = new BufferedReader(new FileReader("data.txt"));
            PrintWriter out = new PrintWriter("rezult.txt");
            String s;
            while ((s = in.readLine()) != null) {
                if (s.length() > 10) {
                    out.println(s);
                }
            }
            out.close();
            in.close();
        } catch (IOException e) {
            System.out.println("Ошибка вводавывода/");
        }
    }
```

Чтение чисел:

```
            Scanner in = new Scanner(new File("data.txt"));
            int sum = 0;
            while (in.hasNextInt()) {
                sum = sum + in.nextInt();
            }
            System.out.println(sum);
```

9. Обработка строк

9.1. Класс String. В этой главе обсуждаются средства языка Java для работы со строками. В языках C и C++ отсутствует встроенная поддержка такого объекта, как строка. В них при необходимости передается адрес последовательности байтов, содержимое которых трактуется как символы до тех пор, пока не будет встречен нулевой байт, отмечающий конец строки. В пакет java.lang встроен класс, инкапсулирующий структуру данных, соответствующую строке. Этот класс, называемый String, не что иное, как объектное представление неизменяемого символьного массива. В этом классе есть методы, которые позволяют сравнивать строки, осуществлять в них поиск и извлекать определенные символы и подстроки. Класс StringBuffer используется тогда, когда строку после создания требуется изменять.

И String, и StringBuffer объявлены final, что означает, что ни от одного из этих классов нельзя производить подклассы. Это было сделано для того, чтобы можно было применить некоторые виды оптимизации позволяющие увеличить производительность при выполнении операций обработки строк.

Класс **String** является основным классом, предназначенным для хранения и обработки строк символов. Тот факт, что объекты типа String в Java неизменны, позволяет транслятору применять к операциям с ними различные способы оптимизации.

Для создания экземпляров класса **String** может быть использован один из следующих конструкторов:

```
String()
String s = new String();

String(String str)
String(StringBuffer strbuf)
String(char[] arr)
char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
String s = new String(chars); //"abcdef"

String(char[] arr, int first, int count)
String s = new String(chars,2,3);//"cde"
```

Первый из них создаёт пустую строку, второй и третий копируют содержимое объектов классов **String** и **StringBuffer** в созданный объект. Последние два конструктора позволяют создать строку на основе символьного массива или его части. Кроме того, любая объектная ссылка типа **String** может быть проинициализирована посредством присвоения ей строкового литерала:

```
String filename = "data.txt";
```

Один из общих методов, используемых с объектами String — метод length, возвращающий число символов в строке. В Java интересно то, что для каждой строки-литерала создается свой представитель класса String, так что вы можете вызывать методы этого класса непосредственно со строками-литералами, а не только со ссылочными переменными.

```
String s = "abc";
System.out.println(s.length());
System.out.println("abc".length());
```

Слияние строк. Последовательность выполнения операторов

Для строк доступна операция +, позволяющая соединить несколько строк в одну. Если один из операндов не строка, то он автоматически преобразуется в строку. Для объектов в этих целях используется метод toString(). Рассмотрим следующую строку:

```
String s = "four: " + 2 + 2;
```

В результате получается "four: 22". Для того, чтобы первым выполнилось сложение целых чисел, нужно использовать скобки:

```
String s = "four: " + (2 + 2);
```

В каждом классе есть метод toString — либо своя собственная реализация, либо вариант по умолчанию, наследуемый от класса Object. Класс в нашем очередном примере замещает наследуемый метод toString своим собственным, что позволяет ему выводить значения переменных объекта.

```
class Point {
    public String toString() {
        return "Point(" + x + ", " + y+ ")";
    }
}
class PointApp {
    public static void main(String args[]) {
        Point p = new Point(10, 20);
        System.out.println("p = " + p);
    }
}Результат

p = Point(10.0, 20.0)
```

Если метод toString() не переопределить, то получим на выходе

```
p = pointapp.Point@18a992f
```

Извлечение символов

Для того, чтобы извлечь одиночный символ из строки, вы можете сослаться непосредственно на индекс символа в строке с помощью метода char charAt(int). Если вы хотите в один прием извлечь несколько символов, можете воспользоваться методом void getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin). В приведенном ниже фрагменте показано, как следует извлекать массив символов из объекта типа String.

```
class getCharsDemo {
public static void main(String args[]) {
String s = "This is a demo of the getChars method.";
int start = 10;
int end = 14;
char buf[] = new char[end − start];
s.getChars(start, end, buf, 0);
System.out.println(buf);
} }Результат

demo
```

Обратите внимание — метод getChars не включает в выходной буфер символ с индексом end. Это хорошо видно из вывода нашего примера — выводимая строка состоит из 4 символов.

Для удобства работы в String есть еще одна функция — char[] toCharArray(), которая возвращает в выходном массиве типа char всю строку. Альтернативная форма того же самого механизма позволяет записать содержимое строки в массив типа byte, при этом значения старших байтов в 16-битных символах отбрасываются. Соответствующий метод называется getBytes, и его параметры имеют тот же смысл, что и параметры getChars, но с единственной разницей — в качестве третьего параметра надо использовать массив типа byte.

Сравнение

Если вы хотите узнать, одинаковы ли две строки, вам следует воспользоваться методом boolean equals() класса String. Альтернативная форма этого метода называется equalsIgnoreCase, при ее использовании различие регистров букв в сравнении не учитывается. Ниже приведен пример, иллюстрирующий использование обоих методов:

```
class equalDemo {

public static void main(String args[]) {

String s1 = "Hello";

String s2 = "Hello";

String s3 = "Good−bye";

String s4 = "HELLO";

System.out.println(s1 + " equals " + s2 + " −> " + s1.equals(s2));

System.out.println(s1 + " equals " + s3 + " −> " + s1.equals(s3));

System.out.println(s1 + " equals " + s4 + " −> " + s1.equals(s4));

System.out.println(s1 + " equalsIgnoreCase " + s4 + " −> " +
s1.equalsIgnoreCase(s4));
} }
```

Результат запуска этого примера :

```
Hello equals Hello −> true
Hello equals Good−bye −> false
Hello equals HELLO −> false
Hello equalsIgnoreCase HELLO −> true
```

Метод equals и оператор == выполняют две совершенно различных проверки. Если метод equal сравнивает символы внутри строк, то оператор == сравнивает две переменные-ссылки на объекты и проверяет, указывают ли они на разные объекты или на один и тот же. В очередном нашем примере это хорошо видно — содержимое двух строк одинаково, но, тем не менее, это — различные объекты, так что equals и == дают разные результаты.

```
class EqualsNotEqualTo {
public static void main(String args[]) {
String s1 = "Hello";
String s2 = new String(s1);
System.out.println(s1 + " equals " + s2 + " −> " + s1.equals(s2));
System.out.println(s1 + " == " + s2 + ", −> " + (s1 == s2));
} }
```

Вот результат запуска этого примера:

```
Hello equals Hello —> true
Hello == Hello —> false
```

Зачастую бывает недостаточно просто знать, являются ли две строки идентичными. Для приложений, в которых требуется сортировка, нужно знать, какая из двух строк меньше другой. Для ответа на этот вопрос нужно воспользоваться методом int compareTo(String str) класса String. Если целое значение, возвращенное методом, отрицательно, то строка, с которой был вызван метод, меньше строки-параметра, если положительно — больше. Если же метод compareTo вернул значение 0, строки идентичны. Ниже приведена программа, в которой выполняется пузырьковая сортировка массива строк, а для сравнения строк используется метод compareTo. Эта программа выдает отсортированный в алфавитном порядке список строк.

```
class SortString {
static String arr[] = {"Now", "is", "the", "time", "for", "all",
"good", "men", "to", "come", "to", "the",
"aid", "of", "their", "country" };
public static void main(String args[]) {
for (int j = 0; i < arr.length; j ++){
for (int i = j + 1; i < arr.length; i++) {
(arr[i].compareTo(arr[j]) < 0) {
String t = arr[j];
arr[j] = arr[i];
arr[i] = t;
}
}
System.out.println(arr[j]);
}
} }
```

В класс String включена поддержка поиска определенного символа или подстроки, для этого в нем имеются два метода — int indexOf(String substr) и int lastIndexOf(String substr). Каждый из этих методов возвращает индекс того символа, который вы хотели найти, либо индекс начала искомой подстроки. В любом случае, если поиск оказался неудачным методы возвращают значение -1. В очередном примере показано, как пользоваться различными вариантами этих методов поиска.

