Министерство образования Республики Беларусь

Учреждение образования
«Полоцкий государственный университет»

С. Г. Сурто

# КРОСС-ПЛАТФОРМЕННОЕ ПРОГРАММИРОВАНИЕ

Методические указания к лабораторным работам
для студентов специальности 1-40 02 01
«Вычислительные машины, системы и сети»

*Текстовое электронное издание*

Новополоцк
Полоцкий государственный университет
2022

Об издании – 1, 2

1 – дополнительный титульный экран – сведения об издании

УДК 004.432
ББК 32.973.3

Одобрено и рекомендовано к изданию методической комиссией
факультета информационных технологий (протокол № 11 от 27.12.2021)

Кафедра вычислительных систем и сетей

РЕЦЕНЗЕНТЫ:

канд. техн. наук, доц., техн. директор ООО «ТриИнком»
К. Я. РАХАНОВ
ст. преподаватель каф. технологий программирования
Полоцкого государственного университета
В. С. РОГУЛЕВ

© Сурто С. Г., 2022
© Полоцкий государственный университет, 2022

Для создания текстового электронного издания «Кросс-платформенное программирование» использованы текстовый процессор Microsoft Word и программа Adobe Acrobat XI Pro для создания и просмотра электронных публикаций в формате PDF.

Сергей Геннадьевич СУРТО

**КРОСС-ПЛАТФОРМЕННОЕ ПРОГРАММИРОВАНИЕ**

Методические указания к лабораторным работам
для студентов специальности 1-40 02 01
«Вычислительные машины, системы и сети»

Редактор *А. А. Прадидова*

Подписано к использованию 01.06.2022.
Объем издания: 4,81 Мб. Заказ 374.

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

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

ЛП № 02330/278 от 08.05.2014.

211440, ул. Блохина, 29,
г. Новополоцк,
Тел. 8 (0214) 59-95-41, 59-95-44
http://www.psu.by

СОДЕРЖАНИЕ

ВВЕДЕНИЕ ........ 6
Лабораторная работа № 1. Введение в программирование на языке Java в среде NetBeans ........ 7
Теоретическая часть ........ 7
Основы языка программирования Java ........ 18
ЗАДАНИЕ К ЛАБОРАТОРНОЙ РАБОТЕ ........ 38
КОНТРОЛЬНЫЕ ВОПРОСЫ ........ 39
СОДЕРЖАНИЕ ОТЧЕТА ........ 44
Порядок защиты лабораторной работы ........ 44
Лабораторная работа № 2. Классы и объекты в Java ........ 44
Теоретическая часть ........ 44
ИНДИВИДУАЛЬНЫЕ ВАРИАНТЫ ЗАДАНИЙ ........ 58
ЗАДАНИЕ К ЛАБОРАТОРНОЙ РАБОТЕ ........ 59
КОНТРОЛЬНЫЕ ВОПРОСЫ ........ 60
СОДЕРЖАНИЕ ОТЧЕТА ........ 60
Порядок защиты лабораторной работы ........ 60
Лабораторная работа № 3. Наследование ........ 61
Теоретическая часть ........ 61
ИНДИВИДУАЛЬНЫЕ ВАРИАНТЫ ЗАДАНИЙ ........ 74
ЗАДАНИЕ К ЛАБОРАТОРНОЙ РАБОТЕ ........ 75
КОНТРОЛЬНЫЕ ВОПРОСЫ ........ 75
СОДЕРЖАНИЕ ОТЧЕТА ........ 76
Порядок защиты лабораторной работы ........ 76
Лабораторная работа № 4. Классы-обертки ........ 76
Теоретическая часть ........ 76
ЗАДАНИЕ К ЛАБОРАТОРНОЙ РАБОТЕ ........ 95
КОНТРОЛЬНЫЕ ВОПРОСЫ ........ 95
СОДЕРЖАНИЕ ОТЧЕТА ........ 96
Порядок защиты лабораторной работы ........ 96
Лабораторная работа № 5. Обработка исключений, работа со строками ........ 96
Теоретическая часть ........ 96
ЗАДАНИЕ К ЛАБОРАТОРНОЙ РАБОТЕ ........ 118
КОНТРОЛЬНЫЕ ВОПРОСЫ ........ 120
СОДЕРЖАНИЕ ОТЧЕТА ........ 120
Порядок защиты лабораторной работы ........ 121
Лабораторная работа № 6. Обобщенные классы в Java, коллекции ........ 121
Теоретическая часть ........ 121
Обобщения и заменяемость ........ 125
Классы-коллекции ........ 137
ЗАДАНИЕ К ЛАБОРАТОРНОЙ РАБОТЕ ........ 146
КОНТРОЛЬНЫЕ ВОПРОСЫ ........ 146
СОДЕРЖАНИЕ ОТЧЕТА ........ 147
Порядок защиты лабораторной работы ........ 147

Лабораторная работа № 7. Потоки ввода/вывода ........ 147
Теоретическая часть ........ 147
Чтение и запись файлов ........ 158
Буферизуемые потоки ........ 162
Буферизируемые символьные потоки ........ 169
ЗАДАНИЕ К ЛАБОРАТОРНОЙ РАБОТЕ ........ 174
КОНТРОЛЬНЫЕ ВОПРОСЫ ........ 174
СОДЕРЖАНИЕ ОТЧЕТА ........ 175
Порядок защиты лабораторной работы ........ 175
Лабораторная работа № 8. Построение кросс-платформенных графических интерфейсов ........ 176
Теоретическая часть ........ 176
ИНДИВИДУАЛЬНЫЕ ВАРИАНТЫ ЗАДАНИЙ ........ 193
ЗАДАНИЕ К ЛАБОРАТОРНОЙ РАБОТЕ ........ 193
КОНТРОЛЬНЫЕ ВОПРОСЫ ........ 193
СОДЕРЖАНИЕ ОТЧЕТА ........ 194
Порядок защиты лабораторной работы ........ 194
ЛИТЕРАТУРА ........ 195

# ВВЕДЕНИЕ

Целью настоящих методических указаний является изучение современных технологий программирования для различных архитектур и платформ.

Достижение поставленной цели предполагает решение следующих задач:

– сформировать системное базовое представление, первичные знания, умения и навыки по основам кросс-платформенного программирования для платформы Java;

– изучить этапы создания приложений в интегрированных средах разработки;

– показать основные характеристики исполняемого кода на различных платформах;

– закрепить навыки объектно-ориентированного программирования, используя язык Java как один из лучших языков, имплементирующих эту парадигму.

## Лабораторная работа № 1. Введение в программирование на языке Java в среде NetBeans

**Цель работы.** Ознакомиться с основными типами данных и операторами работы с данными в языке программирования Java. Познакомиться со средой программирования NetBeans IDE.

### Теоретическая часть

#### Установка Java и NetBeansIDE

Для работы программ на языке Java на целевой машине должна быть установлена JRE (JavaRuntimeEnvironment). JRE представляет собой минимальную реализацию виртуальной машины, а также библиотеку классов. Поэтому, если мы хотим запускать программы, то нам надо установить JRE. Для каждой конкретной платформы имеется своя версия JRE.

Однако, так как мы собираемся не только запускать программы, но и разрабатывать их, нам потребуется специальный комплект для разработки JDK (JavaDevelopmentKit). JDK уже содержит JRE, а также включает ряд дополнительных программ и утилит, в частности, компилятор Java – **javac**.

Есть несколько типов платформ Java. Базовую функциональность обеспечивает стандартная версия языка Java – SE (StandardEdition). Она предназначена для создания небольших приложений в масштабах малого предприятия. Кроме того, существует платформа Java EE (EnterpriseEdition), которая нацелена на создание более сложных приложений и в комплект которой входит веб-сервер Glassfish.

Для наших целей будет достаточно Java SE, поэтому мы можем загрузить и установить соответствующую версию JDK с официального сайта Oracle: http://www.oracle.com/technetwork/java/javase/downloads/index.html.

Название релиза JDK, как правило, отражает его версию и версию его обновления. Например, на текущий момент доступен пакет с названием *Java SE 8v5*, где 8 обозначает 8-ю версию Java, а 5 – версию обновления. Поскольку команда Oracle регулярно выпускает новые обновления, то в нашем случае версия обновления может отличаться. В этом ничего страшного нет, главное, чтобы версия самого языка была 8.

**IDE NetBeans** позволяет быстро и легко разрабатывать настольные, мобильные и веб-приложения Java, а также приложения HTML5 с использованием технологий HTML, JavaScript и CSS. IDE также предоставляет мно-

гофункциональные наборы средств для разработчиков PHP и C/C++. Это бесплатное программное обеспечение с открытым исходным кодом, которое имеет большое сообщество пользователей и разработчиков по всему миру.

По сравнению с другими средами IDE среда IDE NetBeans обеспечивает высококлассную комплексную поддержку новейших технологий Java и последних усовершенствований стандартов Java. Это первая бесплатная среда IDE, поддерживающая JDK 8, JDK 7, Java EE 7, включая соответствующие усовершенствования HTML5 и JavaFX 2.

Скачать и IDE NetBeans можно по адресу: https://netbeans.org/downloads/.

Помимо всего прочего, среду NetBeans можно установить вместе с последней версией JDK с официального сайта Oracle (рисунок 1).

**Рисунок 1. – Среда NetBeans**

Итак, после установки JDK создадим первую программу на Java в среде NetBeans.

**Запускаем среду NetBeans** (рисунок 2).

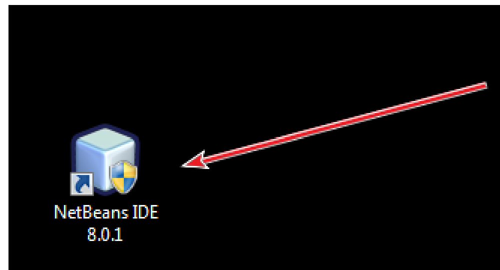


**Рисунок 2. – Запуск среды NetBeans**

Открывается главное окно среды (рисунок 3).

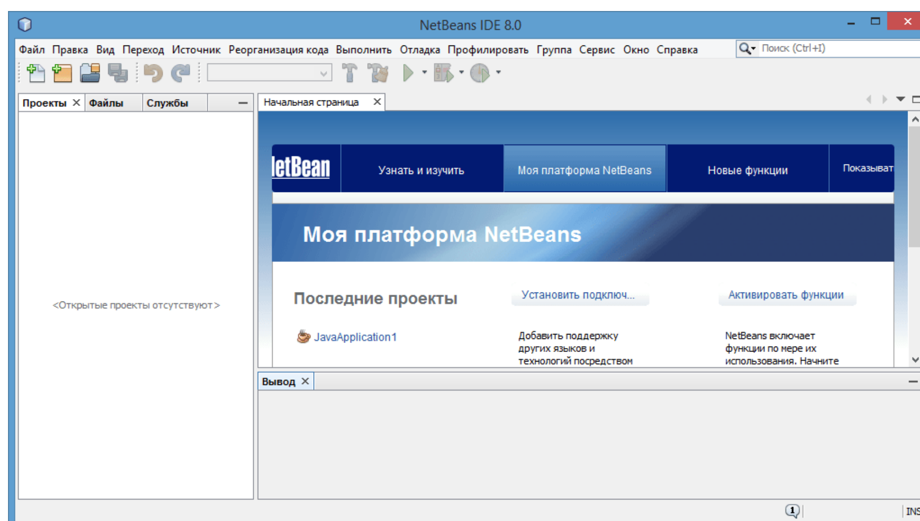


**Рисунок 3. – Главное окно среды**

NetBeans интуитивно понятна и с ней очень легко работать. Создадим новый проект. Для этого выберем в меню пункт **Файл > Создать проект**. После этого перед нами откроется окно создания нового проекта (рисунок 4).

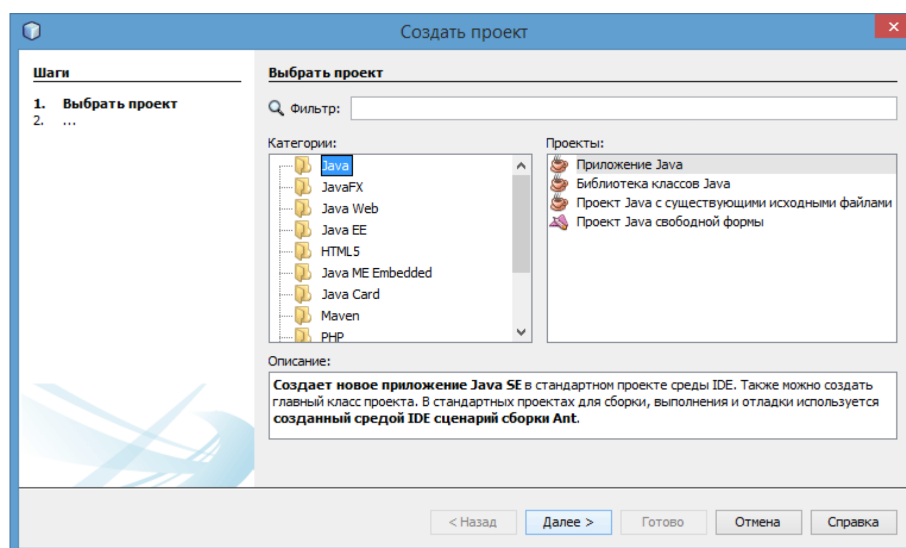


**Рисунок 4. – Создание проекта**

В окне создания нового проекта выберем в левой колонке первый пункт – Java, а в правой в качестве типа проекта – **Приложение Java**. И нажмем кнопку **Далее >**.

Затем откроется окно настроек проекта (рисунок 5).

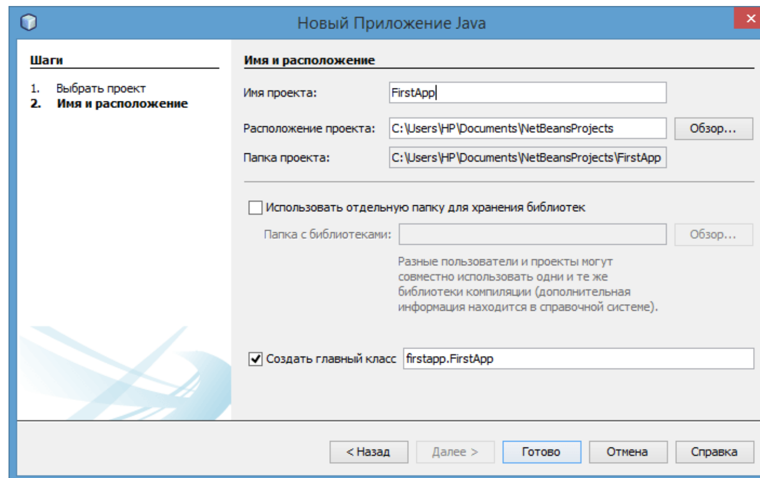


**Рисунок 5. – Настройка нового проекта**

Здесь дадим проекту какое-нибудь название (допустим, это FirstApp). Для всех остальных полей можно оставить значения по умолчанию. Последнее поле **Создать главный класс** указывает, что автоматически в проекте будет создан класс программы FirstApp, который будет находиться в одноименном пакете firstapp.

### Соглашения о присвоении имен пакетов

Имена пакетов пишутся во всем нижнем регистре, чтобы избежать конфликта с именами классов или интерфейсов. Компании используют свое инвертированное имя Интернет-домена в качестве имени пакета, например, com.example.mypackage для названного пакета mypackage, создаваемого программистом в домене example.com.

Коллизии имени, которые происходят в пределах единственной компании, должны быть обработаны условно в пределах той компании, возможно включением области или названия проекта после названия компании (например, com.example.region.mypackage). Пакеты на языке Java начинаются с **java**. или **javax**.

Таким образом, пакет верхнего уровня всегда записывается ASCII-буквами в нижнем регистре и может иметь одно из следующих имен:

– трехбуквенные com, edu, gov, mil, net, org, int (этот список расширяется);

– двухбуквенные, обозначающие имена стран, такие как ru, su, de, uk и другие.

Если имя сайта противоречит требованиям к идентификаторам Java, то можно предпринять следующие шаги:

– если в имени стоит запрещенный символ, например, тире, то его можно заменить знаком подчеркивания;

– если имя совпадает с зарезервированным словом, можно в конце добавить знак подчеркивания;

– если имя начинается с цифры, можно в начале добавить знак подчеркивания.

Примеры имен пакетов, составленных по таким правилам:

com.sun.image.codec.jpeg;

org.omg.CORBA.ORBPackage;

oracle.jdbc.driver.OracleDriver.

В данной лабораторной работе целесообразно именовать базовый пакет, например, следующим образом:

by.psu.fit.cmsn.cross_pp.ivanov.lab1.

Это означает, что написанная программа принадлежит организации **PSU** (**PolotskStateUniversity**) в доменной зоне **by**, факультету информационных технологий (**FacultyofInformationTechnologies**) (**fit**), кафедре вычислительных систем и сетей (**ComputingMachinery, SystemsandNetworks**) (**cmsn**), предмету кросс-платформенное программирование (**Cross-PlatformProgramming**) (**cross_pp**), студенту Иванову (**ivanov**), и выполняется первая лабораторная работа (**lab1**).

В завершении создания проекта нажмем на кнопку **Готово**. Перед нами откроется новый проект в Netbeans (рисунок 6).

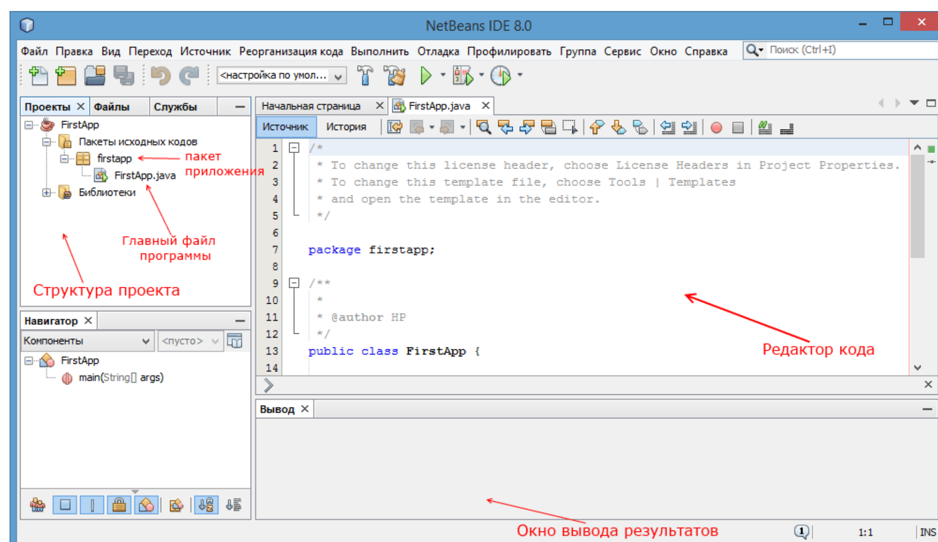


**Рисунок 6. – Редактор исходного кода**

**Первая программа на Java**

Рассмотрим исходный код нашей первой программы подробно (рисунок 7).

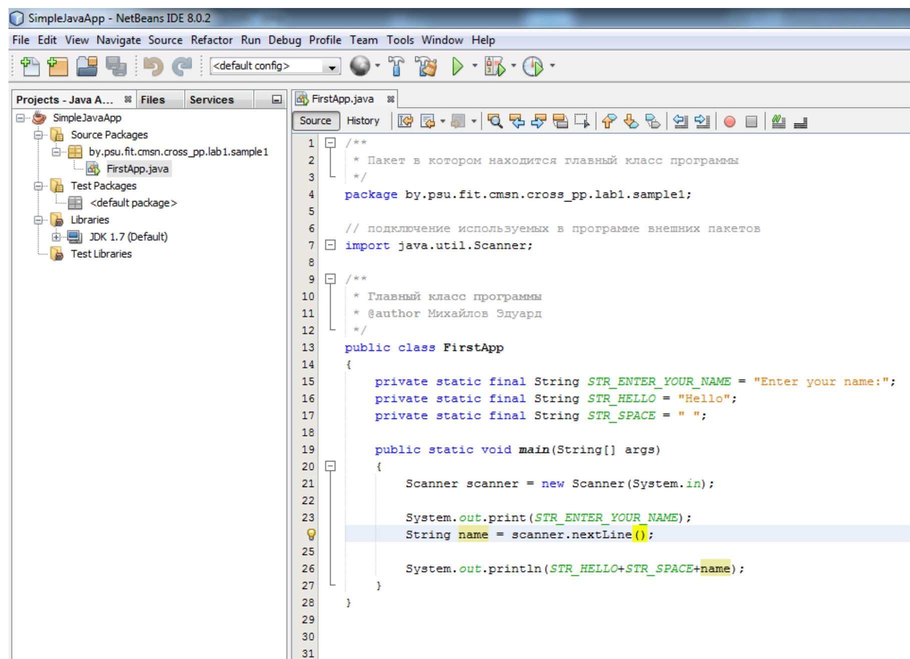


**Рисунок 7. – Исходный код**

В нашей первой программе нам предлагается ввести свое имя. На выходе программа выводит строку «Hello» и через пробел – введенное имя.

В начале файла находится секция с объявлением текущего пакета, которому принадлежит текущий класс.

Далее идет секция с подключенными внешними пакетами с помощью директивы **import**, после которой идут названия подключаемых пакетов и классов. Пакеты представляют собой организацию классов и интерфейсов в общие группы или блоки.

После этого идет многострочный комментарий /** ... */, предназначенный для автоматического создания документации по классу. В нем присутствует инструкция задания метаданных с помощью выражения **@author** – информация об авторе проекта для утилиты создания документации **javadoc**.

**Метаданные** – это некая информация, которая не относится к работе программы и не включается в нее при компиляции, но сопровождает программу и может быть использована другими программами для проверки прав на доступ к ней или ее распространения, проверки совместимости с другими программами,указания параметров для запуска класса и т.п.

В данном месте исходного кода имя – **User** – изначально берется средой разработки из операционной системы по имени папки пользователя.

Так как язык Java имеет си-подобный синтаксис, каждая строка завершается точкой с запятой, а каждый блок кода помещается в фигурные скобки.

Далее идет определение класса программы. Классы объявляются следующим способом: сначала идет модификатор доступа **public**, который указывает, что данный класс будет доступен всем, то есть мы сможем его запустить из командной строки. Далее идет ключевое слово **class**, а потом название класса, и далее блок самого класса в фигурных скобках.

Классы являются теми кирпичиками, из которых состоит программа на Java. Особо следует остановиться на именах классов. Имена классов, а также их методов и переменных, еще называют **идентификаторами**. Идентификаторы представляют произвольную последовательность алфавитных и цифровых символов, а также символа подчеркивания, однако при этом названия не должны начинаться с цифры. Кроме того, идентификаторы не должны представлять зарезервированные ключевые слова, например, такие как class или int и т.д.

Класс может содержать различные переменные и методы. В данном случае у нас объявлен один метод main и три строковых константы, которые понадобятся для работы нашей программы. Как и во многих других си-подобных языках в программе на Java метод **main** является входной точкой программы, с него начинается все управление. Он обязательно должен присутствовать в программе.