```
class indexOfDemo {
public static void main(String args[]) {
String s = "Now is the time for all good men " +
"to come to the aid of their country " +
"and pay their due taxes.";
System.out.println(s);
System.out.println("indexOf(t) = " + s.indexOf('f));
System.out.println("lastIndexOf(t) = " + s.lastIndexOf('f));
System.out.println("indexOf(the) = " + s.indexOf("the"));
System.out.println("lastIndexOf(the) = " + s.lastIndexOf("the"));
System.out.println("indexOf(t, 10) = " + s.indexOf('f , 10));
System.out.println("lastIndexOf(t, 50) = " + s.lastIndexOf('f , 50));
System.out.println("indexOf(the, 10) = " + s.indexOf("the", 10));
System.out.println("lastIndexOf(the, 50) = " + s.lastIndexOf("the", 50));
} }
```

Ниже приведен результат работы этой программы. Обратите внимание на то, что индексы в строках начинаются с нуля.

```
Now is the time for all good men to come to the aid of their country
and pay their due taxes.
indexOf(t) = 7
lastIndexOf(t) = 87
indexOf(the) = 7
lastIndexOf(the) = 77
index0f(t, 10) = 11
lastIndex0f(t, 50) = 44
index0f(the, 10) = 44
lastIndex0f(the, 50) = 44
```

Возможно, несколько неожиданной особенностью класса **String** является то, что экземпляры этого класса *не могут быть изменены после их создания* (immutable). Однако это не создаёт ограничений для их использования, поскольку все методы, которые должны были бы изменять строку, просто создают новую модифицированную строку, оставляя исходную без изменений. Поясним работу этого механизма на примере:

```
String s = "abcd";
s = s.toUpperCase();
```

Здесь метод **toUpperCase()** создаёт новую строку, содержащую последовательность символов "ABCD", и возвращает ссылку на эту строку, которая присваивается переменной **s**, старое значение переменной теряется. Исходная строка остаётся в неизменном виде и, поскольку на неё больше не осталось объектных ссылок, будет удалена сборщиком мусора.

Модификация строк при копировании

Поскольку объекты класса String нельзя изменять, всякий раз, когда вам захочется модифицировать строку, придется либо копировать ее в объект типа StringBuffer, либо использовать один из описываемых ниже методов класса String, которые создают новую копию строки, внося в нее ваши изменения.

substring

Вы можете извлечь подстроку из объекта String, используя метод substring. Этот метод создает новую копию символов из того диапазона индексов оригинальной строки, который вы указали при вызове. Можно указать только индекс первого символа нужной подстроки — тогда будут скопированы все символы, начиная с указанного и до конца строки. Также можно указать и начальный, и конечный индексы — при этом в новую строку будут скопированы все символы, начиная с первого указанного, и до (но не включая его) символа, заданного конечным индексом.

"Hello World".substring(6) -> "World"

"Hello World".substring(3,8) -> "lo Wo"

replace

Методу replace в качестве параметров задаются два символа. Все символы, совпадающие с первым, заменяются в новой копии строки на второй символ.

"Hello".replace('l' , 'w') -> "Hewwo"

toLowerCase и toUpperCase

Эта пара методов преобразует все символы исходной строки в нижний и верхний регистр, соответственно.

"Hello".toLowerCase() -> "helloHello".toUpperCase() -> "HELLO"

trim

И, наконец, метод trim убирает из исходной строки все ведущие и замыкающие пробелы.

int length()	Получение длины строки
char charAt(**int** index)	Извлечение символа
char[] toCharArray()	Получение строки в виде символьного массива
boolean equals(String str)	Сравнение строк на равенство
boolean equalsIgnoreCase(String str)	Сравнение строк без учета регистра
int compareTo(String str)	Лексикографическое сравнение строк
int compareToIgnoreCase(String str)	Лексикографическое сравнение строк без учета регистра
boolean startsWith(String prefix)	Проверка, начинается ли строка с заданной подстроки
boolean endsWith(String suffix)	Проверка, заканчивается ли строка заданной подстрокой
int indexOf(String subStr) **int** indexOf(String subStr, **int** fromIndex)	Поиск первого вхождения подстроки в строке с начала строки/с заданной позиции
int lastIndexOf(String subStr) **int** lastIndexOf(String subStr, **int** fromIndex)	Поиск последнего вхождения подстроки в строке с начала строки/с заданной позиции
String substring(**int** beginIndex, **int** endIndex)	Получение подстроки (символ endIndex не входит в подстроку!)
String substring(**int** beginIndex)	Получение хвоста строки
String concat(String str)	Конкатенация строк
String toUpperCase() String toLowerCase()	Преобразование строки к верхнему/ нижнему регистру
String trim()	Удаление ведущих и завершающих пробелов в строке
String replace(String target, String replacement)	Замена подстроки другой строкой
boolean matches(String regex)	Проверка строки на соответствие регулярному выражению
String replaceFirst(String regex, String replacement) String replaceAll(String regex, String replacement)	Замена первой подстроки/всех подстрок, соответствующих регулярному выражению, заданной подстрокой
String[] split(String regex)	Разбиение строки на подстроки (разделители задаются регулярным выражением)

Таблица 9.1. Основные методы класса **String**

“Hello World “.trim() -> "Hello World"

Основные методы класса **String** приведены в табл. 9.1.

9.2. Регулярные выражения. Регулярные выражения - это выражения, описывающие структуру текстовой строки с использованием обычных символов и *метасимволов*, определяющих свойства строки в целом или её отдельных частей. Средства стандартной библиотеки Java позволяют выполнять проверку строк на соответствие заданному регулярному выражению, а также осуществлять замену подстрок, удовлетворяющих регулярным выражениям, другими подстроками.

Основные метасимволы, используемые в регулярных выражениях в Java, приведены в табл. 9.2. Примеры регуляных выражений приведены в табл. 9.3.

Для проверки строки на соответствие заданному регулярному выражению используется метод

```
boolean matches(String regex)
```

класса **String**. Заметим, что поскольку он проверяет *всю строку* на соответствие регулярному выражению, то использование символов <<⍰>> и <<$>> в этом случае излишне.

Метасимвол(ы)	Значение
\	обозначение специального символа или экранирование обычного
.	любой символ, кроме ограничителей строки (<<\n>>, <<\r>> и пр.)
|	выбор одного из двух регулярных выражений
[ ]	один из символов, входящих в заданный набор
[^]	один из символов, не входящих в заданный набор
[--]	один из символов, входящих в диапазон
^	признак начала строки
$	признак конца строки
*	ноль и более предыдущих символов
+	один и более предыдущих символов
?	ноль или один предыдущий символ
{*k*}	ровно *k* предыдущих символов
{*k*,}	не менее *k* предыдущих символов
{*k*,*m*}	не менее *k* и не более *m* предыдущих символов
( )	группировка/сохранение строк, соответствующих регулярным выражениям
k	*k*-я сохранённая группа

Таблица 9.2. Метасимволы регулярных выражений

Регулярное выражение	Значение
abc|def	последовательность символов <<abc>> или <<def>>
a.c	символы <<a>> и <<c>>, разделённые любым символом
https?://	последовательность символов <<http://>> или <<https://>>.
^a.*z$	строка, начинающаяся с символа <<a>> и заканчивающаяся символом <<z>>
[A--Z]+	непустая последовательность прописных латинских букв
[^0--9]{5,}	последовательность не менее чем из пяти символов, не являющихся цифрами
\.\.\.	последовательность символов <<...>>
(.+)a\1	две одинаковые подстроки, разделённые символом <<a>>

Таблица 9.3. Примеры регулярных выражений

Следующие методы заменяют соответственно первое и все вхождения *подстроки*, соответствующей регулярному выражению:

```
String replaceFirst(String regex, String replacement)
String replaceAll(String regex, String replacement)
```

Например, следующий оператор заменяет многоточие, находящееся в конце строки, вопросительным знаком:

```
someString.replaceFirst("\\.\\.\\.$", "?");
```

Поскольку символ <<.>> является метасимволом, то для того, чтобы использовать его как обычный символ, его следует экранировать символом <<\>>. Однако последний символ используется в Java для обозначения специальных символов (таких, как <<\n>>), поэтому также должен экранироваться: <<\\.>>.

В замещающей строке можно использовать метасимвол <<$>> с последующим номером, обозначающий соответствующую группу в заменяемой строке. Например, следующий оператор заменяет угловые скобки в строке квадратными, при этом содержимое скобок остаётся прежним:

```
someString.replaceFirst("<(.*)>", "[$1]");
```

Следует учитывать, что если в строке содержится несколько открывающих и закрывающих угловых скобок, то заменены будут первая открывающая и последняя закрывающая, поскольку шаблону <<.*>> соответствует любое количество любых символов, в том числе все вложенные скобки.

Метод

```
String[] split(String regex)
```

позволяет разбить строку на подстроки, используя для описания разделителей регулярные выражения. Результат записывается в строковой массив. Например, оператор