В языке Java часто необходимы константы, доступные нескольким методам внутри одного класса. Обычно они называются **константами класса** (classconstants). Константы класса объявляются с помощью ключевых слов – **static final.** Ключевое слово **final** означает, что присвоить какое-либо значение данной переменной можно лишь один раз и навсегда. Использовать в именах констант только прописные буквы или только строчные необязательно. Ключевое слово **static** перед переменной класса означает, что данная переменная является статической, т.е. значение данной переменной доступно всем методам данного класса и существует в одном экземпляре для всех объектов данного класса.

Модификатор доступа **private** говорит о том, что данная переменная или метод (если private находится перед методом) видимы только для внутренних методов данного класса.

Метод main имеет модификатор **public**. Слово **static** указывает, что метод main – статический, а слово **void** – что он не возвращает никакого значения. Позже мы подробнее разберем, что все это значит.

Далее в скобках у нас идут параметры метода String args[] – это массив args, который хранит значения типа **String**, то есть строки. В данном случае они нам пока не нужны, но в реальной программе это те строковые параметры, которые передаются при запуске программы из командной строки.

Вначале на экран выводится приглашение ввести свое имя с помощью класса System и метода println и объявленной выше константы STR_ENTER_YOUR_NAME. Хотя System является классом, размещенным в одном из пакетов, нам не нужно его подключать с помощью директивы импорта, так как System находится в пакете java.lang, все классы которого автоматически подключаются в программу.

Для работы с потоком ввода необходимо создать объект класса Scanner, указав, с каким потоком ввода он будет связан. Стандартный поток ввода (клавиатура) в Java представлен объектом System.in, а стандартный поток вывода (дисплей) – объектом System.out. Есть еще стандартный поток для вывода ошибок – System.err. Так как класс Scanner находится в пакете java.util, то мы вначале его импортируем. Для создания самого объекта Scanner в его конструктор передается объект System.in. После этого мы можем получать вводимые значения. В данном случае мы создаем строковую переменную name, которой присваиваем строку, считанную объектом Scanner. Метод nextLine() считывает всю введенную строку.

После того как строка считана, с помощью потока вывода System.out мы выводим результат в консоль. Результат получается путем склейки трех строк – 2-х констант и одной переменной.

### Компиляция файлов проекта и запуск приложения

Для сборки проекта следует выбрать в меню среды разработки **Run** > **BuildMainProject** (или клавиша <**F11**>, или на панели инструментов иконка с молотком). При этом происходит компиляция только тех файлов проекта, которые были изменены в процессе редактирования после последней сборки (рисунок 8).

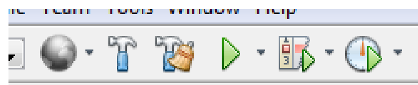

**Рисунок 8. – Панель компиляции и запуска**

Пункт **Run** > **CleanandBuildMainProject** (или комбинация клавиш <**Shift**><**F11**>, или на панели инструментов иконка с молотком и веником) удаляет все выходные файлы проекта (очищает папки **build** и **dist**), после чего заново компилируются все классы проекта.

Пункт **Run** > **RunMainProject** (или, что тоже, клавиша <**F6**>, или на панели инструментов иконка с зеленым и желтыми треугольниками) – после запуска нашего проекта в выходной консоли, которая находится в нижней части окна проекта, появится служебная информация о ходе компиляции и запуска.

В самом начале, перед запуском программы, мы должны запустить очистку и сборку (CleanandBuild) Java приложения (рисунок 9).

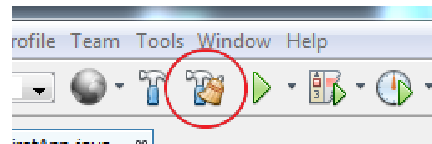


**Рисунок 9. – Сборка проекта**

В окне вывода мы увидим следующую картину (рисунок 10).

Versioning Output | Usages | Notifications | Output - SimpleJavaApp (clean,jar) | Search Results

```
ant -f E:\\Programming\\Projects\\SimpleJavaApp -Dnb.internal.action.name=rebuild clean jar
init:
deps-clean:
Updating property file: E:\Programming\Projects\SimpleJavaApp\build\built-clean.properties
Deleting directory E:\Programming\Projects\SimpleJavaApp\build
clean:
init:
deps-jar:
Created dir: E:\Programming\Projects\SimpleJavaApp\build
Updating property file: E:\Programming\Projects\SimpleJavaApp\build\built-jar.properties
Created dir: E:\Programming\Projects\SimpleJavaApp\build\classes
Created dir: E:\Programming\Projects\SimpleJavaApp\build\empty
Created dir: E:\Programming\Projects\SimpleJavaApp\build\generated-sources\ap-source-output
Compiling 1 source file to E:\Programming\Projects\SimpleJavaApp\build\classes
compile:
Created dir: E:\Programming\Projects\SimpleJavaApp\dist
Copying 1 file to E:\Programming\Projects\SimpleJavaApp\build
Nothing to copy.
Building jar: E:\Programming\Projects\SimpleJavaApp\dist\SimpleJavaApp.jar
To run this application from the command line without Ant, try:
java -jar "E:\Programming\Projects\SimpleJavaApp\dist\SimpleJavaApp.jar"
jar:
BUILD SUCCESSFUL (total time: 0 seconds)
```

**Рисунок 10. – Лог процесса сборки приложения**

В данном случае мы видим лог сборки проекта сборщиком Ant, который по умолчанию используется средой NetBeans.

**ApacheAnt** (англ. Ant – муравей и акроним – «AnotherNeatTool») – утилита для автоматизации процесса сборки программного продукта. Является платформонезависимым аналогом утилиты make, где все команды записываются в XML-формате.

Подробнее узнать про сборщик проектов Ant можно по ссылке: https://habr.com/ru/post/323204/.

Далее запустим наше приложение (рисунок 11).

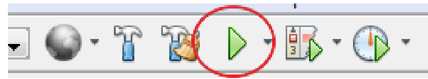

**Рисунок 11. – Запуск приложения**

В консоли мы будем наблюдать следующий результат (рисунок 12).

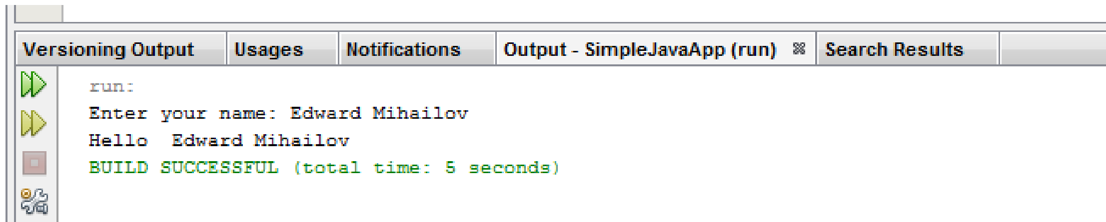


**Рисунок 12. – Результат работы**

### Структура проекта NetBeans

Структура проекта NetBeans представлена на рисунке 13.

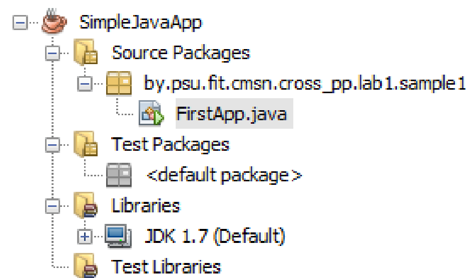


**Рисунок 13. – Структура проекта NetBeans**

Рассмотрим, из каких частей состоит проект NetBeans. На рисунке 13 показаны основные элементы, отображаемые в среде разработки. Это SourcePackages (пакеты исходного кода), TestPackages (пакеты тестирования), Libraries (библиотеки) и TestLibraries (библиотеки поддержки тестирования). Ветви дерева представления элементов проекта можно разворачивать или сворачивать путем нажатия на узлы, отмеченные плюсами и минусами. Мы пока будем пользоваться только пакетами исходного кода.

В компонентной модели NetBeans пакеты приложения объединяются в единую конструкцию – **модуль**. Модули NetBeans являются базовой конструкцией не только для создания приложений, но и для написания библиотек. Они представляют собой оболочку над пакетами (а также могут включать в себя другие модули). В отличие от библиотек Java скомпилированный модуль – это не набор большого количества файлов, а всего один файл, архив JAR (JavaArchive, архив Java). В нашем случае он имеет то же имя, что и приложение, и расширение .jar – это файл SimpleJavaApp.jar.

Модули NetBeans гораздо лучше подходят для распространения, поскольку не только обеспечивают целостность комплекта взаимосвязанных файлов, но и хранят их в заархивированном виде в одном файле, что намного ускоряет копирование и уменьшает объем занимаемого места на носителях. Для того, чтобы установить какой-либо из открытых проектов в качестве главного, следует в дереве проектов правой кнопкой мыши щелкнуть по имени проекта и выбрать пункт меню SetMainProject. Аналогично, для того, чтобы закрыть какой-либо из открытых проектов, следует в дереве проектов правой кнопкой мыши щелкнуть по имени проекта и выбрать пункт меню CloseProject.

Рассмотрим теперь структуру папок проекта NetBeans. По умолчанию головная папка проекта располагается в папке пользователя. Дальнейшее расположение папок и файлов приведено ниже, при этом имена папок выделены жирным шрифтом, а имена вложенных папок и файлов записаны под именами их головных папок и сдвинуты относительно них вправо.

В папке **build** хранятся скомпилированные файлы классов, имеющие расширение .class. В папке **dist** – файлы, предназначенные для распространения как результат компиляции (модуль JAR приложения или библиотеки, а также документация к нему). В папке **nbproject** находится служебная информация по проекту (рисунок 14).

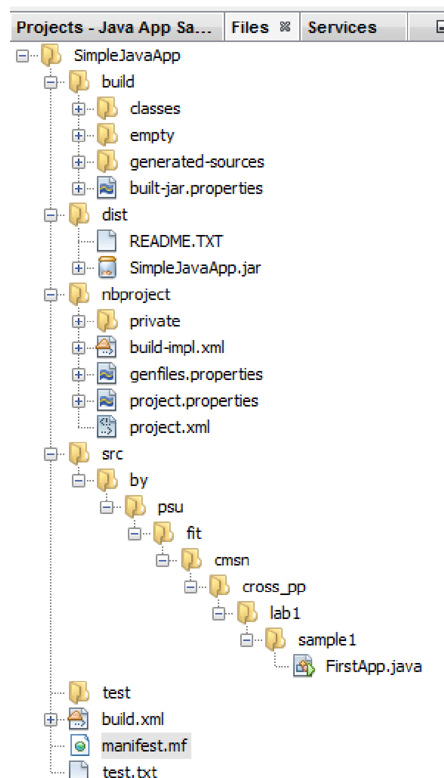


**Рисунок 14. – Информация по проекту**

В папке **src** – исходные коды классов. Кроме того, там же хранится информация об экранных формах (которые будут видны на экране в виде окон с кнопками, текстом и т.п.). Она содержится в XML-файлах, имеющих расширение .form. В папке **test** – сопроводительные тесты, предназначенные для проверки правильности работы классов проекта.

## Основы языка программирования Java

### Типы данных и переменные

Одной из основных особенностей Java является то, что данный язык строго типизированный. А это значит, что каждая переменная представляет определенный тип, и данный тип строго определен.

Итак, рассмотрим систему встроенных базовых типов данных, которая используется для создания переменных в Java. Она представлена следующими типами:

- **boolean**: хранит значение true или false;
- **byte**: хранит целое число от –128 до 127 и занимает 1 байт;
- **short**: хранит целое число от –32768 до 32767 и занимает 2 байта;
- **int**: хранит целое число от –2147483648 до 2147483647 и занимает 4 байта;
- **long**: хранит целое число от –9 223 372 036 854 775 808 до 9 223 372 036 854 775 807 и занимает 8 байт;
- **float**: хранит число с плавающей точкой от $–3,4*10^{38}$ до $3,4*10^{38}$ и занимает 4 байта;
- **double**: хранит число с плавающей точкой от $±4,9*10^{324}$ до $±1,8*10^{308}$ и занимает 8 байт;
- **char**: хранит одиночный символ в кодировке Unicode и занимает 2 байта, поэтому диапазон хранимых значений от 0 до 65536.

### Объявление переменных

Переменные объявляются следующим образом: тип_данных имя_переменной. Например, int x;. В этом выражении мы объявляем переменную x типа int. То есть x будет хранить некоторое число не больше 4 байт.

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

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

– в имени не должно быть знаков пунктуации и пробелов;
– имя не может быть ключевым словом языка Java.

Кроме того, при объявлении и последующем использовании надо учитывать, что Java – **регистрозависимый** язык, поэтому следующие объявления int num; и int NUM; будут представлять две разные переменные.

Объявив переменную, мы можем тут же присвоить ей значение или инициализировать ее. Инициализация переменных представляет присвоение переменной начального значения, например: int x=10;. Если мы не проинициализируем переменную до ее использования, то мы можем получить ошибку (рисунок 15).

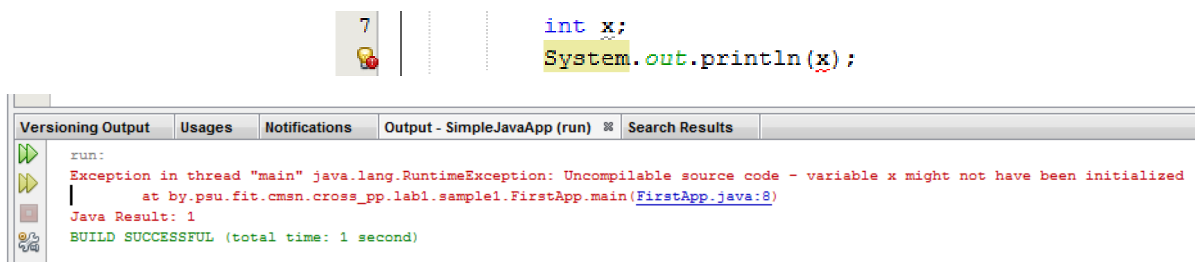


**Рисунок 15. – Исключение при запуске приложения**

Правильный вариант в следующем фрагменте кода:

```
7    int x = 0;
8    System.out.println(x);
```

Следующий фрагмент кода представляет собой варианты объявления переменных:

```
1  boolean active = true;
2  int x;
3  int y=10;
4  byte num = 250;
5  char c='s';
6  double d = 1.5;
7  int a=4;
8  int z=a+5;
```

**Работа с разными системами счисления целых чисел**

Как правило, значения для целочисленных переменных задаются в десятичной системе счисления, однако мы можем применять и другие системы исчисления. Например:

```
1  int num111 = 0x6F; // 16-тиричная система, число 111
2  int num8 = 010; // 8-ричная система, число 8
3  int num13 = 0b1101; // 2-ичная система, число 13
```

Для задания шестнадцатеричного значения после символов 0x указывается число в шестнадцатеричном формате. Таким же образом восьмеричное значение указывается после символа 0, а двоичное значение – после символов 0b.

**Использование суффиксов**

При присвоении переменной типа float значения с плавающей точкой Java автоматически рассматривает это значение как объект типа double. И чтобы указать, что данное значение должно рассматриваться как float, нам надо использовать суффикс f:

```
float fl = 30.6f;
double db = 30.6;
```

И хотя в данном случае обе переменные имеют практически одно значение, но эти значения будут по-разному рассматриваться и будут занимать разное место в памяти.

**Символы и строки**

В качестве значения переменная символьного типа получает одиночный символ, заключенный в одинарные кавычки: char ch='e';. Кроме того, переменной символьного типа также можно присвоить целочисленное значение от 0 до 65536. В этом случае переменная опять же будет хранить символ, а целочисленное значение будет указывать на номер символа в таблице символов Unicode. Например:

```
char ch=102; // символ 'f'
System.out.println(ch);
```

Еще одной формой задания символьных переменных является шестнадцатеричная форма: переменная получает значение в шестнадцатеричной форме, которое следует после символов "\u". Например, charch='\u0066'; опять же будет хранить символ 'f'.

Символьные переменные не стоит путать со строковыми, 'a' не идентично "a". Строковые переменные представляют объект String, который в отличие от char или int не является примитивным типом в Java:

```
String hello = "Hellow...";
System.out.println(hello);
```

В Java также имеются **константы**. В отличие от переменных константам можно присвоить значение только один раз. Константа объявляется так же, как и переменная, только вначале идет ключевое слово **final**:

```
1  final int num=5;
2  num=57; // так мы уже не можем написать, так как num - константа
```

Константы позволяют задать такие переменные, которые не должны больше изменяться. Например, если у нас есть переменная для хранения числа pi, то мы можем объявить ее константой, так как ее значение постоянно.

**Преобразования базовых типов данных**

При рассмотрении типов данных указывалось, какие значения может иметь тот или иной тип и сколько байт памяти он может занимать. И мы можем написать, например, так:

```
1  byte x = 5;
2  byte y = x;
```

Но важно понимать, что эта запись не эквивалентна следующей (хотя результат будет тот же):

```
1  byte x = 5;
2  int y = x;
```

В обоих случаях создается переменная типа byte, которая затем приравнивается другой переменной. Однако, если в первом случае это простое приравнивание, а переменная y просто получает значение переменной x, то во втором примере происходит преобразование типов: данные типа byte преобразуются к типу int. Данный тип преобразований называется **расширяющим**, так как значение типа byte расширяет свой размер до размера типа int. Расширяющие преобразования проходят автоматически и обычно с этим никаких проблем не возникает. Подобным образом происходит преобразование от типа float к типу double или от типа int к типу long.

Кроме расширяющих преобразований есть еще и **сужающие**. Сужающие преобразования позволяют привести данные к типу с меньшей разрядностью, например, от типа int, который занимает 4 байта в памяти, к типу byte, который занимает только 1 байт в памяти:

```
1  int a = 4;
2  byte b = a;
```

Несмотря на то, что значение переменной a (число 4) укладывается в диапазон типа byte, мы все равно получим ошибку. И чтобы безошибочно провести преобразование из одного типа к другому, нам надо применить операцию **приведения типов**. Суть этой операции состоит в том, что в скобках указывается тип, к которому надо привести данное значение:

```
int a = 4;
byte b = (byte) a;
```

**Потеря данных при преобразовании**

В предыдущей ситуации число 4 вполне укладывалось в диапазон значений типа byte. Но что будет в следующем случае:

```
int a = 200;
byte b = (byte) a;
```

**Результатом будет число –56**. В данном случае число 200 вне диапазона для типа byte (от –128 до 127), поэтому произойдет усечение значения. Так как тип byte предполагает 256 возможных значений, то полученное значение будет равно 200 минус 256, то есть –56.

**Усечение рациональных чисел до целых**

При преобразовании значений с плавающей точкой к целочисленным значениям, происходит усечение дробной части:

```
double a = 56.9898;
int b = (int)a;
```

Здесь значение числа b будет равно 56, несмотря на то, что число 57 было бы ближе к 56,9898. Чтобы избежать подобных казусов, надо применять функцию округления, которая есть в математической библиотеке Java:

```
double a = 56.9898;
int b = (int)Math.round(a);
```

**Преобразования при операциях**

Нередки ситуации, когда приходится применять различные операции, например, сложение или произведение, над значениями разных типов. Здесь также действуют некоторые правила:

– если один из операндов операции относится к типу double, то и второй операнд преобразуется к типу double;

– если предыдущее условие не соблюдено, а один из операндов операции относится к типу float, то и второй операнд преобразуется к типу float;

– если предыдущие условия не соблюдены, а один из операндов операции относится к типу long, то и второй операнд преобразуется к типу long;

– иначе все операнды операции преобразуются к типу int.

Например:

```
1 int a = 3;
2 double b = 4.6;
3 double c = a+b;
```

Так как в операции участвует значение типа double, то и другое значение приводится к типу double, и сумма двух значений a+b будет представлять тип double. Другой пример:

```
1 byte a = 3;
2 short b = 4;
3 byte c = (byte)(a+b);
```

Две переменных типа byte и short (не double, float или long), поэтому при сложении они преобразуются к типу int, и их сумма a+b представляет значение типа int. Поэтому если затем мы присваиваем эту сумму переменной типа byte, то нам опять надо сделать преобразование типов к byte.

**Операции языка Java**

Большинство операций в Java аналогичны тем, которые применяются в других си-подобных языках. Есть **унарные** операции (выполняются над одним операндом), **бинарные** – над двумя операндами, а также **тернарные** – выполняются над тремя операндами. Операндом является переменная или значение (например, число), участвующее в операции. Рассмотрим все виды операций.

**Арифметические операции**

Существуют следующие арифметические операции:

1) + – операция сложения двух чисел, например: z=x+y;

2) – – операция вычитания двух чисел: z=x–y;

3) * – операция умножения двух чисел: z=x*y;

4) / – операция деления двух чисел: z=x/y;

5) % – получение остатка от деления двух чисел: z=x%y;

6) ++ (префиксный инкремент) – предполагает увеличение переменной на единицу, например, z=++y (вначале значение переменной y увеличивается на 1, а затем ее значение присваивается переменной z);

7) ++ (постфиксный инкремент) – также увеличение переменной на единицу, например, z=y++ (вначале значение переменной y присваивается переменной z, а потом значение переменной y увеличивается на 1);

8) — (префиксный декремент) – уменьшение переменной на единицу, например, z=—y (вначале значение переменной y уменьшается на 1, а потом ее значение присваивается переменной z);

9) — (постфиксный декремент) – z=y— (сначала значение переменной y присваивается переменной z, а затем значение переменной y уменьшается на 1). Примеры:

```
1  int a1 = 3 + 4; // результат равен 7
2  int a2 = 10 - 7; //результат равен 3
3  int a3 = 10 * 5; //результат равен 50
4  double a4 = 14.0 / 5.0; //результат равен 2.8
5  double a5 = 14.0 % 5.0; //результат равен 4
6  int b1 = 5;
7  int c1 = ++b1; // c1=6; b1=6
8  int b2 = 5;
9  int c2 = b2++; // c2=5; b2=6
10 int b3 = 5;
11 int c3 = --b3; // c3=4; b3=4
12 int b4 = 5;
13 int c4 = b4--; // c4=5; b4=4
```

Операцию сложения также можно применять к строкам, в этом случае происходит конкатенация строк:

```
1  String hello = "hell to " + "world"; //результат равен "hell to world"
```

**Логические операции**

Логические операции над числами представляют поразрядные операции. В данном случае числа рассматриваются в двоичном представлении, например, 2 в двоичной системе равно 10 и имеет два разряда, число 7 – 111 и имеет три разряда.

– & (логическое умножение) – умножение производится поразрядно, и если у обоих операндов значения разрядов равно 1, то операция возвращает 1, иначе возвращается число 0. Например:

```
1  int a1 = 2; //010
2   int b1 = 5;//101
3   System.out.println(a1&b1); // результат 0
4
5  int a2 = 4; //100
6  int b2 = 5; //101
7  System.out.println(a2 & b2); // результат 4
```

В первом случае у нас два числа 2 и 5. Число 2 в двоичном виде представляет число 010, а 5 – 101. Поразрядное умножение чисел (0*1, 1*0, 0*1) дает результат 000.

Во втором случае у нас вместо двойки число 4, у которого в первом разряде 1, так же, как и у числа 5, поэтому здесь результатом операции (1*1, 0*0, 0*1) = 100 будет число 4 в десятичном формате.