```
String[] m = "abc:::de:f::gh".split(":+");
```

заполнит массив m строками <<abc>>, <<de>>, <<f>> и <<gh>>.

В заключение отметим, что регулярные выражения, по-видимому, являются самым мощным инструментом обработки строковых данных, и их возможности значительно шире, чем это удалось показать на приведённых примерах. Более полную информацию о регулярных выражениях можно получить в книге [?].

9.3. Преобразование к строке и операция конкатенации. Класс **String** является в некотором смысле исключительным классом в Java, поскольку любой тип данных может быть преобразован к нему. Для примитивных типов такое преобразование даёт их естественное строковое представление, для объектов вызывается метод **toString()**, определённый в классе **Object** и, следовательно, присутствующий в любом классе Java.

Для строк определена операция конкатенации, обозначаемая знаком **+**. Это бинарная операция, один из аргументов которой должен иметь тип **String**. Она осуществляет автоматическое преобразование другого аргумента к типу **String** (если это необходимо) и слияние полученных строк. Заметим, что это единственный случай, когда преобразование к строке осуществляется неявно. Существует также операция конкатенации с присваиванием **+=**, первый аргумент которой должен иметь тип **String** (и обязательно быть *lvalue*), а второй может быть произвольным. При выполнении операции он будет преобразован к типу **String**.

Приведём пример использования преобразования к строке и операции конкатенации. Для класса <<точка плоскости>> можно определить операцию преобразования к строке следующим образом:

```
class Point
{
    private double x, y;
    public String toString()
    {
        return "(" + x + ", " + y + ")";
    }
    public Point(double _x, double _y)
    {
        x = _x; y = _y;
    }
}
```

Здесь автоматическое преобразование величин типа **double** к строкам осуществляется в выражении оператора **return**. Определённый выше метод **toString()** может быть использован следующим образом:

```
Point p = new Point(2.0, 5.0);
System.out.println(p); // выводнаэкран: (2.0, 5.0)
```

Для того чтобы получить и вывести на экран строковое представление объекта, метод **println()** вызывает метод **toString()** класса **Point**.

Иногда требуется управлять видом строкового представления. Для этого может быть использован статический метод **format()** класса **String**:

```
static String format(String format, Object. . . args)
```

Он принимает форматную строку и переменное количество выражений и преобразует их в строку в формате, определённом форматной строкой. Правила написания форматной строки в целом соответствуют правилам написания форматной строки функции **printf()** в языке C. Например,

```
System.out.println(String.format("%04d%4.1f", 25, 3.58));
```

выведет <<0025 3.6>>.

9.4. Класс StringBuffer. Класс **StringBuffer** предназначен для работы с модифицируемыми строками. Каждый объект этого класса содержит некоторый буфер, хранящий строку. При изменениях строки размер буфера может автоматически изменяться.

Как правило, использование объектов класса **StringBuffer** в программах не является необходимым, поскольку все те же операции можно выполнить, используя объекты класса **String**. Однако в некоторых случаях использование класса **StringBuffer** повышает эффективность программы, поскольку позволяет избежать многократного копирования модифицированных строк.

Для создания экземпляров класса **StringBuffer** используются следующие конструкторы:

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

Первые два из них создают объект, хранящий пустую строку с начальным буфером длины 16 в первом случае и заданной длины во втором случае. Третий конструктор инициализирует буфер заданной строкой. Размер буфера в этом случае устанавливается равным длине строки, увеличенной на 16.

Основные методы класса **StringBuffer** приведены в табл. 9.4.

int length()	Получение длины строки
char charAt(**int** index)	Извлечение символа
char setCharAt(**int** index)	Изменение символа
StringBuffer append(String str) StringBuffer append(**int** i) StringBuffer append(Object obj) ...	Добавление строки к концу (существуют перегруженные версии для всех примитивных типов и типа **Object**)
StringBuffer insert(**int** index, String str)	Вставка строки в заданную позицию
StringBuffer delete(**int** beginIndex, **int** endIndex)	Удаление подстроки
StringBuffer replace(**int** beginIndex, **int** endIndex, String str)	Замена одной подстроки другой
StringBuffer reverse()	Обращение строки

Таблица 9.4. Методы класса **StringBuffer**

В отличие от методов модификации строк класса **String**, аналогичные методы класса **StringBuffer** изменяют именно тот экземпляр объекта, на котором они вызываются. При этом они возвращают ссылку на этот объект (**this**). Последнее сделано для того, чтобы можно было объединять набор последовательных операций над строкой в одно выражение подобно тому, как это сделано в стандартной библиотеке языка C:

```
StringBuffer sb = new StringBuffer("abcd");
sb.append("ef").reverse();
System.out.println(sb); // выводитнаэкран: fedcba
```

Выведем в файл все предложения длиннее 10 слов, заканчивающиеся точкой, где слова разделены пробелами и запятыми.

```
public static void main(String[] args) {
        // TODO code application logic here
        String s, buf="";
        try {
            BufferedReader in = new BufferedReader(new FileReader("data.txt"));
            PrintWriter out = new PrintWriter("rezult.txt");
            while ((s = in.readLine()) != null) {
                if(s.indexOf(".")==-1){
                    buf+=s+"\n";
                }
                else {
                    while(s.indexOf(".")!=-1) {
                        buf+=s.substring(0, s.indexOf("."));
                        if(buf.split("[ ,]+").length>10)
                        //String[] a=buf.split("[ ,]+");
                            out.print(buf+".");
                        buf="";
                        s=s.substring(s.indexOf(".")+1);
                    }
                    buf=s+"\n";
                }
            }
            out.close();
            in.close();
```

```
        } catch (IOException e) {
            System.out.println("Ошибка вводавывода/");
        }
    }
```

10. Контейнеры

10.1. Обзор контейнеров. *Контейнерами* называются объекты, способные хранить другие объекты. Они различаются способом организации, временем выполнения основных операций (добавления, извлечения и удаления элементов), а также поддерживаемыми способами обхода элементов контейнера.

Контейнеры в Java можно разделить на три группы: массивы, коллекции и ассоциативные массивы. Массивы были рассмотрены ранее. Их отличительной чертой является невозможность изменения размеров после создания. Все остальные контейнеры в Java динамически изменяют свой размер при добавлении и удалении элементов.

Самую многочисленную группу контейнеров образуют *коллекции*. Все классы-коллекции в Java являются *параметризованными* типами (generics). Параметризованный тип - это класс или интерфейс, имеющий параметр - имя типа, которое должно задаваться при создании объектов. Параметризованные типы напоминают шаблоны C++ и предназначены в первую очередь для того, чтобы обеспечить контроль правильности типов на этапе компиляции. Они представляют собой очень гибкий и в то же время сложный механизм.

Параметром классов-коллекций Java является тип объектов, которые они могут хранить (здесь и всюду ниже этот тип обозначен буквой **E**). Параметр может быть только классом или интерфейсом. Для хранения в коллекциях данных примитивных типов следует использовать классы-обёртки.

Классы-обертки

Во многих случаях предпочтительней работать именно с объектами, а не с примитивными типами. Так, например, при использовании коллекций просто необходимо значения примитивных типов представлять в виде объектов.

Для этих целей и предназначены так называемые классы-обертки. Для каждого примитивного типа Java существует свой класс-обертка. Такой класс является неизменяемым (если необходим объект, хранящий другое значение, его нужно создать заново), к тому же имеет атрибут final - от него нельзя наследовать класс. Все классы-обертки (кроме Void) реализуют интерфейс Serializable, поэтому объекты любого (кроме Void) класса-обертки могут быть сериализованы. (Сериализация это процесс сохранения состояния объекта в последовательность байт; десериализация это процесс восстановления объекта, из этих байт. Должен быть универсальный и эффективный протокол передачи объектов между компонентами. Сериализация создана для этого, и компоненты Java используют этот протокол для передачи объектов.)

Классы-обертки: Byte, Short, Integer, Long, Float, Double, Character, Boolean. Все классы-обертки реализуют интерфейс Comparable. Все классы-обертки числовых типов имеют метод equals(Object), сравнивающий примитивные значения объектов, hashCode(), toString(). Также классы-обертки содержат статические методы для обеспечения удобного манипулирования соответствующими примитивными типами, например, преобразование к строковому виду (toString()).

Что такое хеш-код?

Хэш-функции это функции, предназначенные для сжатия произвольного сообщения или набора данных, записанных, как правило, в двоичном алфавите, в некоторую битовую комбинацию фиксированной длины, называемую сверткой. Хэш-функции имеют разнообразные применения при проведении статистических экспериментов, при тестировании логи-

ческих устройств, при построении алгоритмов быстрого поиска и проверки целостности записей в базах данных, в системах защиты.

В общем случае однозначного соответствия между исходными данными и хеш-кодом нет в силу того, что количество значений хеш-функций меньше, чем число вариантов значений входного массива; существует множество массивов с разным содержимым, но дающих одинаковые хеш-коды - так называемые коллизии. Вероятность возникновения коллизий играет немаловажную роль в оценке качества хеш-функций.

Существует множество алгоритмов хеширования с различными свойствами (разрядность, вычислительная сложность, криптостойкость и т. п.). Выбор той или иной хеш-функции определяется спецификой решаемой задачи. Простейшими примерами хеш-функций могут служить контрольная сумма.

Хорошая хеш-функция должна удовлетворять двум свойствам: быстро вычисляться; минимизировать количество коллизий.

В Java набор данных - это объект, хеш-код это целочисленный результат работы метода, которому в качестве входного параметра передан объект. Все классы наследуют базовую реализацию hashCode() класса java.lang.Object, но лучше переопределять этот метод для более эффективной обработки специфических данных.Метод hashCode() возвращает значение int (4 байта), которое является числовым представлением объекта. Этот хэш-код используется, например, коллекциями для более эффективного хранения данных и, соответственно, более быстрого доступа к ним.