– | (логическое сложение) – данная операция также производится по двоичным разрядам, но теперь возвращается единица, если хотя бы у одного числа в данном разряде имеется единица (операция «логическое ИЛИ»). Например:

```
1  int a1 = 2; //010
2  int b1 = 5;//101
3  System.out.println(a1|b1); // результат 7 - 111
4  int a2 = 4; //100
5  int b2 = 5;//101
6  System.out.println(a2 | b2); // результат 5 - 101
```

– ^ (логическое исключающее ИЛИ) – также эту операцию называют XOR, нередко ее применяют для простого шифрования. Например:

```
1  int number = 45; // 1001 Значение, которое надо зашифровать - в двоичной форме 101101
2  int key = 102; //Ключ шифрования - в двоичной системе 1100110
3  int encrypt = number ^ key; //Результатом будет число 1001011 или 75
4  System.out.println("Зашифрованное число: " +encrypt);
5
6  int decrypt = encrypt ^ key; // Результатом будет исходное число 45
7  System.out.println("Расшифрованное число: " + decrypt);
```

Здесь также производятся поразрядные операции. Если у нас значения текущего разряда у обоих чисел разные, то возвращается 1, иначе возвращается 0. Например, результатом выражения 9^5 будет число 12. А чтобы расшифровать число, мы применяем обратную операцию к результату.

– ~ (логическое отрицание) – поразрядная операция, инвертирующая все разряды числа: если значение разряда равно 1, то оно становится равным нулю, и наоборот. Например:

```
1  int a = 56;
2  System.out.println(~a);
```

**Операции сдвига**

Операции сдвига также производятся над разрядами чисел. Сдвиг может происходить вправо и влево:

1) a<<b – сдвигает число a влево на b разрядов. Например, выражение 4<<1 сдвигает число 4 (которое в двоичном представлении 100) на один разряд влево, в результате получается число 1000 или число 8 в десятичном представлении;

2) a>>b – смещает число a вправо на b разрядов. Например, 16>>1 сдвигает число 16 (которое в двоичной системе 10000) на один разряд вправо, то есть в итоге получается 1000 или число 8 в десятичном представлении;

3) a>>>b – в отличие от предыдущих типов сдвигов данная операция представляет беззнаковый сдвиг – сдвигает число a вправо на b разрядов. Например, выражение –8>>>2 будет равно 1073741822. Таким образом, если исходное число, которое надо сдвинуть в ту или другую строну, делится на два, то фактически получается умножение или деление на два. Поэтому подобную операцию можно использовать вместо непосредственного умножения или деления на два, так как операция сдвига на аппаратном уровне менее дорогостоящая операция в отличие от операции деления или умножения.

**Операции сравнения**

В данных операциях сравниваются два операнда, и возвращается значение типа boolean – true, если выражение верно, и false, если выражение неверно:

1) **== –** данная операция сравнивает два операнда на равенство: c=a==b (c равно true, если a равно b, иначе c будет равно false);

2) **!= –** c=a!=b; (c равно true, если a не равно b, иначе c будет равно false);

3) **< –** c=a<b; (c равно true, если a меньше b, иначе c будет равно false);

4) **> –** c=a>b; (c равно true, если a больше b, иначе c будет равно false);

5) **<= –** c=a<=b; (c равно true, если a меньше или равно b, иначе c будет равно false);

6) **>= –** c=a>=b; (c равно true, если a больше или равно b, иначе c будет равно false).

**Логические операторы**

Кроме собственно операций сравнения в Java также определены логические операторы, которые возвращают значение типа boolean. Выше мы рассматривали поразрядные операции над числами. Теперь рассмотрим эти же операторы при операциях над булевыми значениями:

1) **| –** c=a|b; (c равно true, если либо a, либо b (либо и a, и b) равны true, иначе c будет равно false);

2) **& –** c=a&b; (c равно true, если и a, и b равны true, иначе c будет равно false);

3) **! –** c!=b; (c равно true, если b равно false, иначе c будет равно false);

4) **^ –** c=a^b; (c равно true, если либо a, либо b (но не одновременно) равны true, иначе c будет равно false);

5) **|| –** c=a||b; (c равно true, если либо a, либо b (либо и a, и b) равны true, иначе c будет равно false);

6) **&& –** c=a&&b; (c равно true, если и a, и b равны true, иначе c будет равно false).

Здесь у нас две пары операций | и || (а также & и &&) выполняют похожие действия, однако же они не равнозначны. Выражение c=a|b; будет вычислять сначала оба значения – a и b и на их основе выводить результат. В выражении же c=a||b; вначале будет вычисляться значение a, и если оно равно true, то вычисление значения b уже смысла не имеет, так как у нас в любом случае уже c будет равно true. Значение b будет вычисляться только в том случае, если a равно false.

То же самое касается пары операций &/&&. В выражении c=a&b; будут вычисляться оба значения – a и b. В выражении же c=a&&b; сначала будет вычисляться значение b, и если оно равно false, то вычисление значения b уже не имеет смысла, так как значение c в любом случае равно false. Значение b будет вычисляться только в том случае, если a равно true.

Таким образом, операции || и && более удобны в вычислениях, позволяют сократить время на вычисление значения выражения и тем самым повышают производительность. А операции | и & больше подходят для выполнения поразрядных операций над числами. Примеры:

```
1 boolean a1 = (5 > 6) || (4 < 6); // 5 > 6 - false, 4 < 6 - true, поэтому возвращается true
2 boolean a2 = (5 > 6) || (4 > 6); // 5 > 6 - false, 4 > 6 - false, поэтому возвращается false
3 boolean a3 = (5 > 6) && (4 < 6); // 5 > 6 - false, 4 < 6 - true, поэтому возвращается false
4 boolean a4 = (50 > 6) && (4 / 2 < 3); // 50 > 6 - true, 4/2 < 3 - true, поэтому возвращается true
5 boolean a5 = (5 > 6) ^ (4 < 6); // 5 > 6 - false, 4 < 6 - true, поэтому возвращается true
6 boolean a6 = (50 > 6) ^ (4 / 2 < 3); // 50 > 6 - true, 4/2 < 3 - true, поэтому возвращается false
```

**Операции присваивания**

В завершении рассмотрим операции присваивания, которые в основном представляют комбинацию простого присваивания с другими операциями:

1) **= –** просто приравнивает одно значение другому: c=b;

2) **+= –** c+=b; (переменной c присваивается результат сложения c и b);

3) **–= –** c–=b; (переменной c присваивается результат вычитания b из c);

4) ***= –** c*=b; (переменной c присваивается результат произведения c и b);

5) **/= –** c/=b; (переменной c присваивается результат деления c на b);

6) **%= –** c%=b; (переменной c присваивается остаток от деления c на b);

7) **&= –** c&=b; (переменной c присваивается значение c&b);
8) **|= –** c|=b; (переменной c присваивается значение c|b);
9) **^= –** c^=b; (переменной c присваивается значение c^b);
10) **<<= –** c<<=b; (переменной c присваивается значение c<<b);
11) **>>= –** c>>=b; (переменной c присваивается значение c>>b);
12) **>>>= –** c>>>=b; (переменной c присваивается значение c>>>b).

**Массивы**

Если переменные предназначены для хранения одиночного значения, то массив представляет набор однотипных значений. Объявление массива похоже на объявление переменной: тип_данных название_массива[], либо тип_данных[] название_массива. Например:

```
1  int nums[];
2  char[] stroka;
```

Вместе с объявлением мы можем сразу же создать массив:

```
1  int nums[] = new int[4];
2  char[] stroka;
3  stroka = new char[6];
```

Создание массива производится с помощью следующей конструкции: newтип_данных[количество_элементов], где new – ключевое слово, выделяющее память для указанного в скобках количества элементов. Например, int nums[] = new int[4]; – в этом выражении создается массив из четырех элементов int и каждый элемент по умолчанию равен нулю.

После создания массива мы можем обратиться к любому его элементу и изменить его:

```
1  int[] nums = new int[4];
2  nums[0] = 1;
3  nums[1] = 2;
4  nums[2] = 4;
5  nums[3] = 100;
6  
7  System.out.println(nums[3]);
```

Отсчет элементов массива начинается с 0, поэтому в данном случае, чтобы обратиться к четвертому элементу в массиве, нам надо использовать выражение nums[3]. Так как у нас массив определен только для 4 элементов, то мы не можем обратиться, например, к шестому элементу: nums[5] = 5;. Если мы так попытаемся сделать, то мы получим ошибку.

В предыдущем примере мы сначала создали массив, а потом по отдельности проинициализировали каждый его элемент. Однако есть и альтернативные пути инициализации массивов:

```
// эти два способа равноценны
int[] nums2 = new int[] { 1, 2, 3, 5 };

int[] nums3 = { 1, 2, 3, 5 };
```

Здесь мы сразу указываем все элементы массива.

**Многомерные массивы**

Ранее мы рассматривали одномерные массивы, которые можно представить как цепочку или строку однотипных значений. Но кроме одномерных массивов также бывают и многомерные. Наиболее известный многомерный массив – таблица, представляющая двухмерный массив:

```
int[] nums1 = new int[] { 0, 1, 2, 3, 4, 5 };

int[][] nums2 = { { 0, 1, 2 }, { 3, 4, 5 } };
```

Визуально оба массива можно представить следующим образом (рисунок 16).

**Рисунок 16. – Визуальное представление массивов**

Поскольку массив nums2 двухмерный, он представляет собой простую таблицу. Его также можно было создать следующим образом: int[][] nums2 = new int[3][3];. Количество квадратных скобок указывает на размерность массива, а числа в скобках – на количество строк и столбцов. И также, используя индексы, мы можем обрабатывать элементы массива в программе:

```
// установим элемент первого столбца второй строки
nums2[1][0]=44;
System.out.println(nums2[1][0]);
```

Объявление трехмерного массива могло бы выглядеть так:

```
1  int[][][] nums3 = new int[2] [3][4];
```

**Массив массивов**

Многомерные массивы могут быть также представлены как «зубчатые массивы». В вышеприведенном примере двухмерный массив имел три строчки и три столбца, поэтому у нас получалась ровная таблица. Но мы можем каждому элементу в двухмерном массиве присвоить отдельный массив с различным количеством элементов:

```
1  int[][] nums = new int[3][];
2  nums[0] = new int[2];
3  nums[1] = new int[3];
4  nums[2] = new int[5];
```

**Работа с массивами и класс Arrays**

Важнейшее свойство, которым обладают массивы, – свойство **length**, возвращающее длину массива, то есть количество его элементов: int length = nums.length;.

Для работы с массивами в библиотеке классов Java в пакете java.util определен специальный класс **Arrays**. С его помощью мы можем производить ряд операций над массивами.

**Копирование массивов**

Массивы, так же, как и переменные, мы можем присваивать:

```
1  int[] numbers = new int[] { 1, 2, 3, 5 };
2  int[] figures = numbers;
3
4  figures[2]=30;
5  System.out.println(numbers[2]); // равно 30
```

Здесь два массива, второму присваивается первый массив. Однако на самом деле при присвоении переменная figures будет хранить ссылку на область в памяти, где находится массив. В итоге и figures, и numbers будут указывать на один и тот же массив, и если мы изменим элемент в массиве figures- figures[2]=30, то изменится и массив numbers, так как это фактически один и тот же массив. Чтобы такой проблемы избежать, надо использовать копирование массивов.

Для копирования используется метод Arrays.copyOf. Например:

```
import java.util.Arrays;

public class Program {

    public static void main(String[] args) {

        int[] numbers = new int[] { 1, 2, 3, 5 };
        int[] figures = Arrays.copyOf(numbers, numbers.length);

        figures[2]=30;
        System.out.println(numbers[2]); // равно 3
    }
}
```

Метод Arrays.copyOf(numbers, numbers.length) принимает два параметра: первый параметр – массив, который надо скопировать, а второй параметр – сколько элементов надо скопировать.

**Сортировка**

С помощью метода Arrays.sort можно отсортировать массив. Фрагмент кода:

```
// элементы массива в произвольном порядке
int[] numbers = new int[] { 1, 7, 3, 5, 2, 6, 4 };

Arrays.sort(numbers);

// в цикле выводим все элементы массива по порядку
for(int i=0;i<numbers.length;i++)
    System.out.println(numbers[i]);
```

**Условные конструкции**

Одним из фундаментальных элементов многих языков программирования являются условные конструкции. Данные конструкции позволяют направить работу программы по одному из путей в зависимости от определенных условий.

В языке Java используются следующие условные конструкции: if..else и switch..case.

**Конструкция if/else**

Выражение **if/else** проверяет истинность некоторого условия и в зависимости от результатов проверки выполняет определенный код:

```
int num1 = 6;
int num2 = 4;
if(num1>num2){
    System.out.println("Первое число больше второго");
}
```

После ключевого слова if ставится условие. Если это условие выполняется, то срабатывает код, который помещен далее в блоке if после фигурных скобок. В качестве условий выступает операция сравнения двух чисел.

Так как в данном случае первое число больше второго, то выражение num1 > num2 истинно и возвращает значение true. Следовательно, управление переходит в блок кода после фигурных скобок и начинает выполнять содержащиеся там инструкции, а конкретно метод System.out.println; («Первое число больше второго»). Если бы первое число оказалось меньше второго или равно ему, то инструкции в блоке if не выполнялись бы.

Но что, если мы захотим, чтобы при несоблюдении условия также выполнялись какие-либо действия? В этом случае мы можем добавить блок else:

```
1  int num1 = 6;
2  int num2 = 4;
3  if(num1>num2){
4      System.out.println("Первое число больше второго");
5  }
6  else{
7      System.out.println("Первое число меньше второго");
```

Но при сравнении чисел мы можем насчитать три состояния: первое число больше второго, первое число меньше второго и числа равны. С помощью выражения elseif, мы можем обрабатывать дополнительные условия как в следующем фрагменте кода:

```
1  int num1 = 6;
2  int num2 = 8;
3  if(num1>num2){
4      System.out.println("Первое число больше второго");
5  }
6  else if(num1<num2){
7      System.out.println("Первое число меньше второго");
8  }
9  else{
10     System.out.println("Числа равны");
11 }
```

Также мы можем соединить сразу несколько условий, используя логические операторы:

```
1  int num1 = 8;
2  int num2 = 6;
3  if(num1 > num2 && num1>7){
4      System.out.println("Первое число больше второго и больше 7");
5  }
```

Здесь блок if будет выполняться, если num1 > num2 равно true и одновременно num1>7 равно true.

**Конструкция switch**

Конструкция **switch/case** аналогична конструкции if/else, так как позволяет обработать сразу несколько условий:

```
int num = 8;
switch(num){

    case 1:
        System.out.println("число равно 1");
        break;
    case 8:
        System.out.println("число равно 8");
        num++;
        break;
    case 9:
        System.out.println("число равно 9");
        break;
    default:
        System.out.println("число не равно 1, 8, 9");
}
```

После ключевого слова switch в скобках идет сравниваемое выражение. Значение этого выражения последовательно сравнивается со значениями, помещенными после оператора case. И если совпадение будет найдено, то будет выполняться определенный блок case.

В конце блока case ставится оператор break, чтобы избежать выполнения других блоков. Например, если бы убрали оператор break в следующем случае:

```
case 8:
    System.out.println("число равно 8");
    num++;
case 9:
    System.out.println("число равно 9");
    break;
```

то так как у нас переменная num равна 8, выполнился бы блок case 8, но так как в этом блоке переменная num увеличивается на единицу, оператор break отсутствует, то начал бы выполняться блок case 9.

Если мы хотим также обработать ситуацию, когда совпадение не будет найдено, то можно добавить блок default, как в примере выше. Хотя блок default необязателен.

Начиная с JDK 7 в выражении switch..case, кроме примитивных типов можно также использовать строки, фрагмент кода:

```
package firstapp;

import java.util.Scanner;

public class FirstApp {

    public static void main(String[] args) {

        Scanner in = new Scanner(System.in);
        System.out.println("Введите Y или N: ");
        String input= in.nextLine();

        switch(input){

            case "Y":
                System.out.println("Вы нажали букву Y");
                break;
            case "N":
                System.out.println("Вы нажали букву N");
                break;
            default:
                System.out.println("Вы нажали неизвестную букву");
```

**Тернарная операция**

Тернарная операция имеет следующий синтаксис: [первый операнд – условие]? [второй операнд]: [третий операнд]. Таким образом, в этой операции участвуют сразу три операнда. В зависимости от условия тернарная операция возвращает второй или третий операнд: если условие равно true, то возвращается второй операнд; если условие равно false, то третий. Например:

```
int x=3;
int y=2;
int z = x<y? (x+y) : (x-y);
System.out.println(z);
```

Здесь результатом тернарной операции является переменная z. Сначала проверяется условие x<y, если оно соблюдается, то z будет равно второму операнду – (x+y), иначе z будет равно третьему операнду.

**Циклы**

Еще одним видом управляющих конструкций являются циклы. Циклы позволяют в зависимости от определенных условий выполнять определенное действие множество раз. В языке Java есть следующие виды циклов:

– for;
– while;
– do..while.

**Цикл for**

Цикл **for** имеет следующее формальное определение:

```
for ([инициализация счетчика]; [условие]; [изменение счетчика])
{
    // действия
}
```

Рассмотрим стандартный цикл for:

```
for (int i = 1; i < 9; i++){
    System.out.printf("Квадрат числа %d равен %d \n", i, i * i);
}
```

Первая часть объявления цикла – int i = 1 создает и инициализирует счетчик i. Счетчик необязательно должен представлять тип int. Это может быть и любой другой числовой тип, например, float. Перед выполнением цикла значение счетчика будет равно 1. В данном случае это то же самое, что и объявление переменной. Вторая часть – условие, при котором будет выполняться цикл. В данном случае цикл будет выполняться, пока i не достигнет 9. И третья часть – приращение счетчика на единицу. Опять же нам необязательно увеличивать на единицу. Можно уменьшать: i––. В итоге блок цикла сработает 8 раз, пока значение i не станет равным 9. И каждый раз это значение будет увеличиваться на 1.

Нам необязательно указывать все условия при объявлении цикла. Например, мы можем написать так:

```
int i = 1;
for (; ;){
    System.out.printf("Квадрат числа %d равен %d \n", i, i * i);
}
```

Определение цикла осталось тем же, только теперь блоки в определении у нас пустые: for (; ;). Теперь нет инициализированной переменной-счетчика, нет условия, поэтому цикл будет работать вечно – бесконечный цикл.

Либо можно опустить ряд блоков:

```
int i = 1;
for (; i<9;){
    System.out.printf("Квадрат числа %d равен %d \n", i, i * i);
}
```

Этот пример эквивалентен первому примеру: у нас также есть счетчик, только создан он вне цикла. У нас есть условие выполнения цикла. И есть приращение счетчика уже в самом блоке for.

Специальная версия цикла for предназначена для перебора элементов в наборах элементов, например, в массивах и коллекциях. Она аналогична действию цикла foreach, который имеется в других языках программирования. Формальное ее объявление:

```
for (тип_данных название_переменной : контейнер){
    // действия
}
```

Например:

```
int[] array = new int[] { 1, 2, 3, 4, 5 };
for (int i : array){

    System.out.println(i);
}
```

В качестве контейнера в данном случае выступает массив данных типа int. Затем объявляется переменная с типом int.

То же самое можно было бы сделать и с помощью обычной версии for:

```
int[] array = new int[] { 1, 2, 3, 4, 5 };
for (int i = 0; i < array.length; i++){
    System.out.println(array[i]);
}
```

В то же время эта версия цикла for более гибкая по сравнению for (int i :array). В частности, в этой версии мы можем изменять элементы:

```
int[] array = new int[] { 1, 2, 3, 4, 5 };
for (int i=0; i<array.length;i++){
    array[i] = array[i] * 2;
    System.out.println(array[i]);
}
```

**Перебор многомерных массивов в цикле**

```
int[][] nums = new int[][]
{
    {1, 2, 3},
    {4, 5, 6},
    {7, 8, 9}
};
for (int i = 0; i < nums.length; i++){
    for(int j=0; j < nums[i].length; j++){

        System.out.print(nums[i][j]);
        System.out.print(" ");
    }
    System.out.println();
}
```

Сначала создается цикл для перебора по строкам, а затем внутри первого цикла создается внутренний цикл для перебора по столбцам конкретной строки. Подобным образом можно перебрать и трехмерные массивы, и наборы с большим количеством размерностей.

### Цикл do

Цикл **do** сначала выполняет код цикла, а потом проверяет условие в инструкции while. И пока это условие истинно, цикл повторяется. Например:

```
int j = 7;
do{
    System.out.println(j);
    j--;
}
while (j > 0);
```

В данном случае код цикла сработает 7 раз, пока j не окажется равным нулю. Важно отметить, что цикл do гарантирует хотя бы однократное выполнение действий, даже если условие в инструкции while не будет истинно. Так, мы можем написать:

```
int j = -1;
do{
    System.out.println(j);
    j--;
}
while (j > 0);
```

Хотя переменная j изначально меньше 0, цикл все равно один раз выполнится.

### Цикл while

Цикл **while** сразу проверяет истинность некоторого условия, и если условие истинно, то код цикла выполняется:

```
int j = 6;
while (j > 0){

    System.out.println(j);
    j--;
}
```

### Операторы continue и break

Иногда требуется выйти из цикла, не дожидаясь его завершения. В этом случае мы можем воспользоваться оператором **break**, например:

```
int[] nums = new int[] { 1, 2, 3, 4, 12, 9 };
for (int i = 0; i < nums.length; i++){
    if (nums[i] > 10)
        break;
    System.out.println(nums[i]);
}
```

Так как в цикле идет проверка, больше ли элемент массива 10, то мы не увидим на консоли последние два элемента, потому что когда nums[i] окажется больше 10 (то есть равно 12), сработает оператор break, и цикл завершится. Правда, мы также не увидим и последнего элемента, который меньше 10.

Теперь сделаем так, чтобы если число больше 10, цикл не завершался, а просто переходил к следующему элементу. Для этого используем оператор continue:

```
int[] nums = new int[] { 1, 2, 3, 4, 12, 9 };
for (int i = 0; i < nums.length; i++){

    if (nums[i] > 10)
        continue;
    System.out.println(nums[i]);
}
```

В этом случае, когда выполнение цикла дойдет до числа 12, которое не удовлетворяет условию проверки, программа просто пропустит это число и перейдет к следующему элементу массива.

## ЗАДАНИЕ К ЛАБОРАТОРНОЙ РАБОТЕ

1. Ознакомиться с теоретической частью.

2. Создать проект с приведенным примером в среде NetBeans и запустить его.

3. Реализовать приложения на языке Java для решения следующих задач:

– в переменных q и w хранятся два натуральных числа. Создайте программу, выводящую на экран результат деления q на w с остатком. Пример вывода программы (для случая, когда в q хранится 21, а в w хранится 8): 21 / 8 = 2 и 5 в остатке;

– в переменной n хранится натуральное двузначное число. Создайте программу, вычисляющую и выводящую на экран сумму цифр числа n;

– в переменной n хранится вещественное число с ненулевой дробной частью. Создайте программу, округляющую число n до ближайшего целого и выводящую результат на экран;

– в переменной n хранится натуральное трехзначное число. Создайте программу, вычисляющую и выводящую на экран сумму цифр числа n;

– создать двумерный массив из N строк по M столбца в каждой (N и M константы) из случайных целых чисел из отрезка [–5;5]. Вывести массив на экран. Определить и вывести на экран индекс строки с наибольшим по модулю произведением элементов. Если таких строк несколько, то вывести индекс первой встретившейся из них (для генерации случайных чисел используйте класс java.util.Random);

– задан целочисленный массив размерности N. Есть ли среди элементов массива простые числа? Если да, то вывести номера этих элементов.

## КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что произойдет после компиляции кода и запуска программы без аргументов?

```
public class MainClass
{
public static void main(String[] args)
    {
System.out.println(args.length);
}
}
```

2. Какой из предложенных вариантов скомпилируется без ошибок и предупреждений. Пояснить ответ.

1) float f=1.3;
2) char c="a";
3) byte b=257;
4) boolean b=null;
5) int i=10;

3. Размер byte равен:

1) –128 до 127
2) (–2 в степени 8)–1 до 2 в степени 8
3) –255 до 256
4) зависит от реализации JVM

4. Какое из данных слов является зарезервированным словом Java:

1) if
2) then
3) goto
4) while
5) case

5. Какое из приведенных слов является правильным идентификатором Java (например, названием переменной):

1) 2variable

2) variable2

3) _whatavariable

4) _3_

5) $anothervar

6) #myvar

6. Какой будет результат выполнения следующего кода:

```
public class MyClass
{
static int i;

public static void main(String argv[])
{
     System.out.println(i);
}
}
```

7. Какой будет результат выполнения следующего кода:

```
public class Q
{
public static void main(String argv[])
{
     int anar[]=new int[]{1,2,3};
     System.out.println(anar[1]);
}
}
```

8. Какой будет результат выполнения следующего кода:

```
public class Q
{
public static void main(String argv[])
{
     int anar[]=new int[5];
     System.out.println(anar[0]);
}
}
```

9. Какой будет результат выполнения следующего участка кода:

```
int i=1;
switch (i)
     {
     case 0:
     System.out.println("zero");
     break;
     case 1:
     System.out.println("one");
     case 2:
     System.out.println("two");
     default:
     System.out.println("default");
 }
```

10. Какой будет результат выполнения следующего участка кода:

```
int i=9;
switch (i)
     {
     default:
          System.out.println("default");
     case 0:
     System.out.println("zero");
     break;
     case 1:
     System.out.println("one");
     case 2:
     System.out.println("two");
}
```

11. Какой будет результат выполнения следующего участка кода:

```
int i=0;
if (i) { System.out.println("Hello"); }
```

12. Какой будет результат выполнения следующего участка кода:

```
boolean b=true;
boolean b2=true;
if(b==b2) {
System.out.println("So true");
 }
```

13. Какой будет результат выполнения следующего участка кода:

```
int i=1;
int j=2;
if(i==1|| j==2)
System.out.println("OK");
```

14. Какой будет результат выполнения следующего участка кода:

```
int i=1;
int j=2;
if(i==1 &| j==2) System.out.println("OK");
```

15. Какой будет результат выполнения следующего кода:

```
 public class MyFor{
public static void main(String argv[]){
 int i;
 int j;
outer:
 for (i=1;i <3;i++)
     inner:
 for(j=1; j<3; j++) {
     if (j==2)
      continue outer;
 System.out.println("Value for i=" + i + " Value for j=" +j);
 }
}
}
```

16. Какой будет результат выполнения следующего участка кода:

```
System.out.println(4 | 3);
```

17. Какой будет результат выполнения следующего кода:

```
public class MyAr{
public static void main(String argv[]){
int[] i = new int[5];
System.out.println(i[5]);
        }
}
```

18. Какой метод используется для получения количества элементов в массиве. Приведите пример.

19. Какой будет результат выполнения следующего участка кода:

```
intk=10;
switch(k){
default:
System.out.println("Thisisthedefaultoutput");
break;
case 10:
System.out.println("ten");
case 20:
System.out.println("twenty");
break;
       }
```

20. Какой из представленных вариантов является верным определением переменной в Java:

1) short myshort = 99S;

2) String name = 'Excellent tutorial Mr Green';

3) char c = 17c;

4) int z = 015;

21. Какой будет результат выполнения следующего участка кода:

```
     System.out.println(Math.round(-2.1)+" "+Math.ceil(-2.1)+"
"+Math.floor(-2.1));
     System.out.println(Math.round(-2.5)+" "+Math.ceil(-2.5)+"
"+Math.floor(-2.5));
     System.out.println(Math.round(2.1)+" "+Math.ceil(2.1)+"
"+Math.floor(2.1));
     System.out.println(Math.round(2.5)+" "+Math.ceil(2.5)+"
"+Math.floor(2.5));
```

22. Какой будет результат выполнения следующего участка кода в классе Cycle, если программа была запущена следующим образом:

```
          java Cycle one two

     public static void main(String bicycle[]){
System.out.println(bicycle[0]);
}
```

23. Какой будет результат выполнения следующего участка кода:

```
System.out.println(010|4);
```

24. Объявлены переменные:

char c = 'c';

int i = 10;

double d = 10;

long l = 1;

String s = "Hello";

какой из предложенных вариантов выполнится без ошибок:

1) c=c+i;

2) s+=i;

3) i+=s;

4) c+=s;

## СОДЕРЖАНИЕ ОТЧЕТА

1. Ф.И.О., группа, название лабораторной работы.
2. Цель работы.
3. Описание проделанной работы.
4. Результаты выполнения лабораторной работы.
5. Выводы.

Отчет и исходные коды запаковать в архив с названием по следующему шаблону:

Lab1–«группа, аббревиатура на латинице»–«Фамилия на латинице».

Пример: Lab1–13VS–Ivanov.zip.

### Порядок защиты лабораторной работы

1. Предоставление отчета о выполнении лабораторной работы.
2. Демонстрация программы.
3. Выполнение дополнительного задания.
4. Ответы на вопросы по теме текущей лабораторной работы.

## Лабораторная работа № 2. Классы и объекты в Java

**Цель работы.** Рассмотреть структуру класса в языке программирования Java. Познакомиться с различными типами объектов и классов, научиться применять их на практике.

### Теоретическая часть

#### Классы и объекты

Java является объектно-ориентированным языком, поэтому такие понятия как «класс» и «объект» играют в нем ключевую роль. Любую программу на Java можно представить как набор взаимодействующих между собой объектов.

Шаблоном или описанием объекта является класс, а объект представляет экземпляр этого класса. Можно еще провести следующую аналогию. У нас у всех есть некоторое представление о человеке – наличие двух рук, двух ног, головы, пищеварительной, нервной системы, головного мозга и т.д. Есть некоторый шаблон – этот шаблон можно назвать классом. Реально же существующий человек (фактически экземпляр данного класса) является объектом этого класса.

**Структура класса**

Класс определяется с помощью ключевого слова **class**:

```
public class Book
{

}
```

Рассмотрим составляющие нашего класса:

1) **public** – модификатор доступа к классу, в данном случае он нам говорит, что этот класс будет доступен не только данному классу, но и другим. В Java существуют и другие модификаторы;

2) **class** – ключевое слово, говорящее о том, что это класс;

3) **Book** – имя класса. Имена классов принято писать с заглавной буквы;

4) { } – фигурные скобки, между которыми разместится тело нашего класса.

Класс содержит обычно члены двух типов: **поля** – переменные, принадлежащие классу и способные определять свойства объекта (или класса), и **методы** – обособленные именованные фрагменты кода. Также класс может содержать вложенные классы и интерфейсы. При объявлении класса перед именем класса используются ключевые слова, называемые **модификаторами**.

Например, класс **Book** мог бы иметь следующее определение:

```
class Book
{
    public String name;
    public String author;
    public int year;

    public void info()
    {
        System.out.printf("Book '%s' (author %s) was published in %d \n",
                          name, author, year);
    }
}

```

Таким образом, в классе **Book** определены три переменных и один метод, который выводит значения этих переменных.

**Функции и методы класса**

**Методы** – это подпрограммы, присоединенные к конкретным определениям классов. Они описываются внутри определения класса на том же уровне, что и переменные объектов. При объявлении метода задаются тип возвращаемого им результата и список параметров. Общая форма объявления метода такова:

```
[модификаторы] тип_возвращаемого_значения название_метода ([параметры]){
    // тело метода
}
```

Модификаторы и параметры необязательны.

По умолчанию главный класс любой программы на Java содержит метод **main**, который служит точкой входа в программу:

```
public static void main(String[] args) {
    System.out.println("привет мир!");
}
```

Ключевые слова public и static являются модификаторами. Далее идет тип возвращаемого значения. Ключевое слово **void** указывает на то, что метод ничего не возвращает.

Затем идут название метода – main и в скобках параметры метода – String[] args. В фигурные скобки заключено тело метода – все действия, которые он выполняет.

Условно методы, которые не возвращают никакого значения, называются **процедурами**.

Кроме **void** методы в Java могут возвращать конкретное значение. Такие методы также условно называют **функциями**.

В функции в качестве типа возвращаемого значения вместо **void** используется любой другой тип. Функции также отличаются тем, что мы обязательно должны использовать оператор **return**, после которого ставится возвращаемое значение. При этом возвращаемое значение всегда должно иметь тот же тип, что значится в определении функции. И если функция возвращает значение типа int, то после оператора return стоит целочисленное значение, которое неявно является объектом типа int.

Кроме обычных методов в классах используются также и специальные методы, которые называются **конструкторами**. Конструкторы нужны для создания нового объекта данного класса и, как правило, выполняют начальную инициализацию объекта. Название конструктора должно совпадать с названием класса, например:

```
class Book
{
    public String name;
    public String author;
    public int year;

    public void info()
    {
        System.out.printf("Book '%s' (author %s) was published in %d \n",
                name, author, year);
    }

    public Book()
    {
    }

    public Book(String name, String author, int year)
    {
        this.name = name;
        this.author = author;
        this.year = year;
    }
}
```

Здесь у класса Book определено два конструктора. Первый конструктор без параметров присваивает неопределенные начальные значения полям. Второй конструктор присваивает полям класса значения, которые передаются через его параметры.

Так как имена параметров и имена полей класса в данном случае у нас совпадают – name, author, year, то мы используем ключевое слово **this**. Это ключевое слово представляет ссылку на текущий объект. Поэтому в выражении this.name = name; первая часть this.name означает, что name – это поле текущего класса, а не название параметра name. Если бы у нас параметры и поля назывались по-разному, то использовать слово this было бы *необязательно.*

Мы можем определить несколько конструкторов для установки разного количества параметров и затем вызывать один конструктор из другого, например:

```
class Book
{
    public String name;
    public String author;
    public int year;

    public void info()
    {
        System.out.printf("Book '%s' (author %s) was published in %d \n",
                    name, author, year);
    }

    public Book(String name, String author)
    {
        this.name = name;
        this.author = author;
    }

    public Book(String name, String author, int year)
    {
        this(name, author);
        this.year = year;
    }
}
```

К примеру, у нас может сложиться ситуация, когда нам нужно установить только два параметра или только три, однако устанавливать конструкторами с тремя параметрами все три поля класса не имеет смысла, так как мы можем передать два из них в другой конструктор класса, где и произойдет их установка. Вызов конструктора производится с помощью ключевого слова **this**, после которого идет в скобках список параметров.

### Создание объекта

Чтобы непосредственно использовать класс в программе, надо создать его объект. Процесс создания объекта двухступенчатый: вначале объявля-

ется переменная данного класса, а затем с помощью ключевого слова new и конструктора непосредственно создается объект, на который и будет указывать объявленная переменная, например:

```
1  public static void main(String[] args)
2  {
3      Book b;
4      b = new Book();
5  }
```

После объявления переменной Book b; эта переменная еще не ссылается ни на какой объект и имеет значение null. Затем мы создаем непосредственно объект класса Book с помощью одного из конструкторов и ключевого слова **new**.

**Инициализаторы**

Кроме конструктора начальную инициализацию объекта вполне можно было проводить с помощью инициализатора объекта. Так, мы бы могли заменить конструктор без параметров следующим фрагментом кода:

```
1  class Book
2  {
3      public static final String DEFAULT_NAME   = "unknown";
4      public static final String DEFAULT_AUTHOR = "unknown";
5      public static final int    DEFAULT_YEAR   = 0;
6
7      public String name;
8      public String author;
9      public int year;
10
11     {
12         name   = DEFAULT_NAME;
13         author = DEFAULT_AUTHOR;
14         year   = 0;
15     }
16
17     public Book(String name, String author, int year)
18     {
19         this.name = name;
20         this.author = author;
21         this.year = year;
22     }
23 }
```

**Пакеты**

Как правило, в Java классы объединяются в пакеты. Пакеты позволяют организовать классы логически в наборы. По умолчанию Java уже имеет ряд встроенных пакетов, например, java.lang, java.util, java.io и т.д. Кроме того, пакеты могут иметь вложенные пакеты.

Организация классов в виде пакетов позволяет избежать конфликта имен между классами. Ведь нередки ситуации, когда разработчики называют свои классы одинаковыми именами. Принадлежность к пакету позволяет гарантировать однозначность имен.

Чтобы указать, что класс принадлежит определенному пакету, надо использовать директиву package, после которой указывается имя пакета, например:

```
1  package by.psu.fit.cmsn.cross_pp.lab1.sample1;
```

Классы необязательно определять в пакеты. Если для класса пакет не определен, то считается, что данный класс находится в пакете по умолчанию, который не имеет имени (см. пункт Соглашения о присвоении имен пакетов в лабораторной работе № 1).

**Импорт пакетов и классов**

Если нам надо использовать классы из других пакетов, то необходимо подключить эти пакеты и классы. Исключение составляют классы из пакета java.lang (например, String), которые подключаются в программу автоматически.

Например, класс Scanner находится в пакете java.util, поэтому мы можем получить к нему доступ следующим способом:

```
1  java.util.Scanner in = new java.util.Scanner(System.in);
```

То есть мы указываем полный путь к файлу в пакете при создании его объекта. Однако такое нагромождение имен пакетов не всегда удобно, и в качестве альтернативы мы можем импортировать пакеты и классы в проект с помощью директивы **import**, которая указывается после директивы **package**:

```
1  package by.psu.fit.cmsn.cross_pp.lab1.sample1;
2
3  import java.util.Scanner;
```

Директива **import** указывается в самом начале кода, после чего идет имя подключаемого класса (в данном случае класса Scanner).

В примере выше мы подключили только один класс, однако пакет java.util содержит еще множество классов, и чтобы не подключать по отдельности каждый класс, мы можем сразу подключить весь пакет:

```
1  import java.util.*; // импорт всех классов из пакета java.util
```

Теперь мы можем использовать любой класс из пакета java.util.

Возможна ситуация, когда мы используем два класса с одним и тем же названием из двух разных пакетов, например, класс Date имеется и в пакете java.util, и в пакете java.sql. И если нам надо одновременно использовать два этих класса, то необходимо указывать полный путь к этим классам в пакете:

```
java.util.Date utilDate = new java.util.Date();
java.sql.Date sqlDate = new java.sql.Date();
```

**Статический импорт**

В Java есть также особая форма импорта – статический импорт. Для этого вместе с директивой **import** используется модификатор **static,** например:

```
package by.psu.fit.cmsn.cross_pp.lab1.sample1;

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

public class FirstApp
{
    public static void main(String[] args)
    {
        double result = sqrt(20);
        out.println(result);
    }
}
```

Здесь происходит статический импорт классов System и Math. Эти классы имеют статические методы. Благодаря операции статического импорта, мы можем использовать эти методы без названия класса. Например, писать не Math.sqrt(20), а sqrt(20), так как функция sqrt(), которая возвращает квадратный корень числа, является статической. Позже мы рассмотрим статические члены класса.

То же самое в отношении класса System: в нем определен статический объект out, поэтому мы можем его использовать без указания класса.

**Модификаторы доступа и инкапсуляция**

Все члены класса в языке Java – поля и методы, свойства – имеют модификаторы доступа. Модификаторы доступа позволяют задать допустимую область видимости для членов класса, то есть контекст, в котором можно употреблять данную переменную или метод (рисунок 17).

При этом:

– **private** члены класса доступны только внутри класса;

– **package–private** или **default** (по умолчанию) члены класса видны внутри пакета. Отсутствие модификатора у поля или метода класса предполагает применение к нему модификатора по умолчанию. Такие поля или методы видны всем классам в текущем пакете;

– **protected** члены класса доступны внутри пакета и в классах-наследниках;

– **public** члены класса доступны всем.

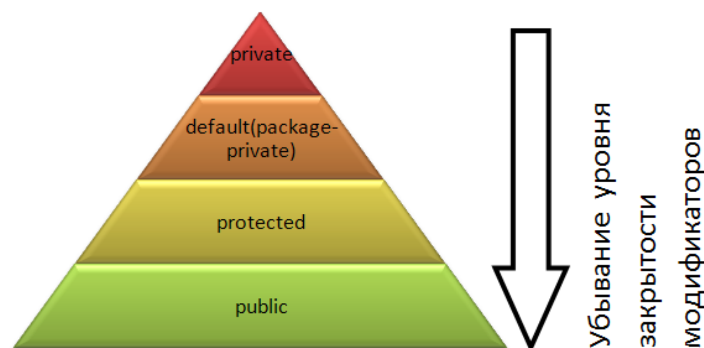


**Рисунок 17. – Модификаторы доступа**

Рассмотрим все возможные случаи. Допустим, у нас есть проект со следующей структурой (рисунок 18).

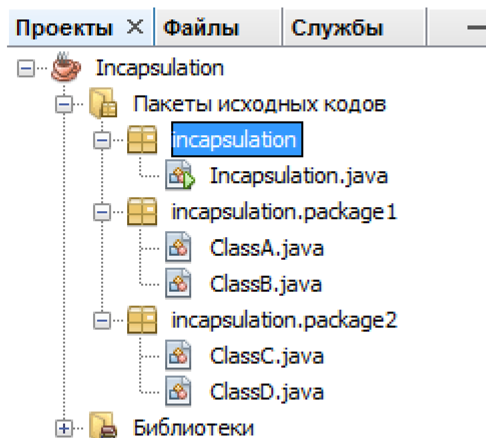


**Рисунок 18. – Представление структуры пакета**

В данном случае имеется три пакета. Пакет incapsulation содержит главный класс программы, в котором и прописан метод main. Его мы трогать не будем. И также определены два пакета: incapsulation.package1, который содержит классы ClassA и ClassB, и incapsulation.package2, который содержит классы ClassC и ClassD.

Пример: класс ClassA содержит весь возможный набор модификаторов доступа:

```
1  package incapsulation.package1;
2
3  public class ClassA
4  {
5      private int num1=3;
6      int num2 = 2;
7      protected int num3 =3;
8      public int num4=4;
9
10     public void displayNum1(){
11         System.out.println(num1);
12     }
13
14     protected void displayNum2(){
15         System.out.println(num2);
16     }
17
18     void displayNum3(){
19         System.out.println(num3);
20     }
21
22     private void displayNum4(){
23         System.out.println(num4);
24     }
25 }
```

Модификатор доступа должен предшествовать остальной части определения переменной или метода.

Пусть класс ClassB использует ClassA в следующем фрагменте кода:

```
1  package incapsulation.package1;
2
3  public class ClassB
4  {
5      public void result()
6      {
7          ClassA classA = new ClassA();
8          //System.out.println(classA.num1); // ошибка, так как num1 - private
9          classA.displayNum1(); // public
10
11         System.out.println(classA.num2); // идентификатор по умолчанию
12         classA.displayNum2(); // protected
13
14         System.out.println(classA.num3); // protected
15         classA.displayNum3(); // идентификатор по умолчанию
16
17         System.out.println(classA.num4); // public
18         //classA.displayNum4(); // ошибка, так как displayNum4() - private
19     }
20 }
```

Так как классы ClassB и ClassA находятся в одном пакете, то мы сможем использовать все поля и методы класса ClassA в ClassB кроме тех, что объявлены как private.

Теперь используем класс ClassA в классе ClassC:

```
package incapsulation.package2;

import incapsulation.package1.ClassA;

public class ClassC
{
    public void result()
    {

        ClassA classA = new ClassA();
        //System.out.println(classA.num1); // ошибка, так как num1 - private
        classA.displayNum1(); // public

        //System.out.println(classA.num2); // ошибка, так как num2 - идентификатор по умолчанию
        //classA.displayNum2(); //ошибка, так как доступ - protected

        //System.out.println(classA.num3); // ошибка, так как доступ - protected
        //classA.displayNum3(); //ошибка, так как доступ - идентификатор по умолчанию

        System.out.println(classA.num4); // public
        //classA.displayNum4(); // ошибка, так как displayNum4() - private
    }
}
```

Так как класс ClassC находится в другом пакете и не является наследником класса ClassA, то в нем можно использовать только те поля и методы класса ClassA, которые объявлены с модификатором public.

И последний случай – ClassD является наследником класса ClassA, но находится в другом пакете (чуть позже мы более подробно разберем наследование):

```
package incapsulation.package2;

import incapsulation.package1.ClassA;

public class ClassD extends ClassA {

    public void result(){

        //System.out.println(num1); // ошибка, так как num1 - private
        displayNum1(); // public

        //System.out.println(num2); // ошибка, так как доступ - идентификатор по умолчанию
        displayNum2(); // protected

        System.out.println(num3); // protected
        //displayNum3();  //ошибка, так как доступ по умолчанию

        System.out.println(num4); // public
        //classA.displayNum4(); // ошибка, так как displayNum4() - private
    }
}
```

Так как ClassD – наследник класса ClassA, то мы можем напрямую использовать методы и поля ClassA без создания объекта. Здесь нам опять недоступны поля и методы private, и также нам недоступны поля и методы с модификатором доступа по умолчанию. В то же время в отличие от находящегося в том же пакете класса ClassC в классе ClassD мы можем использовать методы и поля с доступом protected.

### Инкапсуляция

Почему бы не объявить все переменные и методы с модификатором **public**? Однако использование различных модификаторов гарантирует, что данные не будут искажены или изменены не надлежащим образом. Подобное сокрытие данных называется **инкапсуляцией**.

Вместо непосредственного использования полей, как правило, используют методы доступа, например:

```
class Book{

    private String name;

    public void setName(String name){

        this.name=name;
    }

    public String getName(){

        return name;
    }
}
```

И затем вместо непосредственной работы с полем name в классе Book мы будем работать с методами, которые устанавливают значение этого поля и возвращают его.

### Модификатор final

Модификатор **final** может применяться к классам, методам и переменным. При этом его конкретное значение зависит от контекста. Но в общем модификатор final означает, что элемент не может быть модифицирован, изменен.

В частности, final класс не может иметь подклассов. Например, следующий код просто не откомпилируется, потому что java.lang.Math имеет модификатор final:

```
class SubMath extends java.lang.Math { }
```

Получаем ошибку компиляции: "Can’t subclass final classes." («Невозможно создать подкласс от класса final»): final переменная не может принимать значение, которое отличается от того, что было присвоено при инициализации. Поэтому в Java переменные с модификатором final играют роль const в C++ и #define в C. Например, в классе java.lang.Math есть переменная с модификатором final типа double, которая называется PI. Очевидно, что число пи не должно изменяться по прихоти программиста.

Если объекту назначен модификатор final, то это означает, что ссылка (указатель) на объект должна оставаться неизменной, хотя сам объект может меняться. Например:

```
1  class Walrus
2  {
3      int weight;
4
5      Walrus(int w)
6      {
7          weight = w;
8      }
9  }
10
11 class Tester
12 {
13     final Walrus w1 = new Walrus(1500);
14
15     void test()
16     {
17         //w1 = new Walrus(1400); // ошибка
18         w1.weight = 1800;
19     }
20 }
```

Обратите внимание, как в строке 13 объект w1 был объявлен с модификатором **final**. Именно поэтому в строке 17 выполняется недопустимая операция создания нового указателя на объект. Хотя в строке 18 мы можем изменить значение некоторых полей объекта.