```
public class Main {
    public static void main(String[] args) {
        Object object = new Object();
        int hCode;
        hCode = object.hashCode();
        System.out.println(hCode);
    }
}
```

В результате выполнения программы в консоль выведется целое 10-ти значное число. Это число и есть наша битовая строка фиксированной длины. Хеш-код это число, у которого есть свой предел, который для java ограничен примитивным целочисленным типом int.

Если в Java Вы используете объект собственного класса в качестве наполнителя для коллекций HashSet, HashMap, Hashtable или любых других коллекций, которые хранят объекты в группах, то Вам также необходимо переопределить метод hashCode(). Это необходимо для правильной и более эффективной работы с коллекциями. Также всегда необходимо переопределять метод hashCode() если Вы переопределили метод equals().

Для одного и того же объекта метод hashCode() должен возвращать одно и то же значение в течении всей "жизни"объекта. Следует понимать, что множество возможных хеш-кодов ограничено примитивным типом int, а множество объектов ограничено только нашей фантазией. Отсюда следует утверждение: Множество объектов мощнее множества хеш-кодов. Из-за этого ограничения, вполне возможна ситуация, что хеш-коды разных объектов могут совпасть.

Здесь главное понять, что: Если хеш-коды разные, то и входные объекты гарантированно разные. Если хеш-коды равны, то входные объекты не всегда равны.

Ситуация, когда у разных объектов одинаковые хеш-коды называется коллизией. Вероятность возникновения коллизии зависит от используемого алгоритма генерации хеш-кода. При вычислении хэш-кода для объектов класса Object по умолчанию используется Park-

Miller RNG алгоритм. В основу работы данного алгоритма положен генератор случайных чисел. Это означает, что при каждом запуске программы у объекта будет разный хэш-код.

Рассмотрим более подробно некоторые из классов-оберток.

Integer

Наиболее часто используемые статические методы:

public static int parseInt(String s) - преобразует строку, представляющую десятичную запись целого числа, в int ;

public static int parseInt(String s, int radix) - преобразует строку, представляющую запись целого числа в системе счисления radix, в int.

Оба метода могут возбуждать исключение NumberFormatException, если строка, переданная на вход, содержит нецифровые символы.

Не следует путать эти методы с другой парой похожих методов: public static Integer valueOf(String s), public static Integer valueOf(String s, int radix)

Данные методы выполняют аналогичную работу, только результат представляют в виде объекта-обертки.

Существует также два конструктора для создания экземпляров класса Integer: Integer(String s) - конструктор, принимающий в качестве параметра строку, представляющую числовое значение. Integer(int i) - конструктор, принимающий числовое значение.

public static String toString(int i) - используется для преобразования значения типа int в строку.

Далее перечислены методы, преобразующие int в строковое восьмеричное, двоичное и шестнадцатеричное представление:

public static String toOctalString(int i) - восьмеричное;

public static String toBinaryString(int i)- двоичное;

public static String toHexString(int i) - шестнадцатеричное.

Имеется также две статические константы:

> Integer.MIN_VALUE — минимальное**int** значение;
> Integer.MAX_VALUE — максимальное**int** значение.

Аналогичные константы, описывающие границы соответствующих типов, определены и для всех остальных классов-оберток числовых примитивных типов.

public int intValue() возвращает значение примитивного типа для данного объекта Integer. Классы-обертки остальных примитивных целочисленных типов - Byte, Short, Long - содержат аналогичные методы и константы (определенные для соответствующих типов: byte, short, long).

Рассмотрим пример:

```
public static void main(String[] args) {
    int i = 1;
    byte b = 1;
    String value = "1000";
    Integer iObj = new Integer(i);
    Byte bObj = new Byte(b);
    System.out.println("while i==b is " +
                            (i==b));
    System.out.println("iObj.equals(bObj) is "
                            + iObj.equals(bObj));
    Long lObj = new Long(value);
    System.out.println("lObj = " +
                            lObj.toString());
```

```
    Long sum = new Long(lObj.longValue() +
                        iObj.byteValue() +
                        bObj.shortValue());
    System.out.println("The sum = " +
                       sum.doubleValue());
}
```

В данном примере произвольным образом используются различные варианты классов-оберток и их методов. В результате выполнения на экран будет выведено следующее:

```
while i==b is true
iObj.equals(bObj) is false
lObj = 1000
The sum = 1002.0
```

Оставшиеся классы-обертки числовых типов Float и Double, помимо описанного для целочисленных примитивных типов, дополнительно содержат определения следующих констант:

NEGATIVE_INFINITY — отрицательнаябесконечность;
POSITIVE_INFINITY — положительнаябесконечность;
NaN — нечисловоезначение.

Кроме того, другой смысл имеет значение MIN VALUE - вместо наименьшего значения оно представляет минимальное положительное значение, которое может быть представлено этим примитивным типом.

Кроме классов-оберток для примитивных числовых типов, таковые определены и для остальных примитивных типов Java.

Character

Реализует интерфейсы Comparable и Serializable.

Из конструкторов имеет только один, принимающий char в качестве параметра.

Кроме стандартных методов equals(), hashCode(), toString(), содержит только два нестатических метода:

public char charValue() - возвращает обернутое значение Character; public int compareTo(Character anotherCharacter) - сравнивает обернутые значения char как числа, то есть возвращает значение return this.value - anotherCharacter.value.

Также для совместимости с интерфейсом Comparable метод compareTo() определен с параметром Object:

public int compareTo(Object o) - если переданный объект имеет тип Character, результат будет аналогичен вызову compareTo((Character)o), иначе будет брошено исключение ClassCastException, так как Character можно сравнивать только с Character.

Статических методов в классе Character довольно много, но все они просты и логика их работы понятна из названия. Большинство из них - это методы, принимающие char и проверяющие всевозможные свойства. Например:

public static boolean isDigit(char c) проверяет, является ли char цифрой.

Эти методы возвращают значение истина или ложь, в соответствии с тем, выполнен ли критерий проверки.

Делая краткое заключение по классам-оберткам, можно сказать, что:

каждый примитивный тип имеет соответствующий класс-обертку; все классы-обертки могут быть сконструированы как с использованием примитивных типов, так и с использованием String, за исключением Character, который может быть сконструирован только по char; классы-обертки могут сравниваться с использованием метода equals(); примитивные типы могут быть извлечены из классов-оберток с помощью соответствующего метода

boolean add(E o)	Добавление элемента в коллекцию (возвращает **true** в случае успеха)
boolean remove(Object o)	Удаление элемента из коллекции
boolean contains(Object o)	Проверка принадлежности элемента коллекции
int size()	Получение количества элементов в коллекции
boolean isEmpty()	Проверка коллекции на пустоту
boolean equals(Object o)	Сравнение коллекций на равенство
Iterator<E> iterator()	Получение итератора коллекции
Object[] toArray()	Копирование содержимого коллекции в массив в порядке обхода коллекции итератором

Таблица 10.1. Основные методы интерфейса **Collection<E>**

xxxxValue() (например intValue()); классы-обертки также являются классами-утилитами, т.е. предоставляют набор статических методов для работы с примитивными типами; классы-обертки являются неизменяемыми.

10.2. Общие свойства контейнеров-коллекций. Все классы-коллекции реализуют интерфейс **Collection<E>**. Основные методы этого интерфейса приведены в табл. 10.1. Все они соответствуют операциям, практически не зависящим от типа коллекции. В число наиболее часто используемых коллекций входят динамические массивы, двусвязные списки, множества и упорядоченные множества. Этот интерфейс представляет концепцию хранения данных как последовательности.

При создании класса MyClass переопределяем методы equals(), hashCode(), toString().

10.3. Итераторы. Итераторы предназначены для обхода коллекций в некотором порядке. Использование итератора - это самый быстрый способ перечисления всех элементов коллекции. Классы-итераторы реализуют интерфейс **Iterator<E>**, определяющий три метода: **hasNext()**, **next()** и **remove()**.

Итератор может быть получен путём вызова метода **iterator()** на соответствующем объекте-коллекции. Первоначально итератор устанавливается *перед первым* элементом коллекции. Каждый вызов метода

```
E next()
```

возвращает элемент, перед которым находился итератор до вызова, и сдвигает итератор, помещая его после возвращённого элемента. Если итератор находится после последнего элемента коллекции, метод **next()** выбрасывает исключение типа **NoSuchElementException**. Для проверки наличия элемента перед итератором используется метод

```
boolean hasNext()
```

Для удаления последнего возвращённого методом **next()** элемента может быть использован метод

```
void remove()
```

Допускается лишь однократный вызов метода **remove()** после вызова метода **next()**. В противном случае, а также в случае, когда вызов **next()** не осуществлялся вообще, выбрасывается исключение типа **IllegalStateException**.

В качестве примера рассмотрим использование итератора для вывода содержимого коллекции **collection**, содержащей элементы типа **MyClass**:

```
ArrayList<MyClass> collection = new ArrayList<MyClass>();
for(Iterator<MyClass> it = collection.iterator(); it.hasNext(); )
    System.out.println(it.next());
```

Для коллекций (а также для массивов) существует альтернативный вариант цикла **for**:

```
for([ типданных] переменная: коллекция)оператор
    ;
```

Он осуществляет обход содержимого коллекции в порядке, определяемом итератором (для массивов в порядке от первого элемента к последнему), при этом переменной, заданной в заголовке цикла, последовательно присваиваются ссылки на элементы коллекции. Например:

```
for(MyClass e : collection)
    System.out.println(e);
```

10.4. Упорядочение объектов. Многие задачи обработки контейнеров связаны с размещением объектов в некотором порядке. Для этого необходимо определить для объектов некоторую операцию, которая позволит выяснить, какой из объектов больше или меньше другого. В Java существует два стандартных способа определения такой операции. В первом случае класс, объекты которого подлежат упорядочению, должен реализовывать интерфейс **Comparable<T>**. В качестве параметра **T** должно подставляться имя класса, реализующего данный интерфейс. Например:

```
class MyClass implements Comparable<MyClass>
```

Интерфейс **Comparable<T>** определяет метод

```
int compareTo(T obj)
```

который должен осуществлять сравнение объекта класса, на котором этот объект вызывается (**this**), с переданным данному методу объектом и возвращать значение, меньшее нуля, если объект **this** больше объекта **obj**, равное нулю, если **this.equals(obj) == true** и большее нуля, если объект **obj** больше объекта **this**. Само отношение <<больше>> может быть произвольным и соответствует критерию, по которому производится упорядочение.

Например, если есть класс <<комплексное число>> с двумя вещественными полями **re** и **im** и требуется выполнять упорядочение объектов этого класса по возрастанию или убыванию модулей соответствующих комплексных чисел, операцию **compareTo** можно определить следующим образом:

```
class ComplexNumber implements Comparable<ComplexNumber>
{
    public double re, im;
    public ComplexNumber(double re, double im)
    {
        this.re = re; this.im = im;
    }
    public int compareTo(ComplexNumber c)
    {
        double diff = (re * re + im * im) − (c.re * c.re + c.im * c.im);
        return (int)(Math.signum(diff));
    }
}
```

Способ упорядочения объектов, основанный на определённой указанным образом операции сравнения, называется в Java *естественным упорядочением*. Каждый класс может определить не более одного способа естественного упорядочения.

Другой способ упорядочения заключается в определении внешнего класса (*компаратора*), объекты которого будут осуществлять сравнение объектов требуемого класса. Классы-компараторы должны реализовывать интерфейс **Comparator<T>**, где параметр **T** определяет

void add(**int** index, E element)	Добавление элемента в указанную позицию
E get(**int** index)	Получение элемента с заданным индексом
E set(**int** index, E element)	Изменение элемента с заданным индексом (возвращает старое значение элемента)
E remove(**int** index)	Удаление элемента с заданным индексом (возвращает удалённый элемент)
int indexOf(Object o) **int** lastIndexOf(Object o)	Поиск первого/последнего вхождения элемента (в случае неудачи возвращают --1)
ListIterator<E> listIterator() ListIterator<E> listIterator(**int** index)	Получение списочного итератора, установленного на начало списка/перед заданным элементом списка

Таблица 10.2. Основные методы интерфейса **List<E>**

тип объектов, сравниваемых данным компаратором. Интерфейс **Comparator<T>** определяет метод

```
int compare(T o1, T o2)
```

который осуществляет сравнение двух объектов, передаваемых ему в качестве аргументов, и возвращает в результате сравнения положительное, отрицательное или равное нулю целое число аналогично методу **compare()** интерфейса **Comparable<T>**.

Так, в примере, приведённом выше, вместо метода **compareTo()** можно реализовать компаратор следующим образом:

```
class ComplexNumberLenComp implements Comparator<ComplexNumber>
{
    public int compare(ComplexNumber c1, ComplexNumber c2)
    {
        double diff = (c1.re * c1.re + c1.im * c1.im)
            - (c2.re * c2.re + c2.im * c2.im);
        return (int)(Math.signum(diff));
    }
}
```

Легко видеть, что для одного класса можно определить любое количество компараторов, поэтому данный способ обычно применяется в тех случаях, когда программа использует разные критерии упорядочения объектов.

Многие стандартные алгоритмы обработки контейнеров, а также сами контейнеры требуют определения способа упорядочения объектов. Пример использования различных способов упорядочения при хранении и обработке данных приведён в п. 10.9.

10.5. Списки и динамические массивы. Списки и динамические массивы - это коллекции с произвольным порядком следования элементов. Порядок элементов в таких коллекциях определяется их номерами так же, как в обычных массивах. Все классы, обеспечивающие функциональность этих коллекций, реализуют интерфейс **List<E>** (субинтерфейс интерфейса **Collection<E>**), методы которого приведены в табл. 10.2.

Для рассматриваемых коллекций метод **add()**, определённый в интерфейсе **Collection<E>**, осуществляет вставку в конец коллекции, а метод **remove()** удаляет первое встретившееся вхождение заданного элемента.

Существует две основные разновидности List: **LinkedList<E>**, оптимизированный для последовательного доступа с быстрыми операциями вставки и удаления элемента в середину и медленным доступом к произвольному элементу, **ArrayList<E>**, оптимизированный для быстрого произвольного доступа к элементам.

Класс **LinkedList<E>** представляет собой коллекцию, организованную в виде двусвязного списка. Он обеспечивает получение элемента по индексу за время порядка $O(N)$, а также вставку и удаление элементов в позицию итератора за время порядка $O(1)$.

Для создания экземпляров класса **LinkedList<E>** могут быть использованы следующие конструкторы:

```
LinkedList()
LinkedList(Collection<? extends E> c)
```

Дополнительно к методам, объявленным в интерфейсе **List<E>**, класс **LinkedList<E>** предоставляет методы для вставки, получения и удаления первого и последнего элементов списка:

```
void addFirst(E o)
void addLast(E o)
E getFirst()
E getLast()
E removeFirst()
E removeLast()
```

Все они требуют для своей работы время порядка $O(1)$.

Класс **ArrayList<E>** представляет собой коллекцию, организованную в виде массива переменного размера. Такие массивы имеют некоторый начальный размер (по умолчанию 10), который может увеличиваться при необходимости. Они обеспечивают получение элемента по индексу за время порядка $O(1)$. Вставка и удаление элементов требует времени порядка $O(N)$.

Для создания экземпляров класса **ArrayList<E>** используются следующие конструкторы:

```
ArrayList()
ArrayList(Collection<? extends E> c)
ArrayList(int capacity)
```

Последний конструктор позволяет задать количество памяти, первоначально выделяемое под массив.

Итераторы обходят списки и динамические массивы в порядке увеличения индексов элементов. Коллекции, реализующие интерфейс **List<E>**, поддерживают специальные *списочные итераторы* (интерфейс **ListIterator<E>**), имеющие более широкую функциональность, чем обычные. В частности, такие итераторы могут двигаться по коллекции в обоих направлениях. Для проверки наличия предыдущего элемента и его получения предназначены методы

```
boolean hasPrevious()
E previous()
```

аналогичные методам **hasNext()** и **next()**. Также списочные итераторы позволяют выполнять замену последнего полученного элемента с помощью метода

```
void set(E o)
```

а также вставку элемента в текущую позицию итератора с помощью метода