Наконец, final метод не может быть переопределен (overriden). Например, следующий код не откомпилируется:

```
1. class Mammal {
2.     final void getAround() { }
3. }
4.
5. class Dolphin extends Mammal {
6.     void getAround() { }
7. }
```

Итого для модификатора final справедливо:

– поля не могут быть изменены, методы переопределены;

– классы нельзя наследовать;

– этот модификатор применяется только к классам, методам и переменным (также и к локальным переменным);

– аргументы методов, обозначенные как final, предназначены только для чтения, при попытке изменения будет ошибка компиляции;

– переменные final не инициализируются по умолчанию, им необходимо явно присвоить значение при объявлении или в конструкторе, иначе – ошибка компиляции;

– если final переменная содержит ссылку на объект, объект может быть изменен, но переменная всегда будет ссылаться на тот же самый объект;

– также это справедливо и для массивов, потому что массивы являются объектами, – массив может быть изменен, а переменная всегда будет ссылаться на тот же самый массив;

– если класс объявлен final и abstract (взаимоисключающие понятия), произойдет ошибка компиляции;

– так как final класс не может наследоваться, его методы никогда не могут быть переопределены.

### Статические члены и модификатор static

Кроме обычных методов и полей класс может иметь статические поля и методы. Например, главный класс программы имеет метод main, который является статическим:

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

}
```

Для объявления статических переменных и методов перед их объявлением указывается ключевое слово static. Статические члены класса могут использоваться без создания объектов класса.

Ранее мы уже использовали статические переменные. В частности, в классе System содержится статическая переменная out, с помощью которой выводятся данные на консоль. Например, создадим статическую переменную:

```
class Book{

    private int id;
    private static int counter=1;

    public void displayId(){

        System.out.printf("Id: %d \n", id);
    }

    pivate String author;
    private int year;
    private String name;

    Book(String name, String author, int year){

        this.name = name;
        this.author = author;
        this.year = year;
        id=counter++;
    }
}
```

Класс Book содержит статическую переменную counter, которая увеличивается в конструкторе, и ее значение присваивается переменной id.

После этого мы можем создать несколько объектов Book, и в каждом вызове конструктора переменная counter будет увеличиваться на единицу, так как она относится не к конкретному объекту, а ко всему классу Book в целом или всем объектам Book сразу:

```
Book b1 = new Book("Война и мир", "Л. Н. Толстой", 1863);
b1.displayId(); // выведет Id: 1

Book b2 = new Book("Отцы и дети", "И. Тургенев", 1862);
b2.displayId(); // выведет Id: 2
```

Хотя в примере выше мы сразу инициализировали статическую переменную, но нередко для инициализации статических полей применяется статический блок. Этот блок вызывается один раз в программе при создании первого объекта данного класса:

```
class Book{

    private int id;
    private static int counter;

    static {

        counter=1;
        System.out.println("Вызов статического блока");
    }
    // остальной код класса
}
```

Нередко константы на уровне класса объявляют как статические:

```
public final static double PI = 3.14;
```

Статические методы, подобно статическим переменным, также относятся ко всему классу. Например, создадим новый класс Algorithm и добавим в него две функции для вычисления числа Фибоначчи и факториала:

```
class Algorithm
{
    public final static double PI = 3.14;

    public static int factorial(int x)
    {
        if (x == 1)
            return 1;
        else
            return x * factorial(x - 1); //рекурсия
    }

    public static int fibonachi(int x)
    {
        if (x == 0)
            return 1;

        if (x == 1)
            return 1;
        else
            return fibonachi(x - 1) + fibonachi(x - 2); //рекурсия
    }
}
```

Теперь используем их в программе:

```
1  public static void main(String[] args)
2  {
3      int num1 = Algorithm.factorial(5);
4      int num2 = Algorithm.fibonachi(5);
5
6      System.out.println(Algorithm.PI);
7  }
```

И поскольку методы factorial и fibonachi, а также поле PI являются статическими, то мы можем к ним обратиться напрямую без создания объекта класса: Algorithm.factorial(5).

Итого для модификатора **static** справедливо:

– применяется к внутренним классам, методам, переменным и логическим блокам;

– статические переменные инициализируются во время загрузки класса;

– статические переменные едины для всех объектов класса (одинаковая ссылка);

– статические методы имеют доступ только к статическим переменным;

– к статическим методам и переменным можно обращаться через имя класса;

– статические блоки выполняются во время загрузки класса;

– не static методы не могут быть переопределены как static;

– локальные переменные не могут быть объявлены как static;

– абстрактные методы не могут быть static;

– static поля не сериализуются (только при реализации интерфейса Serializable);

– только static переменные класса могут быть переданы в конструктор с параметрами, вызывающийся через слово super(//параметр//) или this(//параметр//).

## ИНДИВИДУАЛЬНЫЕ ВАРИАНТЫ ЗАДАНИЙ

Таблица 1. – Индивидуальные варианты заданий к лабораторной работе № 2

| Номер варианта | Сущность, которую необходимо представить в виде класса |
|---|---|
| 1 | 2 |
| 1 | СТУДЕНТ |
| 2 | СЛУЖАЩИЙ |
| 3 | КАДРЫ |
| 4 | ИЗДЕЛИЕ |

Окончание таблицы 1

| 1 | 2 |
|---|---|
| 5 | БИБЛИОТЕКА |
| 6 | ЭКЗАМЕН |
| 7 | АДРЕС |
| 8 | ТОВАР |
| 9 | КВИТАНЦИЯ |
| 10 | ЦЕХ |
| 11 | ПЕРСОНА |
| 12 | АВТОМОБИЛЬ |
| 13 | СТРАНА |
| 14 | ЖИВОТНОЕ |
| 15 | КОРАБЛЬ |
| 16 | КОМПЬЮТЕР |
| 17 | КАРТИНА |
| 18 | ЖУРНАЛ |
| 19 | ТЕЛЕВИЗОР |
| 20 | ИГРУШКА |

## ЗАДАНИЕ К ЛАБОРАТОРНОЙ РАБОТЕ

1. Ознакомиться с теоретической частью.

2. Реализовать класс, описывающий сущность согласно индивидуальному варианту, и учесть следующие требования:

– в классе должно быть не менее 3 полей;

– минимум 2 поля должны быть строкового типа;

– в классе обязательно должно быть предусмотрено 2 конструктора с параметрами и конструктор по умолчанию;

– доступ к членам класса должен осуществляться через методы get и set, а сами члены класса скрыты с помощью модификаторов доступа;

– реализовать подсчет созданных объектов класса с помощью статической переменной;

– по умолчанию переменные класса должны инициализироваться с помощью констант, определенных в данном классе;

– создать минимум 2 функции с несколькими параметрами (помимо геттеров и сеттеров) для манипуляции внутренними свойствами объекта;

– реализовать метод для вывода информации об объекте класса;

– в главном классе продемонстрировать работу всех методов класса, используя статический метод (помимо main).

## КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Зачем нужны и какие бывают блоки инициализации?
2. Как влияет модификатор static на класс/метод/поле?
3. О чем говорит ключевое слово final?
4. Можно ли получить доступ к private переменным класса и если да, то каким образом?
5. Какой результат выполнения и/или компиляции следующего участка кода:

```
1   class A
2   {
3       static int a = new A().getNewA();
4
5       int getNewA()
6       {
7           return ++a;
8       }
9
10      {
11          a++;
12      }
13  }
14
15  public class MainClass
16  {
17      public static void main(String[] args)
18      {
19          System.out.println(new A().a);
20      }
21  }
```

## СОДЕРЖАНИЕ ОТЧЕТА

1. Ф.И.О., группа, название лабораторной работы.
2. Цель работы.
3. Описание проделанной работы.
4. Результаты выполнения лабораторной работы.
5. Выводы.

Отчет и исходные коды запаковать в архив с названием по следующему шаблону:

Lab1–«группа, аббревиатура на латинице»–«Фамилия на латинице».

Пример: Lab2–13VS–Ivanov.zip.

### Порядок защиты лабораторной работы

1. Предоставление отчета о выполнении лабораторной работы.
2. Демонстрация программы.
3. Выполнение дополнительного задания.
4. Ответы на вопросы по теме текущей лабораторной работы.

## Лабораторная работа № 3. Наследование

**Цель работы.** Изучить механизм наследования классов в языке программирования Java. Познакомиться с понятиями интерфейсов и классов.

Видеоматериал по лекции находится в файле кпп_лр3.wmv.

### Теоретическая часть

**Наследование**

Наследование – это не что иное, как приобретение свойства другого объекта или класса. Наследование в Java осуществлено отношениями подкласса и суперкласса. **Суперкласс** – это класс, от которого будет происходить наследование, а **подкласс** – это класс, который наследовал качества суперкласса. Когда наследование осуществлено, подкласс получает свойства суперкласса плюс его собственные свойства. Например, рассмотрите классы Транспортное средство и Скутер. Свойства класса Транспортное средство – скорость, вместимость, а свойства класса Скутер – вместимость бензина, механизм. Класс Скутер унаследован от класса Транспортное средство. Тогда класс Транспортное средство становится суперклассом, а класс Скутер становится подклассом. Класс Скутер получает свойства (скорость, вместимость) класса Транспортное средство, плюс он имеет свои собственные свойства (вместимость бензина, механизм). Таким образом, подкласс содержит собственные свойства, так же, как и его суперкласс.

Например, имеется следующий класс Person, описывающий отдельного человека:

```
1  public class Person {
2
3      private String name;
4      private String surname;
5
6      public String getName() { return name; }
7      public String getSurname() { return surname; }
8
9      public Person(String name, String surname){
10
11         this.name=name;
12         this.surname=surname;
13     }
14
15     public void displayInfo(){
16
17         System.out.println("Имя: " + name + " Фамилия: " + surname);
18     }
19 }
```

И, возможно, впоследствии мы решили расширить имеющуюся систему классов, добавив в нее класс, описывающий сотрудника предприятия – класс Employee. Так как этот класс реализует тот же функционал, что и класс Person, поскольку сотрудник – это также и человек, то было бы рационально сделать класс Employee производным (или наследником) от класса Person, который, в свою очередь, называется базовым классом или родителем:

```
class Employee extends Person{

}
```

Чтобы объявить один класс наследником от другого, надо использовать после имени класса-наследника ключевое слово **extends**, после которого идет имя базового класса. Для класса Employee базовым является Person, и поэтому класс Employee наследует все те же поля и методы, которые есть в классе Person.

В классе Employee могут быть определены свои методы и поля, а также конструктор:

```
class Employee extends Person{

    private String company;

    public Employee(String name, String surname, String company) {

        super(name, surname);
        this.company=company;
    }

    public void displayInfo(){

        super.displayInfo();
        System.out.println("Компания: " + company);
    }
}
```

Класс Employee определяет дополнительное поле для хранения компании, в которой работает сотрудник. Кроме того, оно также устанавливается в конструкторе.

Так как поля name и surname в базовом классе Person объявлены с модификатором private, то мы не можем к ним напрямую обратиться из класса Employee. Однако в данном случае нам это не нужно. Чтобы их установить, мы обращаемся к конструктору базового класса с помощью ключевого слова **super**, в скобках после которого идет перечисление передаваемых аргументов.

С помощью ключевого слова super мы можем обратиться к любому члену базового класса – методу или полю, если они не определены с модификатором **private**.

Также в классе Employee переопределяется метод displayInfo() базового класса. В нем с помощью ключевого слова super также идет обращение к методу displayInfo(), но уже базового класса, и затем выводится дополнительная информация, относящаяся только к Employee.

Используя обращение к методам базового класса, можно было бы переопределить метод displayInfo() следующим образом:

```
public void displayInfo(){

    System.out.println("Имя: " + super.getName() + " Фамилия: "
        + super.getSurname() + " Компания: " + company);
}
```

**Запрет наследования**

Хотя наследование очень интересный и эффективный механизм, но в некоторых ситуациях его применение может быть нежелательным. И в этом случае можно запретить наследование с помощью ключевого слова **final**. Например:

```
public final class Person {
}
```

Кроме запрета наследования можно также запретить переопределение отдельных методов. Например, в примере выше переопределен метод displayInfo(), запретим его переопределение:

```
public class Person {

    //........................

    public final void displayInfo(){

        System.out.println("Имя: " + name + " Фамилия: " + surname);
    }
}
```

В этом случае в классе Employee надо будет создать метод с другим именем для вывода информации об объекте.

### Множественное и одиночное наследование

Такие языки, как C++, поддерживают концепцию множественного наследования: в любой точке иерархии класс может наследовать свойства одного или нескольких классов. Язык Java поддерживает **только одиночное наследование**, то есть ключевое слово **extends** можно использовать только с одним классом. Так что иерархия классов для любого заданного класса Java всегда состоит из прямой, охватывающей весь путь вплоть до **java.lang.Object.**

Тем не менее, язык Java поддерживает реализацию нескольких интерфейсов в одном классе, что дает своего рода обходной путь для одиночного наследования.

### Конструкторы и наследование

Чтобы вызвать конструктор суперкласса, отличный от конструктора по умолчанию, нужно сделать это явно. Первое, что делает конструктор наследника – вызывает конструктор по умолчанию своего непосредственного суперкласса, если только вы в первой строке кода конструктора не вызывали другой конструктор. Например, следующие две декларации функционально идентичны, так что выберите любую:

```
1  public class Person
2  {
3      public Person()
4      {
5      }
6  }
7
8  // Между тем, в Employee.java
9  public class Employee extends Person
10 {
11     public Employee()
12     {
13     }
14 }
```

или

```
1  public class Person
2  {
3      public Person()
4      {
5      }
6  }
7
8  // Между тем, в Employee.java
9  public class Employee extends Person
10 {
11     public Employee()
12     {
13         super();
14     }
15 }
```

### Конструкторы без аргументов

Создавая альтернативный конструктор, мы должны явно создать конструктор по умолчанию, иначе он будет недоступен. Например, следующий код вызовет ошибку компиляции:

```
public class Person
{
    private String name;

    public Person(String name)
    {
        this.name = name;
    }
}

// Между тем, в Employee.java
public class Employee extends Person
{
    public Employee() {}
}
```

В этом примере нет конструктора по умолчанию, потому что он содержит альтернативный конструктор без включения явного конструктора по умолчанию. Поэтому конструктор по умолчанию иногда называют конструктором без аргументов (no-arg); поскольку существуют условия, при которых он не включается, это не совсем конструктор по умолчанию.

### Как конструкторы вызывают конструкторы

Конструктор может быть вызван другим конструктором изнутри класса с использованием ключевого слова **this** и списка аргументов. Как и **super()**, вызов **this()** должен быть первой строкой конструктора. Например:

```
public class Person
{
    private String name;

    public Person()
    {
        this("Some reasonable default?");
    }

    public Person(String name)
    {
        this.name = name;
    }
}
```

Мы будем часто встречать это выражение, когда один конструктор делегирует свои полномочия другому, передавая ему некоторое значение по умолчанию при вызове этого конструктора. Это также отличный способ добавить новый конструктор к классу при минимальном влиянии на код, который уже использует старый конструктор.

### Инициализация полей при наследовании классов

Уточним определение инициализации объектов в случае наследования классов.

Все действия по инициализации выполняются этап за этапом в порядке наследования классов:

– при первом обращении к классу выделяется память под статические поля класса и выполняется их инициализация;
– выполняется распределение памяти под создаваемый объект;
– выполняются все инициализаторы нестатических полей класса;
– выполняется вызов конструктора класса.

Рассмотрим абстрактный пример:

```
1  class A {
2      ...
3      A() {...}
4      ...
5  }
6
7  class B extends A {
8      ...
9      B() { ... }
10     ...
11 }
```

При создании экземпляра объекта B (Bb = newB();) сначала выполнится конструктор A(), потом конструктор B().

### Наследование и абстракция

В контексте объектно-ориентированного программирования **абстрагирование** означает обобщение данных и поведения до типа, более высокого по иерархии наследования, чем текущий класс. При перемещении переменных или методов из подкласса в суперкласс говорят, что эти члены *абстрагируются*. Основной причиной этого является возможность многократного использования общего кода путем продвижения его как можно выше по иерархии. Когда общий код собран в одном месте, облегчается его обслуживание.

### Абстрактные классы и методы

Бывают моменты, когда нужно создавать классы, которые служат только как абстракции, и создавать их экземпляры, быть может, никогда не придется. Такие классы называются абстрактными классами. К тому же мы обнаружим, что бывают моменты, когда определенные методы должны быть реализованы по-разному для каждого подкласса, реализуемого суперклассом. Это абстрактные методы. Вот некоторые основные правила для абстрактных классов и методов:

– любой класс может быть объявлен абстрактным;

– абстрактные классы не допускают создания своих экземпляров;
– абстрактный метод не может содержать тела метода;
– класс, содержащий абстрактный метод, должен объявляться как абстрактный.

Предположим, что мы не хотим допустить, чтобы экземпляры класса Employee можно было создавать напрямую. Для этого остаточно объявить его с помощью ключевого слова **abstract**:

```
1 public abstract class Employee extends Person
2 {
3     //...
4 }
```

Кроме обычных методов абстрактный класс может содержать абстрактные методы. Такие методы определяются с помощью ключевого слова **abstract** и не имеют никакого функционала.

Производный класс обязан переопределить и реализовать все абстрактные методы, которые имеются в базовом абстрактном классе. Также следует учитывать, что если класс имеет хотя бы один абстрактный метод, то данный класс должен быть определен как абстрактный.

Допустим, мы делаем программу для обслуживания банковских операций и определяем в ней три класса: Person, который описывает человека, Employee, который описывает банковского служащего, и класс Client, который представляет клиента банка. Очевидно, что классы Employee и Client будут производными от класса Person, так как оба класса имеют некоторые общие поля и методы. И так как все объекты будут представлять либо сотрудника, либо клиента банка, то напрямую мы от класса Person создавать объекты не будем. Поэтому имеет смысл сделать его абстрактным, например:

```
1  abstract class Person
2  {
3      private String name;
4      private String surname;
5
6      public String getName()
7      {
8          return name;
9      }
10
11     public String getSurname()
12     {
13         return surname;
14     }
15
16     public Person(String name, String surname)
17     {
18         this.name = name;
19         this.surname = surname;
20     }
21
22     public abstract void displayInfo();
23 }
```

```
25  class Employee extends Person
26  {
27
28      private String bank;
29
30      public Employee(String name, String surname, String company)
31      {
32          super(name, surname);
33          this.bank = company;
34      }
35
36      public void displayInfo()
37      {
38          System.out.println("Имя: " + super.getName() + " Фамилия: "
39                  + super.getSurname() + " Работает в банке: " + bank);
40      }
41  }
43  class Client extends Person
44  {
45      private String bank;
46
47      public Client(String name, String surname, String company)
48      {
49          super(name, surname);
50          this.bank = company;
51      }
52
53      public void displayInfo()
54      {
55          System.out.println("Имя: " + super.getName() + " Фамилия: "
56                  + super.getSurname() + " Клиент банка: " + bank);
57      }
58  }
```

Другим хрестоматийным примером являются системы фигур. В реальности не существует геометрической фигуры как таковой. Есть круг, прямоугольник, квадрат, но просто фигуры нет. Однако же и круг, и прямоугольник имеют что-то общее и являются фигурами:

```
1   //абстрактный класс фигуры
2   public abstract class Figure
3   {
4       float x; // x-координата точки
5       float y; // y-координата точки
6
7       public Figure(float x, float y)
8       {
9           this.x=x;
10          this.y=y;
11      }
12
13      // абстрактный метод для получения периметра
14      public abstract float getPerimeter();
15
16      // абстрактный метод для получения площади
17      public abstract float getArea();
18  }
20  // производный класс прямоугольника
21  class Rectangle extends Figure
22  {
23      private float width;
24      private float height;
25
26      // конструктор с обращением к конструктору класса Figure
27      public Rectangle(float x, float y, float width, float height)
28      {
29          super(x,y);
30          this.width = width;
31          this.height = height;
32      }
33
34      public float getPerimeter()
35      {
36          return width * 2 + height * 2;
37      }
38
39      public float getArea()
40      {
41          return width * height;
42      }
43  }
```

**Когда (не) следует абстрагировать: два правила**

Первое правило – **не применяйте абстракции при первоначальной разработке**. Использование абстрактных классов на ранней стадии проектирования вынуждает следовать определенному пути и может сделать ваше приложение ограниченным. Помните, что общее поведение (в котором заключается суть абстрактных классов) всегда можно организовать выше по графу наследования. Почти всегда это лучше делать только тогда, когда возникнет такая необходимость.

Во-вторых, какими бы мощными они ни были, по мере возможности **сопротивляйтесь использованию абстрактных классов**. Пока в поведении суперклассов не будет много общего, а сами они не будут малозначимыми, пусть остаются неабстрактными. Глубокие графы наследования могут затруднить поддержку кода. Ищите компромисс между слишком большими классами и кодом, удобным для поддержки.

**Присвоения: классы**

Присвоение ссылки из одного класса на переменную типа, принадлежащую к другому классу, допускается, но существуют определенные правила. Рассмотрим следующий пример:

```
class Person
{
    ...
}

abstract class Employee extends Person
{
    ...
}

class Manager extends Employee
{
    ...
}

Manager m = new Manager();
Employee e = new Employee();
Person p = m; // нормально
p = e; // тоже нормально
Employee e2 = e; // да, так можно
e = m; // тоже нормально
e2 = p; // Неверно!
```

Вызываемая переменная должна быть супертипом класса, принадлежащего к источнику ссылки, иначе компилятор выдаст сообщение об ошибке. В принципе, все, что справа от присвоения, должно быть подклассом или тем же классом, того, что слева. В противном случае при присвоении объектов с разными графами наследования (например, Manager и Employee) можно присвоить переменную недопустимого типа.

**Оператор instanceof**

Используя оператор **instanceof**, можно узнать, от какого класса произошел объект. Этот оператор имеет два аргумента. Слева указывается ссылка на объект, а справа – имя типа, на совместимость с которым проверяется объект.

Правый операнд также может быть и названием интерфейса. И в этом случае оператор определяет, имплементирует ли левый аргумент данный интерфейс.

Опертор также используется для тестирования, является ли объект массивом. Поскольку массивы в Java являются объектами, то такое тестирование логично. Но тест проводится в два шага: (1) является ли объект массивом и (2) является ли тип элементов массива (под)классом правого аргумента. Это отражает идею, что массив, скажем, объектов типа Button (кнопки) является массивом объектов типа Component (компоненты), поскольку Button – это Component. Тест для такого массива будет выглядеть так:

**if (x instanceof Component[])**

Заметьте, что невозможно провести тестирование для «любого массива с элементами любого типа», то есть следующая строка недопустима:

**if (x instanceof [])**

Недопустимо также тестирование на массив с элементами типа Object:

**if (x instanceof Object [])**

поскольку элементы массива могут быть примитивного типа и тест попросту не сработает.

**Если левый операнд равен null, то instanceof возвратит false. Никакого исключения не возникнет.** Пример:

```
1  class Student
2  {
3  // some code
4  }
5
6  class Graduate extends Student
7  {
8  //some code
9  }
10
11 class UndergraduateStudent extends Student
12 {
13 // some code
14 }
15
16 public class st
17 {
18     public static void main(String []args)
19     {
20         Student x = new UndergraduateStudent();
21
22         if (x instanceof UndergraduateStudent)
23             System.out.println("Mr. X is an undergraduate student.");
24         else
25             System.out.println("Mr. X is an graduate student.");
26     }
27 }
```

В вышеупомянутом примере инструкция if проверяет, объект x – instanceof класс UndergraduateStudent. В этом случае условие оценивается как истина и тогда печатается "Mr. x is an undergraduatestudent."

**Класс Object**

В Java фактически все классы строятся на базе наследования, так как даже если не написать «extends имя_класса», будет подразумеваться extends Object, **т.е. класс Object является базовым классом для всех классов Java.**

Класс Object имеет ряд методов, при наследовании методы базового класса наследуются его потомками. Следовательно, все классы Java имеют как минимум те методы, которые есть в классе Object.

Взглянем на документацию по классу Object. В классе Object определены методы, большая часть из которых имеют непонятное для нас (на текущий момент) назначение. Но некоторые из них мы можем уже сейчас рассмотреть:

- **public String toString()** – предназначен для формирования некоторого текстового представления объекта;
- **protected Object clone() throws CloneNotSupportedException** – предназначен для создания копии объекта;
- **public Boolean equals(Object obj)** – позволяет сравнивать объекты;
- **public final Class getClass()** – выдает класс данного объекта в виде объекта класса Class;
- **protected void finalize() throws Throwable** – этот метод вызывается при удалении объекта из памяти сборщиком мусора.

Все указанные методы, кроме getClass и, частично, toString, не предназначены для непосредственного использования (в том смысле, что их нужно переопределить в порожденных классах).

Метод getClass возвращает специальный объект класса Class, содержащий много полезной информации о классе объекта. Метод toString по умолчанию выдает полное имя класса объекта и его адрес, что можно использовать при отладке программы.

Если почитать документацию по остальным методам, то мы увидим, что она, в основном, определяет, что должен делать тот или иной метод и лишь в конце описывается, как этот метод реализован в классе Object.

Разберем, например, метод equals. Его назначение – сравнивать объекты на равенство. Наличие его в классе Object (базовом для всех остальных классов) позволяет нам применять этот метод для любых объектов, что очень удобно. Но в классе Object он реализован «самым жестким» образом – как сравнение адресов объектов, т.е. при сравнении двух объектов мы получим равенство только в том случае, если на самом деле это один и тот же объект. Естественно, что чаще всего нам потребуется какая-то другая реализация данного метода.

**Интерфейсы**

Интерфейс – это именованный набор моделей поведения (и/или постоянных элементов данных), который должен обеспечить реализатор. Интерфейс определяет поведение реализации, но не то, каким образом оно достигается.

Определить интерфейс легко:

```
public interface interfaceName
{
    returnType methodName( argumentList );
}
```

Объявление интерфейса выглядит как объявление класса, за исключением того, что при этом используется ключевое слово **interface**. Интерфейс можно назвать как угодно (в соответствии с правилами языка), но, по соглашению, имена интерфейсов выглядят так же, как имена классов.

Методы, определенные в интерфейсе, не имеют тела метода. За предоставление тела метода отвечает реализатор интерфейса (как в случае с абстрактными методами).

Иерархии интерфейсов определяются как для классов, за исключением того, что один класс может реализовать сколько угодно интерфейсов. Класс, как мы помним, может расширять только один класс. Если один класс расширяет другой и реализует интерфейс(ы), то эти интерфейсы перечисляются после расширенного класса, вот так:

```
1  public class Manager extends Employee implements BonusEligible, StockOptionRecipient
2  {
3      // и т.д.
4  }
```

**Все методы интерфейса не имеют модификаторов доступа**, но фактически по умолчанию доступ public, так как цель интерфейса – определение

функционала для реализации его классом. Поэтому весь функционал должен быть открыт для реализации.

И также при объявлении интерфейса надо учитывать, что только один интерфейс в файле может иметь тип доступа public. А его название должно совпадать с именем файла. Остальные интерфейсы (если такие имеются в файле java) не должны иметь модификаторов доступа.

Интерфейс может вообще не иметь никакого тела. На самом деле, вполне приемлемо следующее определение:

```
1  public interface BookPrintable
2  {
3  }
```

Вообще говоря, такие интерфейсы называются **интерфейсами-маркерами** (marker interfaces), потому что они маркируют класс как реализацию этого интерфейса, но не предлагают особого явного поведения. Пример:

```
1  interface Printable
2  {
3      void print();
4  }
5
6  class Book implements Printable
7  {
8      String name;
9      String author;
10     int year;
11
12     Book(String name, String author, int year)
13     {
14         this.name = name;
15         this.author = author;
16         this.year = year;
17     }
18
19     public void print()
20     {
21         System.out.printf("Книга '%s' (автор %s) была издана в %d году \n", name, author, year);
22     }
23 }
```

При этом надо учитывать, что если класс применяет интерфейс, то он должен реализовать все методы интерфейса, как в случае выше реализован метод print.

Потом в главном классе мы можем использовать данный класс и его метод print:

```
1  Book b1 = new Book("Война и мир", "Л. Н. Толстой", 1863);
2  b1.print();
```

В то же время, мы не можем напрямую создавать объекты интерфейсов, как и абстрактных классов.

Интерфейсы, как и классы, могут наследоваться:

```
interface BookPrintable extends Printable
{
    void paint();
}
```

При применении этого интерфейса класс Book должен будет реализовать как методы интерфейса BookPrintable, так и методы базового интерфейса Printable.

Кроме методов в интерфейсах могут быть определены статические константы:

```
interface Printable
{
    int MAX_NUMBER = 400;

    void print();
}
```

**Хотя такие константы также не имеют модификаторов**, но по умолчанию они имеют модификатор доступа **public static final**, и поэтому **их значение доступно из любого места программы**.

## ИНДИВИДУАЛЬНЫЕ ВАРИАНТЫ ЗАДАНИЙ

Таблица 2. – Индивидуальные варианты заданий к лабораторной работе № 3

| Номер варианта | Сущность, которую необходимо представить в виде класса |
|---|---|
| 1 | 2 |
| 1 | студент, преподаватель, персона, заведующий кафедрой |
| 2 | служащий, персона, рабочий, инженер |
| 3 | рабочий, кадры, инженер, администрация |
| 4 | деталь, механизм, изделие, узел |
| 5 | организация, страховая компания, судостроительная компания, завод |
| 6 | журнал, книга, печатное издание, учебник |
| 7 | тест, экзамен, выпускной экзамен, испытание |
| 8 | место, область, город, мегаполис |
| 9 | игрушка, продукт, товар, молочный продукт |
| 10 | квитанция, накладная, документ, чек |
| 11 | автомобиль, поезд, транспортное средство, экспресс |
| 12 | двигатель, двигатель внутреннего сгорания, дизель, турбореактивный двигатель |
| 13 | республика, монархия, королевство, государство |

Окончание таблицы 2

| 1 | 2 |
|---|---|
| 14 | млекопитающие, парнокопытные, птицы, животное |
| 15 | корабль, пароход, парусник, корвет |
| 16 | компьютер, ноутбук, сервер, планшет |
| 17 | произведение искусства, картина, фреска, эскиз |
| 18 | носитель данных, жесткий диск, флэшка, SSD-диск |
| 19 | товар, телевизор, медиа-приставка, игровая приставка |
| 20 | программа, компьютерная игра, комп. вирус, троян |

## ЗАДАНИЕ К ЛАБОРАТОРНОЙ РАБОТЕ

1. Ознакомиться с теоретической частью.
2. Реализовать иерархию классов и интерфейсов, описывающих набор сущностей согласно индивидуальному варианту и учесть следующие требования:

– в иерархию классов необходимо добавить интерфейс для реализации общей логики;

– в иерархии классов обязательно должен присутствовать минимум 1 абстрактный класс;

– в базовых классах обязательно должно быть предусмотрено 2 конструктора с параметрами и конструктор по умолчанию;

– в наследниках необходимо продемонстрировать вызов конструктора базового класса;

– продемонстрировать вызов конструктора текущего класса;

– доступ к членам класса должен осуществляться через методы get и set, а сами члены класса скрыты с помощью модификаторов доступа;

– по умолчанию переменные класса должны инициализироваться с помощью констант, определенных в базовом классе или в интерфейсе;

– в главном классе продемонстрировать инициализацию объектов их построенной иерархии;

– продемонстрировать использование оператора **instanceof**.

## КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Каково назначение директивы instanceof?
2. Чем отличаются абстрактные классы от обычных?

3. В чем назначение абстрактных методов?

4. Что такое interface, в чем его отличие от абстрактных классов?

5. В чем особенности реализации интерфейса в сравнении с наследованием?

6. В чем особенность класса Object?

7. Что такое конструктор по умолчанию?

8. Как можно вызвать один конструктор из другого?

## СОДЕРЖАНИЕ ОТЧЕТА

1. Ф.И.О., группа, название лабораторной работы.
2. Цель работы.
3. Описание проделанной работы.
4. Результаты выполнения лабораторной работы.
5. Выводы.

Отчет и исходные коды запаковать в архив с названием по следующему шаблону:

Lab1–«группа, аббревиатура на латинице»–«Фамилия на латинице».

Пример: Lab3–13VS–Ivanov.zip.

### Порядок защиты лабораторной работы

1. Предоставление отчета о выполнении лабораторной работы.
2. Демонстрация программы.
3. Выполнение дополнительного задания.
4. Ответы на вопросы по теме текущей лабораторной работы.

## Лабораторная работа № 4. Классы-обертки

**Цель работы.** Изучить методы класса Object. Научиться использовать классы-обертки (wrappers), изучить механизмы boxing и widening.

Видеоматериал по лекции находится в файле кпп_лр4.wmv.

### Теоретическая часть

**Класс java.lang.Object**

**Object** это базовый класс для всех остальных объектов в Java. Каждый класс наследуется от Object. Соответственно, все классы наследуют методы

класса Object. Ссылочная переменная типа Object может указывать на объект любого другого класса, на любой массив, так как массивы реализуются как классы. В классе Object определен набор методов, который наследуется всеми классами:

```
class A {}

class B extends A {}

public class SampleApp
{
    public static void main(String... args)
    {
        A object1 = new A();

        //все пользовательские классы и их наследники наследуются от Object
        System.out.println(object1 instanceof Object); //true

        System.out.println(new B() instanceof Object); //true

        //массивы наследуются от Object
        System.out.println(new int[10][10][10] instanceof Object); //true

        //сроки наследуются от Object
        System.out.println("label" instanceof Object); //true
    }
}
```

Методы класса **Object:**

- public final native Class getClass();
- public native int hashCode();
- public boolean equals(Object obj);
- protected native Object clone() throws CloneNotSupportedException;
- public String toString();
- public final native void notify();
- public final native void notifyAll();
- public final native void wait(long timeout) throws InterruptedException;
- public final void wait(long timeout, intnanos) throws InterruptedException;
- public final void wait() throws InterruptedException;
- protected void finalize() throws Throwable.

Рассмотрим подробно основные методы и их назначение.

### Методы hashCode() и equals()

Метод **hashCode()** используется для получения уникального целого номера для данного объекта. Когда необходимо сохранить объект как структуру данных в некой хэш-таблице (такой объект также называют корзиной – bucket), этот номер используется для определения его местонахождения в этой таблице. По умолчанию, метод hashCode() для объекта возвращает номер ячейки памяти, где объект сохраняется.

Метод **equals()**, как и следует из его названия, используется для простой проверки равенства двух объектов. Реализация этого метода по умолчанию просто проверяет по ссылкам два объекта на предмет их эквивалентности.

Все работает отлично до тех пор, пока мы не переопределяем ни один из этих методов в своих классах. Но иногда в приложениях необходимо изменять поведение по умолчанию некоторых объектов.

Давайте возьмем пример, где в нашем приложении имеется объект Employee. Напишем минимально возможную структуру такого класса:

```
1  public class Employee
2  {
3      private Integer id;
4      private String firstname;
5      private String lastName;
6      private String department;
7
8      public Integer getId() {
9          return id;
10     }
11     public void setId(Integer id) {
12         this.id = id;
13     }
14     public String getFirstname() {
15         return firstname;
16     }
17     public void setFirstname(String firstname) {
18         this.firstname = firstname;
19     }
20     public String getLastName() {
21         return lastName;
22     }
23     public void setLastName(String lastName) {
24         this.lastName = lastName;
25     }
26     public String getDepartment() {
27         return department;
28     }
29     public void setDepartment(String department) {
30         this.department = department;
31     }
32 }
```

Описанный выше класс Employee имеет некоторые основные атрибуты и методы доступа. Сейчас рассмотрим простую ситуацию, где необходимо сравнить два объекта класса Employee, например:

```
1  public class EqualsTest {
2      public static void main(String[] args) {
3          Employee e1 = new Employee();
4          Employee e2 = new Employee();
5
6          e1.setId(100);
7          e2.setId(100);
8          //Печатает false в консоли
9          System.out.println(e1.equals(e2));
10     }
11 }
```

Написанный выше метод вернет “false”. Но правильно ли это на самом деле, учитывая, что эти оба объекта одинаковые? Чтобы достигнуть корректного поведения, нам нужно переопределить метод equals(), как и сделано в следующем фрагменте кода:

```
1  public boolean equals(Object o) {
2          if(o == null)
3          {
4              return false;
5          }
6          if (o == this)
7          {
8              return true;
9          }
10         if (getClass() != o.getClass())
11         {
12             return false;
13         }
14         Employee e = (Employee) o;
15         return (this.getId() == e.getId());
16 }
```

Таким образом, метод equals() при сравнении двух объектов возвращает истину, если содержимое объектов эквивалентно, и ложь – в противном случае. При переопределении метода equals() должны выполняться соглашения, предусмотренные спецификацией языка Java, а именно:

– рефлексивность – объект равен самому себе;

– симметричность – если x.equals(y) возвращает значение true, то и y.equals(x) всегда возвращает значение true;

– транзитивность – если метод equals() возвращает значение true при сравнении объектов x и y, а также y и z, то и при сравнении x и z будет возвращено значение true;

– непротиворечивость – при многократном вызове метода для двух не подвергшихся изменению за это время объектов возвращаемое значение всегда должно быть одинаковым;

– ненулевая ссылка при сравнении с литералом null всегда возвращает значение false.

Метод hashCode() переопределен, как правило, в каждом классе и возвращает число, являющееся уникальным идентификатором объекта, зависящим в большинстве случаев только от значения объекта. Его следует переопределять всегда, когда переопределен метод equals(). Метод hashCode()

возвращает хэш-код объекта, вычисление которого управляется следующими соглашениями:

– во время работы приложения значение хэш-кода объекта не изменяется, если объект не был изменен;

– все одинаковые по содержанию объекты одного типа **должны** иметь одинаковые хэш-коды;

– различные по содержанию объекты одного типа **могут** иметь различные хэш-коды.

Среда NetBeans предоставляет возможность корректного переопределения методов **equals** и **hashCode.**

Рассмотрим класс Student в следующем фрагменте кода:

```
1  public class Student
2  {
3      private String name;
4      private String group;
5  
6      public String getName()
7      {
8          return name;
9      }
10 
11     public void setName(String name)
12     {
13         this.name = name;
14     }
15 
16     public String getGroup()
17     {
18         return group;
19     }
20 
21     public void setGroup(String group)
22     {
23         this.group = group;
24     }
25 
26     public Student(String name, String group)
27     {
28         this.name = name;
29         this.group = group;
30     }
31 }
```

С помощью контекстного меню или меню (Source) или комбинации Alt + Insert вызываем контекстное меню генерации исходного кода (рисунок 19).

В появившемся меню выбираем пункт **equals()** and **hashCode()** (рисунок 20).

Рисунок 19. – Генерация исходного кода

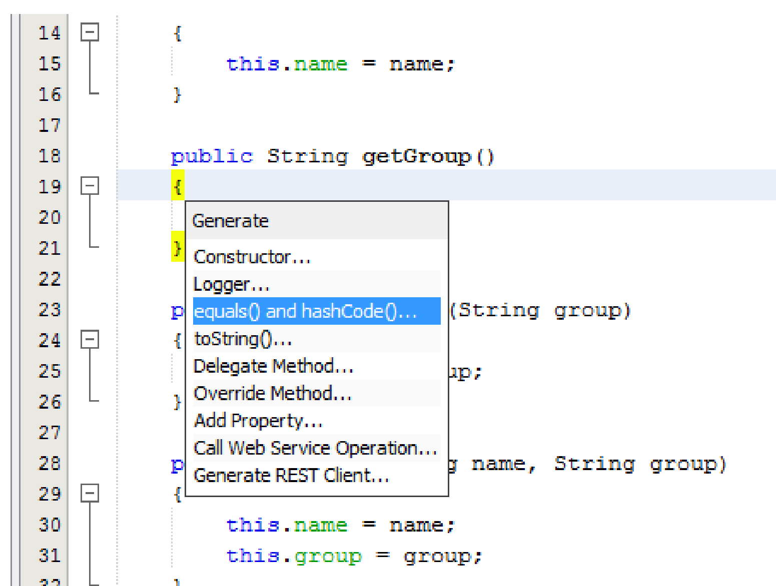


Рисунок 20. – Генерация методов equals() и hashcode()

В появившемся окне отмечаем, какие поля будут использоваться для сравнения, а какие для расчета хэш-кода (рисунок 21).

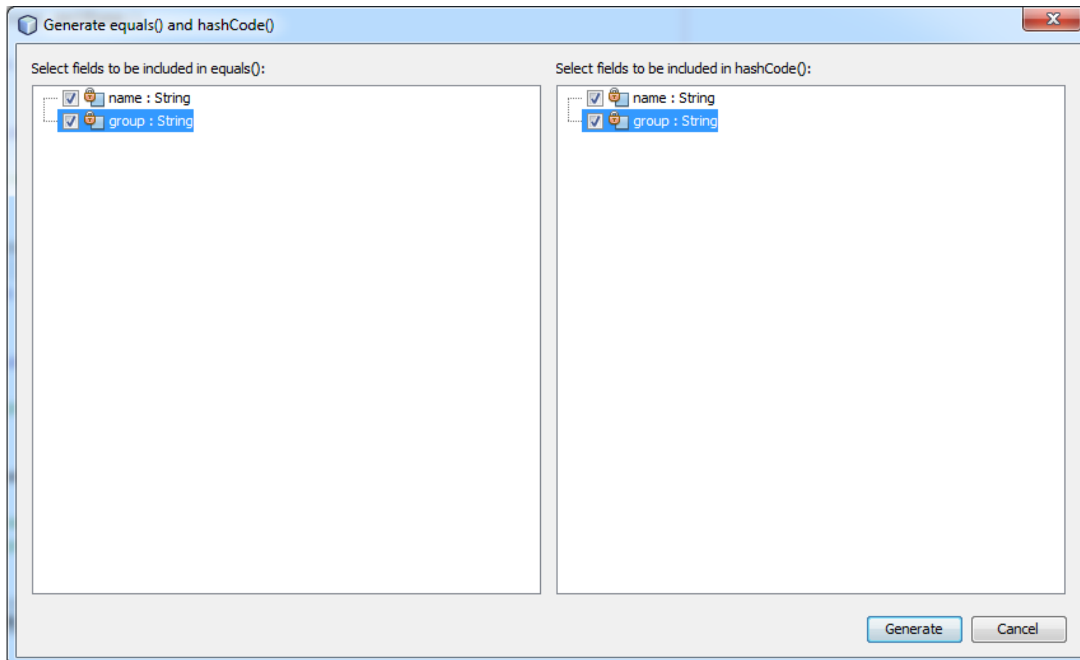


**Рисунок 21. – Выбор полей для генерации методов equals и hashcode()**

В нашем примере мы выбираем все поля класса Student. Для соблюдения требований Java относительно методов **equals()** и **hashCode()** рекомендуется в обоих частях окна отмечать одинаковый набор полей.

По кнопке **Generate**, класс Student будет дополнен следующим фрагментом кода:

```java
    @Override
    public int hashCode()
    {
        int hash = 7;
        hash = 13 * hash + Objects.hashCode(this.name);
        hash = 13 * hash + Objects.hashCode(this.group);
        return hash;
    }

    @Override
    public boolean equals(Object obj)
    {
        if (obj == null)
        {
            return false;
        }
        if (getClass() != obj.getClass())
        {
            return false;
        }
        final Student other = (Student) obj;
        if (!Objects.equals(this.name, other.name))
        {
            return false;
        }
        if (!Objects.equals(this.group, other.group))
        {
            return false;
        }
        return true;
    }
}
```

Проверим корректность работы данных методов на следующем примере:

```
1   public class SampleApp
2   {
3       public static void main(String... args)
4       {
5           Student student1 = new Student("Иванов Петр Михайлович", "23-ВС");
6           Student student2 = new Student("Иванов Петр Михайлович", "23-ВС");
7
8           //вернет false - т.к. сравниваются ссылки на объекты
9           System.out.println(student1 == student2);
10
11          //вернет true - т.к. сравниваются объекты по содержимому
12          System.out.println(student1.equals(student2));
13
14          //вернет true - т.к. объекты равны по содержимому
15          System.out.println(student1.hashCode() == student2.hashCode());
16      }
17  }
```

Если наша хэш-функция требует много времени для вычисления хэш-кода, мы можем хранить его значение (полученное на этапе инициализации или при первом обращении к объекту). Не забывайте при этом, что при изменении данных объекта необходимо пересчитать его хэш-код.

**Метод toString()**

Метод toString в Java используется для предоставления ясной и достаточной информации об объекте (Object) в удобном для человека виде. Правильное переопределение метода toString может помочь в ведении журнала работы и в отладке Java программы, предоставляя ценную и важную информацию. Поскольку toString() определен в java.lang.Object класса и его реализация по умолчанию не предоставляет много информации, всегда лучшей практикой является переопределение данного метода в классе-потомке.

По умолчанию реализация toString создает вывод в виде package.class @hashCode, к примеру, toString() метод класса Student напечатает by.psu.fit. cmsn.cross_pp.sample1.main.Student@c774c391, где c774c391 – это хэш-код объекта в шестнадцатеричном виде. Эта информация не особо полезна во время поиска какой-либо проблемы при отладке программы.

Переопределить метод toString можно вручную, а можно с помощью IDE. В NetBeansэто делается с помощью генератора исходногокода рассмотренного выше.

В меню генератора выбираем **toString()…** (рисунок 22).

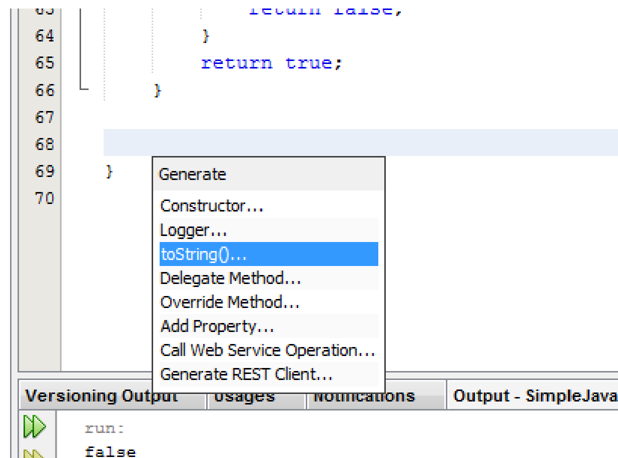


**Рисунок 22. – Генерация метода toString()**

В появившемся окне отмечаем поля, которые будут включены в возвращаемую методом toString строку (рисунок 23).

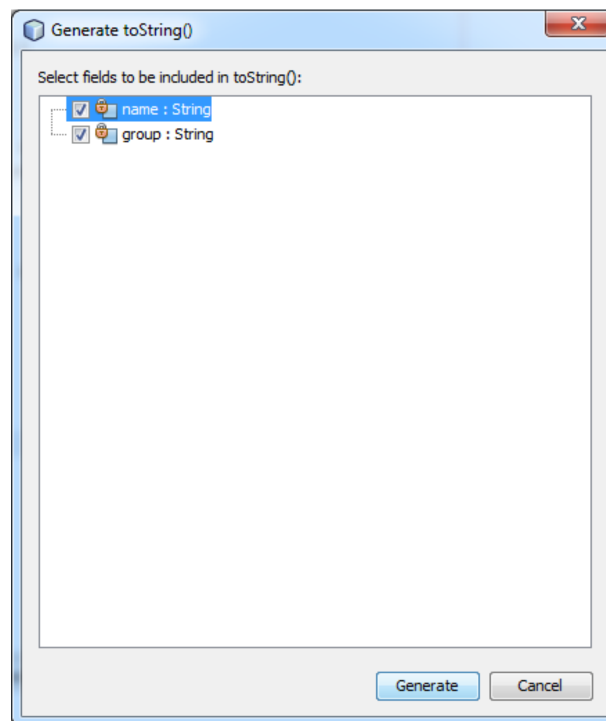


**Рисунок 23. – Выбор полей для включения в метод toString()**

В итоге в наш класс будет добавлен следующий код:

```
64      @Override
65      public String toString()
66      {
67          return "Student{" + "name=" + name + ", group=" + group + '}';
68      }
69  }
```

Проверим работу метода toString() на примере:

```
public class SampleApp
{
    public static void main(String... args)
    {
        Student student1 = new Student("Иванов Петр Михайлович", "23-ВС");

        //при преобразовании в строку у объекта всегда автоматически
        //вызывается метод toString()
        System.out.println(student1);

        System.out.println(student1.toString());

        System.out.println(""+student1);
    }
}
```

**Метод clone()**

При выполнении метода clone() сначала проверяется, можно ли клонировать исходный объект. Если разработчик хочет сделать объекты своего класса доступными для клонирования через Object.clone(), то он должен реализовать в своем классе интерфейс Cloneable. В этом интерфейсе нет ни одного элемента, он служит лишь признаком для виртуальной машины, что объекты могут быть клонированы. Если проверка не выполняется успешно, метод порождает ошибку **CloneNotSupportedException**.

Если интерфейс **Cloneable** реализован, то порождается новый объект от того же класса, от которого был создан исходный объект. При этом копирование выполняется на уровне виртуальной машины, никакие конструкторы не вызываются. Затем значения всех полей, объявленных, унаследованных либо объявленных в родительских классах, копируются. Полученный объект возвращается в качестве клона.

Обратите внимание, что сам класс Object не реализует интерфейс Cloneable, а потому попытка вызова newObject().clone() будет приводить к ошибке. Метод clone() предназначен скорее для использования в наследниках, которые могут обращаться к нему с помощью выражения super.clone(). Примитивные поля копируются и далее существуют независимо в исходном и клонированном объектах. Изменение одного не сказывается на другом.

А вот ссылочные поля копируются по ссылке, оба объекта ссылаются на одну и ту же область памяти (исходный объект). Поэтому изменения, происходящие с исходным объектом, сказываются на клонированном.

Если нам нужно полное/глубокое (deep) копирование класса, то в методе clone() этого класса, после получения клона суперкласса, необходимо скопировать нужные поля. Один из недостатков метода clone() – это тот факт, что возвращается тип Object, поэтому требуется нисходящее преобразование типа. Метод clone() несовместим с final полями. Если мы попробуем клонировать final поле, компилятор остановит нас. Единственное решение – отказаться от final. Рассмотрим следующий пример. В классе Group будет храниться описание группы для класса Student:

```
1   import java.util.Objects;
2
3   public class Group implements Cloneable
4   {
5       private String title;
6
7       public String getTitle()
8       {
11
12      public void setTitle(String title)
13      {
16
17      @Override
18      public int hashCode()
19      {
24
25      @Override
26      public boolean equals(Object obj)
27      {
43
44      @Override
45      public Object clone() throws CloneNotSupportedException
46      {
47          Group group = (Group)super.clone();
48          group.setTitle(getTitle());
49          return group;
50      }
51
52      public Group(String title)
53      {
54          this.title = title;
55      }
56  }
```

Пример класса Studentс полем типа Group и с реализацией глубокого клонирования в следующем фрагменте кода:

```
1   public class Student implements Cloneable
2   {
3       private String name;
4       private Group  group;
5
6       public String getName()
7       {
10
11      public void setName(String name)
12      {
15
16      public Group getGroup()
17      {
20
21      public void setGroup(Group group)
22      {
25
26      @Override
27      public int hashCode()
28      {
34
35      @Override
36      public boolean equals(Object obj)
37      {
57
```

```
58      @Override
59      public Object clone() throws CloneNotSupportedException
60      {
61          Student cloned = (Student)super.clone();
62          cloned.setName(getName());
63          cloned.setGroup((Group)group.clone());
64          return cloned;
65      }
66
67      @Override
68      public String toString()
69      {
70          return "Student{" + "name=" + name + ", group=" + group + '}';
71      }
72  }
```

Демонстрация глубокого клонирования в следующем фрагменте кода:

```
1   public class SampleApp
2   {
3       //т.к. метод clone может вызвать исключение CloneNotSupportedException
4       //для метода main добавляем конструкцию throws CloneNotSupportedException
5       public static void main(String... args) throws CloneNotSupportedException
6       {
7           Student student1 = new Student("Иванов Петр Михайлович", new Group("23-ВС"));
8           Student student2 = (Student)student1.clone();
9
10          //false т.к. student1 и student2 должны быть разными объектами
11          //но одинаковы по содержанию
12          System.out.println(student1 == student2);
13
14          //true т.к. объекты student1 и student2 одинаковы по содержанию
15          System.out.println(student1.equals(student2));
16
17          System.out.println(student1);
18          System.out.println(student2);
19      }
20  }
```

Результат выполнения показан на рисунке 24.

```
run:
false
true
Student{name=Иванов Петр Михайлович, group=Group{title=23-ВС}}
Student{name=Иванов Петр Михайлович, group=Group{title=23-ВС}}
BUILD SUCCESSFUL (total time: 0 seconds)
```

**Рисунок 24. – Лог выполнения программы**

### Метод finalize()

Иногда при уничтожении объект должен будет выполнять какое-либо действие. Например, если объект содержит какой-то ресурс, отличный от ресурса Java (вроде файлового дескриптора или шрифта), может требоваться гарантия освобождения этих ресурсов перед уничтожением объекта. Для подобных ситуаций Java предоставляет механизм, называемый **финализацией**. Используя финализацию, можно определить конкретные действия, которые будут выполняться непосредственно перед удалением объекта сборщиком мусора.

Чтобы добавить в класс средство выполнения финализации, достаточно определить метод finalize(). Среда времени выполнения Java вызывает этот метод непосредственно перед удалением объекта данного класса. Внутри метода finalize() нужно указать те действия, которые должны быть выполнены перед уничтожением объекта. Сборщик мусора запускается периодически, проверяя наличие объектов, на которые отсутствуют ссылки как со стороны какого-либо текущего состояния, так и косвенные ссылки через другие ссылочные объекты. Непосредственно перед освобождением ресурсов среда времени выполнения Java вызывает метод finalize() по отношению к объекту.

Объект может подлежать утилизации в разных случаях:

– если переменная ссылочного типа, которая ссылается на объект, установлена в положение «0», объект подлежит утилизации, в том случае, если на него нет других ссылок;

– если переменная ссылочного типа, которая ссылается на объект, создана для ссылки на другой объект, объект подлежит утилизации, в том случае, если на него нет других ссылок;

– объекты, созданные локально в методе, подлежат утилизации, когда метод завершает работу, если только они не экспортируются из этого метода (т.е. возвращаются или генерируются как исключение);

– объекты, которые ссылаются друг на друга, могут подлежать утилизации, если ни один из них не доступен живому потоку.

Рассмотрим пример:

```
public class TestGC
  {
    public static void main(String [] args)
    {
      Object o1 = new Integer(3);               // Line 1
      Object o2 = new String("Tutorial");      // Line 2
      o1 = o2;                                    // Line 3
      o2 = null;                                  // Line 4
      // Rest of the code here
    }
  }
```

В этом примере объект Integer (целочисленный), на который первоначально ссылается указатель o1, может подвергаться утилизации после строки 3, так как o1 теперь ссылается на объект String (строковый). Несмотря на то, что o2 создан для ссылки к нулю, объект String (строковый) не подлежит утилизации, так как o1 ссылается на него.

Важно понимать, что метод finalize() вызывается только непосредственно перед сборкой мусора. Например, он не вызывается при выходе объекта за рамки области определения. Это означает, что неизвестно, когда

будет – и даже будет ли вообще – выполняться метод finalize(). Поэтому программа должна предоставлять другие средства освобождения используемых объектом системных ресурсов и тому подобного. Нормальная работа программы не должна зависеть от метода finalize().

Для метода finalize справедливы следующие высказывания:

1. finalize() можно использовать только в двух случаях:

1.1. Проверка/подчистка ресурсов с логированием.

1.2. При работе с нативным кодом, который не критичен к утечке ресурсов.

2. finalize() замедляет работу GC по очистке объекта в 430 раз.

3. finalize() может быть не вызван.

**Классы-обертки в Java**

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.

Также классы-обертки содержат статические методы для обеспечения удобного манипулирования соответствующими примитивными типами, например, преобразование к строковому виду. На рисунке 25 приведены примитивные типы и соответствующие им классы-обертки:

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

**Рисунок 25. – Классы-обертки**

При этом классы-обертки числовых типов Byte, Short, Integer, Long, Float, Double наследуются от одного класса – Number. В нем объявлены методы, возвращающие числовое значение во всех числовых форматах Java (byte, short, int, long, float и double).

Все классы-обертки реализуют интерфейс **Comparable**. Все классы-обертки числовых типов имеют метод **equals(Object),** сравнивающий примитивные значения объектов.

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.

**pubic 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:

**pubic static String toOctalString(int i)** – восьмеричное;

**pubic static String toBinaryString(int i)** – двоичное;

**pubic 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).

**Character**

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 char;

**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.

**Boolean**

Представляет класс-обертку для примитивного типа boolean. Для получения примитивного значения используется метод **booleanValue().**

**Void**

Этот класс-обертка, в отличие от остальных, не имеет открытого конструктора. Более того, экземпляр этого класса вообще не может быть получен. Он нужен только для получения ссылки на объект Class, соответствующий void. Эта ссылка представлена статической константой TYPE.

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 xxxxValue() (например intValue());

– классы-обертки также являются классами-утилитами, т.е. предоставляют набор статических методов для работы с примитивными типами;

– классы-обертки являются неизменяемыми.

**Автоупаковка (Boxing) и Распаковывание (Unboxing)**

Автоупаковка это механизм неявной инициализации объектов классов-оберток (Byte, Short, Character, Integer, Long, Float, Double) значениями соответствующих им исходных примитивных типов (соответственно byte, short, char, int, long, float, double), без явного использования конструктора класса.

Автоупаковка происходит при прямом присвоении примитива классу-обертке (с помощью оператора "="), либо при передаче примитива в параметры метода (типа «класса-обертки»). Автоупаковке в «классы-обертки» могут быть подвергнуты как переменные примитивных типов, так и константы времени компиляции (литералы и final-примитивы). При этом литералы должны быть синтаксически корректными для инициализации переменной исходного примитивного типа.

Пример кода для автоупаковки и распаковки представлен ниже.

**Автоупаковка**

```
1 Integer integer = 9;
```

**Распаковка**

```
1 int in = 0;
2 in = new Integer(9);
```

Когда используется автоупаковка и распаковка?

Автоупаковка применяется компилятором Java в следующих условиях:

– когда значение примитивного типа передается в метод в качестве параметра метода, который ожидает объект соответствующего класса-оболочки;

– когда значение примитивного типа присваивается переменной соответствующего класса оболочки.

Распаковка применяется компилятором Java в следующих условиях:

– когда объект передается в качестве параметра методу, который ожидает соответствующий примитивный тип;

– когда объект присваивается переменной соответствующего примитивного типа.

**Автоупаковка и распаковка при перегрузке метода**

Автоупаковка и распаковка выполняется при перегрузке метода на основании следующих правил:

– *расширение «побеждает» упаковку. В ситуации, когда становится выбор между расширением и упаковкой, расширение предпочтительней.*

Пример кода, показывающий преимущество перегрузки:

```
1  public class WideBoxed {
2    public class WideBoxed {
3    static void methodWide(int i) {
4          System.out.println("int");
5     }
6
7    static void methodWide( Integer i ) {
8        System.out.println("Integer");
9    }
10
11    public static void main(String[] args) {
13         short shVal = 25;
14         methodWide(shVal);
15     }
16    }
17  }
```

Вывод программы – тип int.

– *расширение побеждает переменное количество аргументов. В ситуации, когда становится выбор между расширением и переменным количеством аргументов, расширение предпочтительней.*

Пример кода, показывающий преимущество перегрузки:

```
1  public class WideVarArgs {
2
3    static void methodWideVar(int i1, int i2) {
4      System.out.println("int int");
5    }
6
7    static void methodWideVar(Integer... i) {
8       System.out.println("Integers");
9    }
10
11   public static void main( String[] args) {
12       short shVal1 = 25;
13      short shVal2 = 35;
14     methodWideVar( shVal1, shVal2);
15   }
16 }
```

Вывод программы – int int.

– *упаковка побеждает переменное количество аргументов. В ситуации, когда становится выбор между упаковкой и переменным количеством аргументов, упаковка предпочтительней.*

Пример кода, показывающий преимущество перегрузки:

```
1  public class BoxVarargs {
2       static void methodBoxVar(Integer in) {
3           System.out.println("Integer");
4       }
5
6       static void methodBoxVar(Integer... i) {
7           System.out.println("Integers");
8       }
9       public static void main(String[] args) {
10          int intVal1 = 25;
11          methodBoxVar(intVal1);
12      }
13 }
```

Вывод программы – Integer.

Мы должны помнить о следующих вещах, используя автоупаковку:

– при сравнении объектов оператором "==" может возникнуть путаница, так как он может применяться к примитивным типам и объектам. Когда этот оператор применяется к объектам, он фактически сравнивает ссылки на объекты, а не сами объекты;

– смешивание объектов и примитивных типов с оператором равенства и отношения. Если мы сравниваем примитивный тип с объектом, то происходит распаковывание объекта, который может бросить NullPointerException, если объект null;

– кэширование объектов. Метод valueOf() создает контейнер примитивных объектов, которые он кэширует. Поскольку значения кэшируются в диапазоне от –128 до 127 включительно, эти кэшируемые объекты могут себя вести по-разному;

– ухудшение производительности. Автоупаковка или распаковка ухудшают производительность приложения, поскольку это создает нежелательный объект, из-за которого сборщику мусора приходится работать чаще.

## ЗАДАНИЕ К ЛАБОРАТОРНОЙ РАБОТЕ

1. Ознакомиться с теоретической частью.
2. Во всех классах из лабораторной работы № 3 изменить примитивные типы на классы-обертки;
3. Продемонстрировать в данных классах механизм автоупаковки и автораспаковки на примере перегруженных методов getters и setters.
4. Продемонстрировать в методах данных классов инициализацию переменных с числовыми типами из строк.
5. Реализовать для двух любых классов из лабораторной работы № 3 механизм глубокого клонирования и переопределить в этих же классах методы hashCode, equals, toString. Продемонстрировать клонирование и работу переопределенных методов.

## КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Каковы особенности класса Object?
2. Объясните назначение методов hashCode() и equals().
3. Для чего используется метод toString()?
4. Объясните как использовать клонирование объектов с помощью метода clone(), условия использования и ограничения.
5. Каково назначение метода finalize()?
6. Что такое автоупаковка и автораспаковка?
7. В каких случаях появляется необходимость использовать классы-обертки, привести примеры.

## СОДЕРЖАНИЕ ОТЧЕТА

1. Ф.И.О., группа, название лабораторной работы.
2. Цель работы.
3. Описание проделанной работы.
4. Результаты выполнения лабораторной работы.
5. Выводы.

Отчет и исходные коды запаковать в архив с названием по следующему шаблону:

Lab1–«группа, аббревиатура на латинице»–«Фамилия на латинице».

Пример: Lab4–13VS–Ivanov.zip.

### Порядок защиты лабораторной работы

1. Предоставление отчета о выполнении лабораторной работы.
2. Демонстрация программы.
3. Выполнение дополнительного задания.
4. Ответы на вопросы по теме текущей лабораторной работы.

## Лабораторная работа № 5.
## Обработка исключений, работа со строками

**Цель работы.** Изучить методы работы с исключениями в Java. Научиться работать со строками в Java.

### Теоретическая часть

**Исключения**

**Исключение** – это нештатная ситуация, ошибка во время выполнения программы. Самый простой пример – деление на ноль. Можно вручную отслеживать возникновение подобных ошибок, а можно воспользоваться специальным механизмом исключений, который упрощает создание больших надежных программ, уменьшает объем необходимого кода и повышает уверенность в том, что в приложении не будет необработанной ошибки.

В методе, в котором происходит ошибка, создается и передается специальный объект. Метод может либо обработать исключение самостоя-

тельно, либо пропустить его. В любом случае исключение ловится и обрабатывается. Исключение может появиться благодаря самой системе, либо вы сами можете создать его вручную. Системные исключения возникают при неправильном использовании языка Java или запрещенных приемов доступа к системе. Ваши собственные исключения обрабатывают специфические ошибки вашей программы.

Вернемся к примеру с делением. Деление на ноль можно предотвратить проверкой соответствующего условия. Но что делать, если знаменатель оказался нулем? Возможно, в контексте вашей задачи известно, как следует поступить в такой ситуации. Но, если нулевой знаменатель возник неожиданно, деление в принципе невозможно, и тогда необходимо возбудить исключение, а не продолжать исполнение программы.

Существует пять ключевых слов, используемых в исключениях**: try, catch, throw, throws, finally**.

Порядок обработки исключений следующий.

Операторы программы, которые вы хотите отслеживать, помещаются в блок **try**. Если исключение произошло, то оно создается и передается дальше. Ваш код может перехватить исключение при помощи блока **catch** и обработать его. Системные исключения автоматически передаются самой системой. Чтобы передать исключение вручную, используется **throw**. Любое исключение, созданное и передаваемое внутри метода, должно быть указано в его интерфейсе ключевым словом **throws**. Любой код, который следует выполнить обязательно после завершения блока **try**, помещается в блок **finally.**

Схематически код выглядит как в следующем фрагменте:

```
1   try
2   {
3       // блок кода, где отслеживаются ошибки
4   }
5   catch (тип_исключения_1 exceptionObject)
6   {
7       // обрабатываем ошибку
8   }
9   catch (тип_исключения_2 exceptionObject)
10  {
11      // обрабатываем ошибку
12  }
13  finally
14  {
15      // код, который нужно выполнить после завершения блока try
16  }
```

Существует специальный класс для исключений **Throwable**. В него входят два класса **Exception** и **Error**. Класс **Exception** используется для обработки исключений вашей программой. Вы можете наследоваться от него для создания собственных типов исключений. Для распространенных ошибок уже существует класс **RuntimeException**, который может обрабатывать деление на ноль или определять ошибочную индексацию массива. Класс **Error** служит для обработки ошибок в самом языке Java, и на практике вам не придется иметь с ним дело.

Прежде чем научиться обрабатывать исключения, нам хочется посмотреть, что происходит, если ошибку не обработать. Давайте разделим число котов в нашей квартире на ноль. Фрагмент кода:

```
1  package by.psu.fit.cmsn.cross_pp.sample1.main;
2
3  public class SampleApp
4  {
5      public static void main(String... args)
6      {
7          int catNumber;
8          int zero;
9          catNumber = 1; // у меня один кот
10         zero = 0; // ноль, он и в Африке ноль
11         int result = catNumber / zero;
12     }
13 }
```

Когда система времени выполнения Java обнаруживает попытку деления на ноль, она создает объект исключения и передает его. Однако никто не перехватывает его, хотя это должны были сделать мы. В данном случае, объект перехватывает стандартный системный обработчик Java. Он останавливает нашу программу и выводит сообщение об ошибке, которое можно увидеть в терминале (рисунок 26).

```
run:
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at by.psu.fit.cmsn.cross_pp.sample1.main.SampleApp.main(SampleApp.java:11)
Java Result: 1
BUILD SUCCESSFUL (total time: 0 seconds)
```

**Рисунок 26. – Результат выполнения**

Как видно, созданный объект исключения принадлежит к классу ArithmeticException, далее системный обработчик вывел краткое описание ошибки и место возникновения.

Вряд ли пользователи нашей программы будут довольны, если мы так и оставим обработку ошибки системе. Если программа будет завершаться с такой ошибкой, то скорее всего нашу программу просто удалят. Посмотрим, как мы можем исправить ситуацию.

Поместим проблемный код в блок **try**, а в блоке **catch** обработаем исключение в следующем фрагменте кода:

```
1  public class SampleApp
2  {
3      public static void main(String... args)
4      {
5          int catNumber;
6          int zero;
7
8          try
9          {
10             // мониторим код
11             catNumber = 1; // у меня один кот
12             zero = 0; // ноль, он и в Африке ноль
13             int result = catNumber / zero;
14             System.out.println("Не увидите это сообщение!");
15         }
16         catch (ArithmeticException e)
17         {
18             System.out.println("Нельзя котов делить на ноль!");
19         }
20
21         System.out.println("Жизнь продолжается");
22     }
23 }
```

Теперь программа аварийно не закрывается, так как мы обрабатываем ситуацию с делением на ноль.

В данном случае мы уже знали, к какому классу принадлежит получаемая ошибка, поэтому в блоке **catch** сразу указали конкретный тип. Обратите внимание, что последний оператор в блоке **try** не срабатывает, так как ошибка происходит раньше строчкой выше. Далее выполнение передается в блок **catch**, после чего выполняются следующие операторы в обычном порядке.

Операторы **try** и **catch** работают совместно в паре. Хотя возможны ситуации, когда **catch** может обрабатывать несколько вложенных операторов **try**.

Если мы хотим увидеть описание ошибки, то параметр **e** и поможет это сделать, например:

```
16         catch (ArithmeticException e)
17         {
18             System.out.println(e + ": Нельзя котов делить на ноль!");
19         }
```

Результат выполнения представлен на рисунке 27.