```
void add(E o)
```

После вставки итератор размещается после вставленного элемента.

Контейнер **LinkedList<E>** обладает более широкими возможностями.

ArrayList это список, реализованный на основе массива, а LinkedList - это классический связный список, основанный на объектах со ссылками между ними.

Преимущества ArrayList: в возможности доступа к произвольному элементу по индексу за постоянное время (так как это массив), минимум накладных расходов при хранении такого списка, вставка в конец списка в среднем производится так же за постоянное время. В среднем потому, что массив имеет определенный начальный размер n (в коде это параметр capacity), по умолчанию n = 10, при записи n+1 элемента, будет создан новый массив размером (n * 3) / 2 + 1, в него будут помещены все элементы из старого массива + новый, добавляемый элемент. В итоге получаем, что при добавлении элемента при необходимости расширения массива, время добавления будет значительно больше, нежели при записи элемента в готовую пустую ячейку. Тем не менее, в среднем время вставки элемента в конец списка является постоянным. Удаление последнего элемента происходит за константное время. Недостатки ArrayList проявляются при вставке/удалении элемента в середине списка - это вызывает перезапись всех элементов размещенных правее в списке на одну позицию влево, кроме того, при удалении элементов размер массива не уменьшается, до явного вызова метода trimToSize().

LinkedList наоборот, за постоянное время может выполнять вставку/удаление элементов в списке (именно вставку и удаление, поиск позиции вставки и удаления сюда не входит). Доступ к произвольному элементу осуществляется за линейное время (но доступ к первому и последнему элементу списка всегда осуществляется за константное время, т.к. ссылки постоянно хранятся на первый и последний, так что добавление элемента в конец списка вовсе не значит, что придется перебирать весь список в поисках последнего элемента). В целом же, LinkedList в абсолютных величинах проигрывает ArrayList и по потребляемой памяти и по скорости выполнения операций. LinkedList предпочтительно применять, когда происходит активная работа (вставка/удаление) с серединой списка или в случаях, когда необходимо гарантированное время добавления элемента в список.

Что вы обычно используете (ArrayList или LinkedList)? Почему? Это вопрос является слегка замаскированной версией предыдущего, так как ответ на этот вопрос приведет к постепенному изложению ответа на предыдущей вопрос. В 90% случае ArrayList будет быстрее и экономичнее LinkedList, так что обычно используют ArrayList, но тем не менее всегда есть 10% случаев для LinkedList. Я говорю, что обычно ArrayList использую, ссылаясь на тесты и последний абзац из предыдущего вопроса, но не забываю и про LinkedList (в каких случаях? так же последний абзац предыдущего вопроса помогает).

Что быстрее работает ArrayList или LinkedList? Правильным же действием будет встречный вопрос о том, какие действия будут выполняться над структурой. Необходимо добавить 1млн. элемент, какую структуру вы используете? Нужно задавать дополнительные вопросы: в какую часть списка происходит добавление элементов? есть ли информация о том, что потом будет происходить с элементами списка? какие то ограничения по памяти или скорости выполнения?

Как происходит удаление элементов из ArrayList? Как меняется в этом случае размер ArrayList? Опять же, ответ на вопрос 1 содержит ответ и на этот вопрос. При удалении произвольного элемента из списка, все элементы находящиеся правее смещаются на одну ячейку влево и реальный размер массива (его емкость, capacity) не изменяется никак. Механизм автоматического расширения массива существует, а вот автоматического сжатия нет, можно только явно выполнить сжатие командой trimToSize().

10.6. Множества и упорядоченные множества. Множества и упорядоченные множества - это коллекции, обеспечивающие быстрое добавление и удаление элементов, однако не позволяющие устанавливать порядок их следования и хранить дубликаты.

Множества представлены интерфейсом **Set<E>**, а также реализующим его классом **HashSet<E>**. Ни один из них не добавляет новых методов по сравнению с интерфейсом

V put(K key, V value)	Добавление пары ключ-значение (возвращает старое значение, ассоциированное с ключом или **null**, если такого ключа не было)
V get(Object key)	Получение значения, соответствующего ключу (возвращает **null**, если ключа нет в ассоциативном массиве)
V remove(Object key)	Удаление пары ключ-значение (возвращает значение удаляемой пары)
boolean containsKey(Object key)	Проверка, содержит ли ассоциативный массив заданный ключ
int size()	Получение размера ассоциативного массива
boolean isEmpty()	Проверка ассоциативного массива на пустоту
void clear()	Очистка ассоциативного массива
boolean equals(Object o)	Сравнение ассоциативных массивов на равенство
Set<K> keySet()	Получение набора ключей в виде множества

Таблица 10.3. Основные методы интерфейса **Map<K, V>**

Collection<E>. Для хранения элементов класс **HashSet<E>** использует хэширование, поэтому объекты, хранимые в коллекциях-множествах, обязательно должны реализовывать методы **equals()** и **hashCode()**. Благодаря хэшированию время выполнения основных операций (добавление, удаление и проверка вхождения элемента в множество) практически не зависит от размера множества. Порядок обхода элементов множества итератором оказывается достаточно произвольным, так как определяется хэш-кодами элементов. Для создания экземпляров класса **HashSet<E>** используются следующие конструкторы:

```
HashSet()
HashSet(Collection<? extends E> c)
```

Упорядоченные множества представлены интерфейсом **SortedSet** и реализующем его классом **TreeSet<E>**. Последний использует для хранения элементов сбалансированные бинарные деревья, что позволяет гарантировать порядок обхода множества итератором в возрастающем порядке. Операции добавления, удаления и проверки вхождения элемента в множество требуют времени порядка $N \log N$.

Для создания экземпляров класса **TreeSet<E>** используются следующие конструкторы:

```
TreeSet()
TreeSet(Collection<? extends E> c)
TreeSet(SortedSet<E> s)
TreeSet(Comparator<? super E> c)
```

Последний из них позволяет создавать множества, упорядоченные в произвольном порядке, определяемом передаваемым в конструктор компаратором (см. п. 10.4). Во всех остальных случаях множества упорядочиваются в естественном порядке. В этом случае классы элементов множества обязаны реализовывать интерфейс **Comparable<T>**.

10.7. Ассоциативные массивы. *Ассоциативные массивы* (также употребляются термины <<словарь>> и <<карта отображений>>) являются более сложными типами данных, чем коллекции. Фактически они представляют собой массивы, использующие вместо индексов произвольные объекты.

Как уже отмечалось, ассоциативные массивы можно рассматривать как массивы, использующие вместо индексов произвольные объекты. Можно считать, что они задают отображение множества *объектов-ключей* во множество *объектов-значений*. Они не являются коллек-

циями в том смысле, что не реализуют интерфейс **Collection<E>**. Его заменяет интерфейс **Map<K, V>**, имеющий два параметра. Первый определяет тип данных ключей ассоциативного массива, а второй — тип его значений. Основные методы интерфейса **Map<K, V>** приведены в табл. 10.3.

Ассоциативные массивы используют для хранения пар значений контейнеры-множества. Поэтому как иерархия, так и свойства тех и других совпадают: классам **HashSet** и **TreeSet** соответствуют классы **HashMap** и **TreeMap** соответственно, а интерфейсу **SortedSet** — интерфейс **SortedMap**. Оценки времени выполнения основных операций для ассоциативных списков такие же, как и для соответствующих им множеств.

Ассоциативные списки не имеют собственных итераторов. Для их обхода используются итераторы множеств их ключей, возвращаемые методом **keySet()**. Следующий фрагмент кода демонстрирует использование ассоциативных массивов:

```
TreeMap<String, String> capitals = new TreeMap<String, String>();
capitals.put("Россия", "Москва");
capitals.put("Франция", "Париж");
capitals.put("Италия", "Лондон"); // неправильно
capitals.put("Италия", "Рим"); // исправлениеошибки
System.out.println("Столица Италии— " + capitals.get("Италия"));
// Выводвсегомассива, отсортированногопоназваниямстран
for(String s : capitals.keySet())
    System.out.println(s + ": " + capitals.get(s));
```

7. Как устроена HashMap?

Это второй из списка самых популярных вопросов по коллекциям. Уж даже не помню был ли случай, когда этот вопрос мне не задавали.

Вкратце, HashMap состоит из ?корзин? (bucket`ов). С технической точки зрения ?корзины? ? это элементы массива, которые хранят ссылки на списки элементов. При добавлении новой пары ключ-значение, вычисляет хеш-код ключа, на основании которого вычисляется номер корзины (номер ячейки массива), в которую попадет новый элемент. Если корзина пустая, то в нее сохраняется ссылка на вновь добавляемый элемент, если же там уже есть элемент, то происходит последовательный переход по ссылкам между элементами в цепочке, в поисках последнего элемента, от которого и ставится ссылка на вновь добавленный элемент. Если в списке был найден элемент с таким же ключом, то он заменяется. Добавление, поиск и удаление элементов выполняется за константное время. Вроде все здорово, с одной оговоркой, хеш-функций должна равномерно распределять элементы по корзинам, в этом случае временная сложность для этих 3 операций будет не ниже lg N, а в среднем случае как раз константное время.

В целом, этого ответа вполне хватит на поставленный вопрос, дальше скорее всего завяжется диалог по HashMap, с углубленным пониманием процессов и тонкостей.

Опять же, рекомендую к прочтению статью tarzan82 по HashMap.

8. Какое начальное количество корзин в HashMap?

Довольно неожиданный вопрос, опять же меня он когда-то заставил угадывать число корзин при использовании конструктора по умолчанию.

Ответ здесь ? 16. Отвечая, стоит заметить, что можно используя конструкторы с параметрами: через параметр capacity задавать свое начальное количество корзин.

9. Какая оценка временной сложности выборки элемента из HashMap? Гарантирует ли HashMap указанную сложность выборки элемента?

Ответ на первую часть вопроса, можно найти в ответе на вопрос 7 ? константное время необходимо для выборки элемента. Вот на второй части вопроса, я недавно растерялся. И

устройство HashMap знал и про хеш-функцию тоже знал, а вот к такому вопросу не был готов, в уме кинулся вообще в другом направлении и сосредоточился на строении HashMap откинув проблему хеш-кода, который в голове всегда привык считать хеш-кодом с равномерным распределением. На самом деле ответ довольно простой и следует из ответа вопроса 7.

Если вы возьмете хеш-функцию, которая постоянно будет возвращать одно и то же значение, то HashMap превратится в связный список, с отвратной производительностью. Затем даже, если вы будете использовать хеш-функцию с равномерным распределением, в предельном случае гарантироваться будет только временная сложность lg N. Так что, ответ на вторую часть вопроса ? нет, не гарантируется.

10. Роль equals и hashCode в HashMap?

Ответ на этот вопрос следует из ответа на вопрос 7, хотя явно там и не прописан. hashCode позволяет определить корзину для поиска элемента, а equals используется для сравнения ключей элементов в списке внутри корзины и искомого ключа.

11. Максимальное число значений hashCode()?

Здесь все довольно просто, достаточно вспомнить сигнатуру метода: int hashCode(). То есть число значений равно диапазону типа int ? 2 в 32 (точного диапазона никогда не спрашивали, хватало такого ответа).

12. Как и когда происходит увеличение количества корзин в HashMap?

Вот это довольно тонкий вопрос. Как показал мой мини-опрос, если суть устройства HashMap себе представляют многие более-менее ясно, то этот вопрос часто ставил собеседника в тупик.

Помимо capacity в HashMap есть еще параметр loadFactor, на основании которого, вычисляется предельное количество занятых корзин (capacity*loadFactor). По умолчанию loadFactor = 0,75. По достижению предельного значения, число корзин увеличивается в 2 раза. Для всех хранимых элементов вычисляется новое ?местоположение? с учетом нового числа корзин.

13. В каком случае может быть потерян элемент в HashMap?

Этот интересный вопрос мне прислал LeoCcoder, у меня подобного не спрашивали и честно признаюсь, после прочтения сходу не смог придумать сценарий для потери элемента. Все опять же оказалось довольно просто, хоть и не так явно: допустим в качестве ключа используется не примитив, а объект с несколькими полями. После добавления элемента в HashMap у объекта, который выступает в качестве ключа, изменяют одно поле, которое участвует в вычислении хеш-кода. В результате при попытке найти данный элемент по исходному ключу, будет происходить обращение к правильной корзине, а вот equals (ведь equals и hashCode должны работать с одним и тем же набором полей) уже не найдет указанный ключ в списке элементов. Тем не менее, даже если equals реализован таким образом, что изменение данного поля объекта не влияет на результат, то после увеличения размера корзин и пересчета хеш-кодов элементов, указанный элемент, с измененным значением поля, с большой долей вероятности попадет совсем в другую корзину и тогда он уже совсем потеряется.

10.8. Унаследованные (legacy) классы-контейнеры. Кроме классов **ArrayList<E>**, **HashSet<E>** и **HashMap<E>**, существуют близкие им по функциональности классы **Vector<E>** и **Hashtable<E>** и **Dictionary<E>**. Это так называемые унаследованные (legacy) классы из старых версий библиотеки. Основное их назначение — обеспечивать обратную совместимость. Хотя они не считаются устаревшими (deprecated), вероятно, лучше избегать использования их в новых программах.

10.9. Стандартные алгоритмы обработки контейнеров. Стандартная библиотека Java содержит реализацию некоторых стандартных алгоритмов обработки контейнеров, а также

E max(Collection<? **extends** E> coll) E max(Collection<? **extends** T> coll, Comparator<? **super** T> comp) E min(Collection<? **extends** E> coll) E min(Collection<? **extends** T> coll, Comparator<? **super** T> comp)	Нахождение наибольшего/наименьшего элемента коллекции в смысле естественного порядка элементов/порядка элементов, определяемого компаратором
void sort(List<T> list) **void** sort(List<T> list, Comparator<? **super** T> c) **int** binarySearch(List<? **extends** T> list, T key) **int** binarySearch(List<? **extends** T> list, T key, Comparator<? **super** T> c)	Сортировка коллекции с произвольным порядком следования элементов в естественном порядке/в порядке, определяемом компаратором Бинарный поиск индекса элемента в упорядоченной коллекции. Если элемент не найден, возвращается величина, равная *(минус позиция, в которой должен был быть элемент, минус единица)*
Collection<T> unmodifiableCollection(Collection<? **extends** T> c) List<T> unmodifiableList(List<? **extends** T> c) и т. д.	Получение неизменяемой обертки коллекции (существуют аналогичные методы для всех стандартных типов контейнеров)

Таблица 10.4. Основные методы класса **Collections**. Все методы являются статическими

void sort(**int**[] a)	Сортировка массива
void sort(T[] a, Comparator<? **super** T> c)	Сортировка массива объектов в порядке, определяемом компаратором
int binarySearch(**int**[] a, **int** key) **int** binarySearch(T[] a, T key, Comparator<? **super** T> c)	Бинарный поиск индекса элемента в упорядоченном массиве. Если элемент не найден, возвращается величина, равная *(минус позиция, в которой должен был быть элемент, минус единица)*
boolean equals(**int**[] a, **int**[] a2)	Сравнение массивов на равенство
void fill(**int**[] a, **int** val)	Заполнение всех элементов массива заданным значением
List<T> asList(T[] a) List<T> asList(T. . . a)	Преобразование массива/набора элементов в список

Таблица 10.5. Основные методы класса **Arrays**. Приведены методы для работы с массивами типа **int[]**. Существуют аналогичные методы для работы с массивами всех остальных примитивных типов и типа **Object**. Все методы являются статическими

некоторых сервисных операций. Все они представлены в виде статических методов классов **Collections** (для коллекций и ассоциативных списков) и **Arrays** (для массивов). Соответствующие методы приведены в табл. 10.4, 10.5.

Особо отметим семейство методов, возвращающих неизменяемые обёртки контейнеров (**unmodifiableCollection()**, **unmodifiableList()** и т. д.). Эти методы возвращают объект, реализующий интерфейс контейнера соответствующего типа. При использовании методов перемещения и получения значений возвращённого объекта последний ведёт себя точно так же, как и исходный объект. При попытке вызова любого из методов изменения контейнера выбрасывается исключение типа **UnsupportedOperationException**.

11. Графические приложения

11.1. Первое приложение JavaFX 8.0. Документация
http://docs.oracle.com/javase/8/javase-clienttechnologies.htm
http://docs.oracle.com/javase/8/javafx/api/toc.htm
Создадим приложение JavaFX. В меню File выберите New Project. В категории JavaFX выберите JavaFX Application, далее Назовите проект HelloWorld, Готово.

Netbeans создаст проект и файл HelloWorld.java, который будет содержать простую JavaFX программу.

```
 package helloworld;

import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;

public class HelloWorld extends Application {
@Override
    public void start(Stage primaryStage) {
        Button btn = new Button();
        btn.setText("Say 'Hello World'");
        btn.setOnAction(new EventHandler<ActionEvent>() {

            @Override
            public void handle(ActionEvent event) {
                System.out.println("Hello World!");
            }
        });

        StackPane root = new StackPane();
        root.getChildren().add(btn);

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

        primaryStage.setTitle("Hello World!");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

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

Рассмотрим основные компоненты структуры JavaFX приложения.

Главный класс JavaFX приложения унаследован от javafx.application.Application. Класс Application обеспечивает жизненный цикл приложения. Метод launch() - запуск, графическое приложение запускается как отдельный процесс в операционной системе. В нем происходит обращение к методу start(). Метод start() — главный метод приложения, содержит основные действия по работе приложения.

В методе main, где находится точка входа в программу, вызывается метод launch(), он в свою очередь, является точкой входа в FXприложение. Метод start() вызывается при создании потока приложения, в его параметрах можно увидеть объект класса Scene. Этот класс связан с экземпляром окна, которое будет видеть пользователь. Stage имеет большой набор методов.

В JavaFX приложении присутствуют два главных компонента Stage и Scene. Stage - окно приложения, основной графический контейнер, методы позволяют определить свойства окна.

Scene - сцена, контейнер для видимых элементов. В примере мы создаем сцену с определенными размерами и делаем ее видимой.

Программа JavaFX имеет иерархическую структуру в виде дерева, где узлы — элементы программы (кнопка, текст и т.д.). В нашем случае корневой узел — объект StackPane — панель с изменяемым размером, это означает, что размер окна программы можно будет изменять.

Наш корневой узел содержит одного потомка — кнопку с обработчиком нажатия (для вывода сообщения в консоль).

JavaFX Node является одним из основных базовых классов для всех узлов графа сцены. Возможности : масштабирование, преобразования, сдвиг, эффекты. Некоторые из наиболее часто используемых узлов являются элементами управления и объекты Shape. Структура, граф сцены содержит дочерние узлы с помощью классов контейнеров, таких как группа или Панель.

Для того, чтобы создать сцену, нужно указать главного родителя(root), куда можно будет добавлять элементы.

Еще есть много контейнеров, некоторые представлены ниже.

HBox(горизонтальная очередь элементов) VBox(вертикальная очередь элементов) FlowPane(многострочная горизонтальная очередь элементов) BorderPane(контейнер с 5 местами, сверху, снизу, слева, справа и по центру) GridPane(сетка элементов) ScrollPane(контейнер с возможностью прокрутки)

Главным родителем может быть экземпляр класса Group, он наследуется от класса Parent, поэтому может содержать в себе другие элементы.

Он представляет собой NullLayoutиз Swing, то есть, какие координаты мы зададим элементу, то там он и будет рисоваться.

Расположение элементов

```
 package javafxapplication1;

import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;
import javafx.scene.control.Label;
import javafx.scene.layout.FlowPane;
import javafx.geometry.Orientation;
```

```
import javafx.scene.Group;

public class JavaFXApplication1 extends Application {

    @Override
    public void start(Stage primaryStage) {
        Label label = new Label("LABEL");
        label.setLayoutX(20);
        label.setLayoutY(20);
        Button btn = new Button();
        btn.setLayoutX(20);
        btn.setLayoutY(50);
        btn.setText("Say 'Hello World'");
        btn.setOnAction(new EventHandler<ActionEvent>() {

            @Override
            public void handle(ActionEvent event) {
                System.out.println("Hello World!");
            }
        });

        //StackPane root = new StackPane();
        //FlowPane root = new FlowPane(Orientation.VERTICAL,10,10);
        Group root = new Group();
        root.getChildren().add(btn);
        root.getChildren().add(label);

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

        primaryStage.setTitle("Hello World!");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }

}
```

Внешний вид элементов, размещение изображений

```
package javafxapplication1;

import java.io.IOException;
import javafx.application.Application;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.layout.StackPane;
import javafx.stage.Stage;
import javafx.scene.control.Label;
```

```
import javafx.scene.layout.FlowPane;
import javafx.geometry.Orientation;
import javafx.scene.Group;
import javafx.scene.image.Image;
import javafx.scene.image.ImageView;
import java.io.File;
import javafx.scene.text.Font;
import javafx.scene.text.FontWeight;
import javafx.scene.layout.*;
import javafx.scene.paint.Color;
import javafx.geometry.Insets;

public class JavaFXApplication1 extends Application {

    @Override
    public void start(Stage primaryStage) {
        Label label = new Label("LABEL");
        label.setLayoutX(20);
        label.setLayoutY(20);
        label.setPrefSize(200, 20);
        label.setFont(Font.font("Tahoma", FontWeight.NORMAL, 20));
        label.setBackground(new Background(new BackgroundFill(Color.AQUA,
CornerRadii.EMPTY, Insets.EMPTY)));
        /*try {
          File file = new File("images/smiley.jpeg");
          String localUrl = file.toURI().toURL().toString();
          Image image = new Image(localUrl);
          ImageView imv = new ImageView(image);
          label.setGraphic(imv);
          } catch (IOException e){System.out.println("Error img");}*/
        Button btn = new Button();
        btn.setLayoutX(20);
        btn.setLayoutY(250);
        btn.setText("Say 'Hello World'");
        btn.setOnAction(new EventHandler<ActionEvent>() {

            @Override
            public void handle(ActionEvent event) {
                //System.out.println("Hello World!");
                try {
                    File file = new File("images/smiley.jpeg");
                    String localUrl = file.toURI().toURL().toString();
                    Image image = new Image(localUrl);
                    ImageView imv = new ImageView(image);
                    label.setGraphic(imv);
                } catch (IOException e) {
                    System.out.println("Error img");
                }
            }
        });

        //StackPane root = new StackPane();
```

```
        //FlowPane root = new FlowPane(Orientation.VERTICAL,10,10);
        Group root = new Group();
        root.getChildren().add(btn);
        root.getChildren().add(label);

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

        primaryStage.setTitle("Hello World!");
        primaryStage.setScene(scene);
        primaryStage.show();
    }

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        launch(args);
    }

}
```

Canvas (холст) - элемент для рисования, требуется определить размер.

GraphicsContext - Этот класс используется для рисования на холсте с помощью буфера, а также обработка событий перерисовки окна.

Сначала все рисуемые элементы помещаются в буфер, затем на холст.

Холст содержит только одир GraphicsContext, и только один буфер. После того, как холст подключен к сцене, он может быть изменен JavaFX приложением.

gc.setFill(Color.GREEN); Устанавливает текущий атрибут кисть (заполнения краской). Значение по умолчанию черный.

gc.setStroke(Color.BLUE); stroke (штрих)

gc.strokeRoundRect(160, 60, 30, 30, 10, 10); x - координата X в верхнем левом углу, границы овала. y - Y координата верхнего левого угла границы овала. W - ширина в центре овала. H - высота в центре овала. arcWidth - ширина дуги углов прямоугольника. arcHeight - высота дуги углов прямоугольника.

gc.fillPolygon(new double[]10, 40, 10, 40, new double[]210, 210, 240, 240, 4); xPoints - array containing the x coordinates of the polygon's points or null. yPoints - array containing the y coordinates of the polygon's points or null. nPoints - the number of points that make the polygon.

GridPane

grid.add(scenetitle, 0, 0, 2, 1); child - the node being added to the gridpane columnIndex - the column index position for the child within the gridpane, counting from 0 rowIndex - the row index position for the child within the gridpane, counting from 0 colspan - the number of columns the child's layout area should span rowspan - the number of rows the child's layout area should span

```
 String year = valueYear.getText();
                    Calendar calendar = Calendar.getInstance(TimeZone.getDefault(),
Locale.getDefault());
                    calendar.setTime(new Date());
                    int age=calendar.get(Calendar.YEAR) − Integer.parseInt(year);
                    actiontarget.setFill(Color.FIREBRICK);
                    actiontarget.setText(nameTextField.getText()+
                    " was "+Integer.toString(age)+" years old");
```

public static Calendar getInstance(TimeZone zone, Locale aLocale) зона - часовой пояс, aLocale - стандарт для недели, представляет определенную географическую, политическую, или культурную область (локализация).

public static Locale getDefault()

Gets the current value of the default locale for this instance of the Java Virtual Machine.

The Java Virtual Machine sets the default locale during startup based on the host environment. It is used by many locale-sensitive methods if no locale is explicitly specified. It can be changed using the setDefault method.

Returns: the default locale for this instance of the Java Virtual Machine

11.2. Диалоги. java 8

java.util

Class Optional<T>

Контейнер. A container object which may or may not contain a non-null value. If a value is present, isPresent() will return true and get() will return the value.

javafx.stage

Class FileChooser

public File showOpenDialog(Window ownerWindow)

Window ownerWindow устанавливаеися блокировка окна-владельца, пока диалог не будет закрыт,

public List<File> showOpenMultipleDialog(Window ownerWindow)

11.3. Классы, реализующие MVC. В JavaFX предпочтительно использовать Properties для всех полей класса-модели. Property позволяет нам получать автоматические уведомления при любых изменениях переменных, таких как lastName или других.

Объекты инициируют события при изменении.

11.4. Лямбда-выражения. Функционалъным называется такой интерфейс, которы й содержит один и только один абстрактный метод. Лямбда-выражение, по существу, является анонимным (т.е. безымянным) методом . Но этот метод не выполняется самостоятельно, а служит для реализации метода, определяемого в функциональном интерфейсе. Таким образом , лямбда-выражение приводит к некоторой форме анонимного класса.

Лямбда-выражение вносит новый элемент в синтаксис и оператор в язык Java. Этот новый оператор называется лямбда-оператором, или операцией "стрелка"(->). Он разделяет лямбда-выражение на две части. В левой части указываются любые параметры, требующиеся в лямбда-выражении. (Если же параметры не требуются , то они указываются пустым списком .) А в правой части находится тело лямбда-выражения, где указываются действия, выполняемые лямбда-выражением.

В Java определены две разновидности тел лямбда-выражений. Одна из них состоит из единственного выражения, а другая - из блока кода.

Timer timer = new Timer(); timer.scheduleAtFixedRate(new TimerTask() @Override public void run() //System.out.print("I would be called every 2 seconds"); n++; if (n > 10) timer.cancel(); //numberTextField.setText(Integer.toString(n)); , 0, 1000);