```
run:
java.lang.ArithmeticException: / by zero: Нельзя котов делить на ноль!
Жизнь продолжается
BUILD SUCCESSFUL (total time: 0 seconds)
```

**Рисунок 27. – Результат выполнения с перехватом исключения**

По умолчанию класс **Throwable**, к которому относится **ArithmeticException**, возвращает строку, содержащую описание исключения. Но вы можете и явно указать метод **e.toString**.

### Несколько исключений

Фрагмент кода может содержать несколько проблемных мест. Например, кроме деления на ноль, возможна ошибка индексации массива. В таком случае вам нужно создать два или более операторов **catch** для каждого типа исключения. Причем они проверяются по порядку. Если исключение будет обнаружено у первого блока обработки, то он будет выполнен, а остальные проверки пропускаются, и выполнение программы продолжается с места, которое следует за блоком try/catch. Фрагмент кода:

```
1  public class SampleApp
2  {
3      public static void main(String... args)
4      {
5          int catNumber;
6          int zero;
7  
8          try
9          {
10             catNumber = 1; // у меня один кот
11 
12             zero = 1; // ноль, он и в Африке ноль
13 
14             int result = catNumber / zero;
15 
16             // Создадим массив из трёх котов
17             String[] catNames = {"Васька", "Барсик", "Мурзик"};
18 
19             catNames[3] = "Рыжик";
20 
21             System.out.println("Не увидите это сообщение!");
22         }
23         catch (ArithmeticException e)
24         {
25             System.out.println(e.toString() + ": Нельзя котов делить на ноль!");
26         }
27         catch (ArrayIndexOutOfBoundsException e)
28         {
29             System.out.println("Ошибка: " + e.toString());
30         }
31 
32         System.out.println("Жизнь продолжается");
33     }
34 }
```

В примере мы добавили массив с тремя элементами, но обращаемся к четвертому элементу, так как забыли, что отсчет у массива начинается с нуля. Если оставить значение переменной **zero** равным нулю, то сработает обработка первого исключения деления на ноль, и мы даже не узнаем о существовании второй ошибки. Но допустим, что в результате каких-то вычислений значение переменной стало равно единице. Тогда наше исключение **ArithmeticException** не сработает. Однако сработает новое добавленное исключение **ArrayIndexOutOfBoundsException**. А дальше все пойдет как раньше.

Тут всегда нужно помнить одну особенность. При использовании множественных операторов catch обработчики подклассов исключений должны находиться выше, чем обработчики их суперклассов. Иначе суперкласс будет перехватывать все исключения, имея большую область перехвата. Иными словами, Exception не должен находиться выше ArithmeticException и ArrayIndexOutOfBoundsException. К счастью, среда разработки сама замечает непорядок и предупреждает вас, что такой порядок не годится. Увидев такую ошибку, попробуйте перенести блок обработки исключений ниже.

**Вложенные операторы try**

Операторы **try** могут быть вложенными. Если вложенный оператор **try** не имеет своего обработчика **catch** для определения исключения, то идет поиск обработчика **catch** у внешнего блока **try** и т.д. Если подходящий **catch** не будет найден, то исключение обработает сама система.

**Оператор throw**

Часть исключений может обрабатывать сама система. Но можно создать собственные исключения при помощи оператора **throw**. Код выглядит так:

```
throw экземпляр_Throwable
```

Нам нужно создать экземпляр класса Throwable или его наследников. Получить объект класса Throwable можно в операторе catch или стандартным способом через оператор new.

Поток выполнения останавливается непосредственно после оператора throw, и другие операторы не выполняются. При этом ищется ближайший блок try/catch соответствующего исключению типа, например:

```
public class SampleApp
{
    public static void main(String... args)
    {
        try
        {
            throw new NullPointerException("Кота не существует");
        }
        catch (NullPointerException e)
        {
            System.out.println(e.getMessage());
        }
    }
}
```

Результат показан на рисунке 28.

```
run:
Кота не существует
BUILD SUCCESSFUL (total time: 0 seconds)
```

**Рисунок 28. – Результат перехвата NullPointerException**

Мы создали новый объект класса **NullPointerException**. Многие классы исключений кроме стандартного конструктора по умолчанию с пустыми скобками имеют второй конструктор со строковым параметром, в котором можно разместить подходящую информацию об исключении. Получить текст из него можно через метод **getMessage()**, что мы и сделали в блоке **catch**.

**Оператор throws**

Если метод может породить исключение, которое он сам не обрабатывает, он должен задать это поведение так, чтобы вызывающий его код мог позаботиться об этом исключении. Для этого к объявлению метода добавляется конструкция **throws**, которая перечисляет типы исключений (кроме исключений **Error** и **RuntimeException** и их подклассов).

Общая форма объявления метода с оператором **throws**, например:

```
тип имя_метода(список_параметров) throws список_исключений
{
    // код внутри метода
}
```

В фрагменте **список_исключений** можно указать список исключений через запятую.

Создадим метод, который может породить исключение, но не обрабатывает его, например:

```
1  package by.psu.fit.cmsn.cross_pp.sample1.main;
2
3  public class SampleApp
4  {
5      // Метод без обработки исключения
6      public static void createCat()
7      {
8          System.out.println("Вы создали котёнка");
9          throw new NullPointerException("Кота не существует");
10     }
11
12     public static void main(String... args)
13     {
14         createCat();
15     }
16 }
```

Если мы запустим пример, то получим ошибку (рисунок 29).

```
run:
Вы создали котёнка
Exception in thread "main" java.lang.NullPointerException: Кота не существует
        at by.psu.fit.cmsn.cross_pp.sample1.main.SampleApp.createCat(SampleApp.java:9)
        at by.psu.fit.cmsn.cross_pp.sample1.main.SampleApp.main(SampleApp.java:14)
Java Result: 1
BUILD SUCCESSFUL (total time: 0 seconds)
```

**Рисунок 29. – Лог ошибки NullPointerException**

Исправим код в следующем фрагменте:

```
1  public class SampleApp
2  {
3      // Метод без обработки исключения
4      public static void createCat() throws NullPointerException
5      {
6          System.out.println("Вы создали котёнка");
7          throw new NullPointerException("Кота не существует");
8      }
9
10     public static void main(String... args)
11     {
12         try
13         {
14             createCat();
15         }
16         catch (NullPointerException e)
17         {
18             System.out.println(e.getMessage());
19         }
20     }
21 }
```

Мы поместили вызов метода в блок **try** и вызвали блок **catch** с нужным типом исключения. Теперь ошибки не будет.

**Оператор finally**

Когда исключение передано, выполнение метода направляется по нелинейному пути. Это может стать источником проблем. Например, при входе метод открывает файл и закрывает при выходе. Чтобы закрытие файла не было пропущено из-за обработки исключения, был предложен механизм **finally**.

Ключевое слово **finally** создает блок кода, который будет выполнен после завершения блока **try/catch**, но перед кодом, следующим за ним. Блок будет выполнен независимо от того, передано исключение или нет. Оператор **finally** не обязателен, однако каждый оператор **try** требует наличия либо **catch**, либо **finally**.

**Встроенные исключения Java**

Существует несколько готовых системных исключений. Большинство из них являются подклассами типа RuntimeException, и их не нужно включать в список **throws**. Вот небольшой список **непроверяемых** исключений:

- ClassNotFoundException – класс не найден;
- CloneNotSupportedException – попытка клонировать объект, который не реализует интерфейс Cloneable;
- IllegalAccessException – запрещен доступ к классу;
- InstantiationException – попытка создать объект абстрактного класса или интерфейса;
- InterruptedException – поток прерван другим потоком;
- NoSuchFieldException – запрашиваемое поле не существует;
- NoSuchMethodException – запрашиваемый метод не существует;
- ReflectiveOperationException – исключение, связанное с рефлексией.

Список **проверяемых** системных исключений, которые можно включать в список **throws**:

- ArithmeticException – арифметическая ошибка, например, деление на ноль;
- ArrayIndexOutOfBoundsException – выход индекса за границу массива;
- ArrayStoreException – присваивание элементу массива объекта несовместимого типа;
- ClassCastException – неверное приведение;
- EnumConstantNotPresentException – попытка использования неопределенного значения перечисления;

– IllegalArgumentException – неверный аргумент при вызове метода;
– IllegalMonitorStateException – неверная операция мониторинга;
– IllegalStateException – некорректное состояние приложения;
– IllegalThreadStateException – запрашиваемая операция несовместима с текущим потоком;
– IndexOutofBoundsException – тип индекса вышел за допустимые пределы;
– NegativeArraySizeException – создан массив отрицательного размера;
– NullPointerException – неверное использование пустой ссылки;
– NumberFormatException – неверное преобразование строки в числовой формат;
– SecurityException – попытка нарушения безопасности;
– StringIndexOutOfBounds – попытка использования индекса за пределами строки;
– TypeNotPresentException – тип не найден;
– UnsupportedOperationException – обнаружена неподдерживаемая операция.

**Создание собственных подклассов исключений**

Система не может предусмотреть все исключения, иногда вам придется создать собственный тип исключения для вашего приложения. Вам нужно наследоваться от **Exception** (как отмечалось ранее, этот класс наследуется от **Throwable**) и переопределить нужные методы класса **Throwable**:

– final void addSuppressed(Throwable exception) – добавляет исключение в список подавляемых исключений (JDK 7);
– Throwable fillInStackTrace() – возвращает объект класса Throwable, содержащий полную трассировку стека;
– Throwable getCause() – возвращает исключение, лежащее под текущим исключением или null;
– String getLocalizedMessage() – возвращает локализованное описание исключения;
– String getMessage() – возвращает описание исключения;
– StackTraceElement[] getStackTrace() – возвращает массив, содержащий трассировку стека и состояний из элементов класса StackTraceElement;
– final Throwable[] getSuppressed() – получает подавленные исключения (JDK 7);
– Throwable initCause(Throwable exception) – ассоциирует исключение с вызывающим исключением. Возвращает ссылку на исключение;

– void printStackTrace() – отображает трассировку стека;

– void printStackTrace(PrintStream stream) – посылает трассировку стека в заданный поток;

– void printStackTrace(PrintWriter stream) – посылает трассировку стека в заданный поток;

– void setStackTrace(StackTraceElement elments[]) – устанавливает трассировку стека для элементов (для специализированных приложений);

– String toString() – возвращает объект класса String, содержащий описание исключения.

**Перехват произвольных исключений**

Можно создать универсальный обработчик, перехватывающий любые типы исключения. Осуществляется это перехватом базового класса всех исключений **Exception**, например:

```
catch(Exception e)
{
    System.out.println("Перехвачено исключение");
}
```

Подобная конструкция не упустит ни одного исключения, поэтому ее следует размещать в самом конце списка обработчиков, во избежание блокировки следующих за ней обработчиков исключений.

**Основные правила обработки исключений**

Используйте исключения для того, чтобы:

– обработать ошибку на текущем уровне (избегайте перехватывания исключений, если не знаете, как с ними поступить);

– исправить проблему и снова вызвать метод, возбудивший исключение;

– предпринять все необходимые действия и продолжить выполнение без повторного вызова действия;

– попытаться найти альтернативный результат вместо того, который должен был бы произвести вызванный метод;

– сделать все возможное в текущем контексте и заново возбудить это же исключение, перенаправив его на более высокий уровень;

– сделать все, что можно в текущем контексте, и возбудить новое исключение, перенаправив его на более высокий уровень;

– завершить работу программы;

– упростить программу (если используемая схема обработки исключений делает все только сложнее, значит, она никуда не годится);

– добавить к вашей библиотеке и программе безопасности.

**Работа со строками в Java**

String – это класс в Java, который прописан в пакете java.lang. Это не примитивный тип данных, как int и long. Класс String представляет строковый набор символов. Строки используются практически по всех Java-приложениях, и есть несколько фактов, которые мы должны знать о классе String.

Это неизменяемый (immutable) и финализированный тип данных в Java, и все объекты класса String виртуальная машина хранит в пуле строк.

Еще одной особенностью String является способ получения объектов класса String, используя двойные кавычки и перегружая оператор "+" для конкатенации.

**Конструкторы**

Как и в случае любого другого класса, мы можем создавать объекты типа String с помощью оператора **new**. Для создания пустой строки используется конструктор без параметров:

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

Приведенный ниже фрагмент кода создает объект s типа **String**, инициализируя его строкой из трех символов, переданных конструктору в качестве параметра в символьном массиве, например:

```
char chars[] = { 'a', 'b', 'c' };
String s = new String(chars);
System.out.println(s);
```

Этот фрагмент кода выводит строку «abc».

Итак, у этого конструктора – 3 параметра:

```
String(char chars[], int начальныйИндекс, int числоСимволов);
```

Используем такой способ инициализации в нашем очередном примере:

```
char chars[] = { 'a', 'b', 'c', 'd', 'e', 'f' };
String s = new String(chars,2,3);
System.out.println(s);
```

Этот фрагмент выведет «cde».

### Специальный синтаксис для работы со строками

В Java включено несколько приятных синтаксических дополнений, цель которых – помочь программистам в выполнении операций со строками. В числе таких операций создание объектов типа String, слияние нескольких строк и преобразование других типов данных в символьное представление.

### Создание строк

Java включает в себя стандартное сокращение для этой операции – запись в виде литерала, в которой содержимое строки заключается в пару двойных кавычек. Приводимый ниже фрагмент кода эквивалентен одному из предыдущих, в котором строка инициализировалась массивом типа **char**.

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

Один из общих методов, используемых с объектами **String**, – метод **length**, возвращающий число символов в строке. Очередной фрагмент выводит число 3, поскольку в используемой в нем строке – 3 символа.

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

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. Очередной пример также выводит число 3.

```
System.out.println("abc".length());
```

### Слияние строк

Строку

```
String s = "He is " + age + " years old.";
```

в которой с помощью оператора + три строки объединяются в одну, прочесть и понять безусловно легче, чем ее эквивалент, записанный с явными вызовами тех самых методов, которые неявно были использованы в первом примере:

```
String s = new StringBuilder("He is ").append(age).append(" years old.").toString();
```

По определению каждый объект класса String не может изменяться. Нельзя ни вставить новые символы в уже существующую строку, ни поменять в ней одни символы на другие. И добавить одну строку в конец дру-

гой тоже нельзя. Поэтому транслятор Java преобразует операции, выглядящие как модификация объектов String, в операции с родственным классом StringBuilder.

*Замечание*. Все это может показаться нам необоснованно сложным. А почему нельзя обойтись одним классом String, позволив ему вести себя примерно так же, как StringBuilder? Все дело в производительности. Тот факт, что объекты типа String в Java неизменны, позволяет транслятору применять к операциям с ними различные способы оптимизации.

**Последовательность выполнения операторов**

Давайте еще раз обратимся к нашему последнему примеру:

```
String s = "He is " + age + " years old.";
```

В том случае, когда age – не **String**, а переменная, скажем, типа **int**, в этой строке кода заключено еще больше магии транслятора. Целое значение переменной **int** передается совмещенному методу append класса StringBuilder, который преобразует его в текстовый вид и добавляет в конец содержащейся в объекте строки. Нам нужно быть внимательными при совместном использовании целых выражений и слияния строк, в противном случае результат может получиться совсем не тот, который мы ждали.

Взгляните на следующую строку:

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

Может быть, можно надеяться, что в s будет записана строка «four: 4»? Не угадали – с нами сыграла злую шутку последовательность выполнения операторов. Так что в результате получается «four: 22».

Для того, чтобы первым выполнилось сложение целых чисел, нужно использовать скобки:

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

**Извлечение символов**

Для того, чтобы извлечь одиночный символ из строки, вы можете сослаться непосредственно на индекс символа в строке с помощью метода charAt. Если мы хотим в один прием извлечь несколько символов, можно воспользоваться методом getChars. В приведенном ниже фрагменте кода показано, как следует извлекать массив символов из объекта типа String.

```
public class SampleApp
{
    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);
    }
}
```

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 – **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 мы хотим узнать, одинаковы ли две строки, нам следует воспользоваться методом **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:

```
public class SampleApp
{
    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 приведен на рисунке 30.

```
run:
Hello equals Hello -> true
Hello equals Good-bye -> false
Hello equals HELLO -> false
Hello equalsIgnoreCase HELLO -> true
BUILD SUCCESSFUL (total time: 0 seconds)
```

**Рисунок 30. – Результат выполнения методов equals()**

В классе **String** реализована группа сервисных методов, являющихся специализированными версиями метода **equals**. Метод **regionMatches** используется для сравнения подстроки в исходной строке с подстрокой в строке-параметре. Метод **startsWith** проверяет, начинается ли данная подстрока фрагментом, переданным методу в качестве параметра. Метод **endsWith** проверяет, совпадает ли с параметром конец строки.

### Равенство

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, например:

```
1  public class SampleApp
2  {
3      public static void main(String... args)
4      {
5          String s1 = "Hello";
6          String s2 = new String(s1);
7          System.out.println(s1 + " equals " + s2 + " -> " + s1.equals(s2));
8          System.out.println(s1 + " == " + s2 + ", -> " + (s1 == s2));
9      }
10 }
```

Вот результат запуска этого примера (рисунок 31).

```
run:
Hello equals Hello -> true
Hello == Hello, -> false
BUILD SUCCESSFUL (total time: 0 seconds)
```

**Рисунок 31. – Другой результат выполнения методов equals()**

### Упорядочение

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 вос-

пользоваться методом **compareTo** 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**. Этот фрагмент кода выдает отсортированный в алфавитном порядке список строк.

```
1  public class SampleApp
2  {
3      static final String arr[] = {"Now", "is", "the", "time", "for", "all",
4                                   "good", "men", "to", "come", "to", "the",
5                                   "aid", "of", "their", "country"};
6
7
8      public static void main(String... args)
9      {
10         for (int j = 0; j < arr.length; j++)
11         {
12             for (int i = j + 1; i < arr.length; i++)
13             {
14                 if (arr[i].compareTo(arr[j]) < 0)
15                 {
16                     String t = arr[j];
17                     arr[j] = arr[i];
18                     arr[i] = t;
19                 }
20             }
21
22             System.out.println(arr[j]);
23         }
24     }
25 }
```

### Методы indexOf и lastIndexOf

В класс **String** включена поддержка поиска определенного символа или подстроки, для этого в нем имеются два метода – **indexOf** и **lastIndexOf**. 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.

```
1  public class SampleApp
2  {
3      public static void main(String... args)
4      {
5          String s = "Now is the time for all good men " +
6                     "to come to the aid of their country " +
7                     "and pay their due taxes.";
8
9          System.out.println(s);
10         System.out.println("indexOf(t) = " + s.indexOf('f'));
11         System.out.println("lastIndexOf(t) = " + s.lastIndexOf('f'));
12         System.out.println("indexOf(the) = " + s.indexOf("the"));
13         System.out.println("lastIndexOf(the) = " + s.lastIndexOf("the"));
14         System.out.println("indexOf(t, 10) = " + s.indexOf('f' , 10));
15         System.out.println("lastIndexOf(t, 50) = " + s.lastIndexOf('f' , 50));
16         System.out.println("indexOf(the, 10) = " + s.indexOf("the", 10));
17         System.out.println("lastIndexOf(the, 50) = " + s.lastIndexOf("the", 50));
18     }
19 }
```

Ниже приведен результат работы этой программы (рисунок 32). Обратите внимание на то, что индексы в строках начинаются с нуля.

```
run:
Now is the time for all good men to come to the aid of their country and pay their due taxes.
indexOf(t) = 16
lastIndexOf(t) = 53
indexOf(the) = 7
lastIndexOf(the) = 77
indexOf(t, 10) = 16
lastIndexOf(t, 50) = 16
indexOf(the, 10) = 44
lastIndexOf(the, 50) = 44
BUILD SUCCESSFUL (total time: 0 seconds)
```

**Рисунок 32. – Результат выполнения indexOf()**

### Модификация строк при копировании

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

### Метод substring

Мы можем 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, например:

```
"Hello World".substring(6) -> "World"
"Hello World".substring(3,8) -> "lo Wo"
```

### Метод concat

Слияние, или конкатенация, строк выполняется с помощью метода **concat**. Этот метод создает новый объект **String**, копируя в него содержимое исходной строки и добавляя в ее конец строку, указанную в параметре метода:

```
"Hello".concat(" World") -> "Hello World"
```

**Метод 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"
```

**Методы toLowerCase и toUpperCase**

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

```
"Hello".toLowerCase() -> "hello"
"Hello".toUpperCase() -> "HELLO"
```

**Метод trim**

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

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

**Метод valueOf**

Если мы имеем дело с каким-либо типом данных и хотим вывести значение этого типа в удобочитаемом виде, сначала придется преобразовать это значение в текстовую строку. Для этого существует метод **valueOf**. Такой статический метод определен для любого существующего в Java типа данных (все эти методы совмещены, то есть используют одно и то же имя). Благодаря этому не составляет труда преобразовать в строку значение любого типа.

**Объект StringBuilder**

StringBuilder – близнец класса String, предоставляющий многое из того, что обычно требуется при работе со строками. Объекты класса String представляют собой строки фиксированной длины, которые нельзя изменять. Объекты типа StringBuilder представляют собой последовательности символов, которые могут расширяться и модифицироваться. Java активно использует оба класса, но многие программисты предпочитают работать только с объектами типа String, используя оператор +. При этом Java выполняет всю необходимую работу со StringBuilder за сценой.

**Конструкторы**

Объект StringBuilder можно создать без параметров. Мы также можем передать конструктору целое число, для того чтобы явно задать требуемый

размер буфера. И, наконец, мы можем передать конструктору строку, при этом она будет скопирована в объект. Текущую длину StringBuilder можно определить, вызвав метод **length**, а для определения всего места, зарезервированного под строку в объекте StringBuilder, нужно воспользоваться методом **capacity**. Ниже приведен пример, поясняющий это:

```
1  public class SampleApp
2  {
3      public static void main(String... args)
4      {
5          StringBuilder sb = new StringBuilder("Hello");
6          System.out.println("buffer = " + sb);
7          System.out.println("length = " + sb.length());
8          System.out. println("capacity = " + sb.capacity());
9      }
10 }
```

На рисунке 33 показан вывод этой программы, из которого видно, что в объекте StringBuilder для манипуляций со строкой зарезервировано дополнительное место.

```
run:
buffer = Hello
length = 5
capacity = 21
BUILD SUCCESSFUL (total time: 0 seconds)
```

**Рисунок 33. – Результат работы StringBuilder**

### Метод ensureCapacity

Если мы после создания объекта StringBuilder захотим зарезервировать в нем место для определенного количества символов, мы можем для установки размера буфера воспользоваться методом **ensureCapacity**. Это бывает полезно, когда мы заранее знаем, что нам придется добавлять к буферу много небольших строк.

### Метод setLength

Если нам вдруг понадобится в явном виде установить длину строки в буфере, воспользуемся методом **setLength**. Если мы зададим значение, большее чем длина содержащейся в объекте строки, этот метод заполнит конец новой, расширенной строки символами с кодом ноль. В приводимой чуть дальше программе метод **setLength** используется для укорачивания буфера.

### Методы charAt и setCharAt

Одиночный символ может быть извлечен из объекта StringBuilderс помощью метода **charAt**. Другой метод **setCharAt** позволяет записать в задан-

ную позицию строки нужный символ. Использование обоих этих методов проиллюстрировано в примере:

```
1   public class SampleApp
2   {
3       public static void main(String... args)
4       {
5           StringBuilder sb = new StringBuilder("Hello");
6           System.out.println("buffer before = " + sb);
7           System.out.println("charAt(1) before = " + sb.charAt(1));
8           sb.setCharAt(1, 'i');
9           sb.setLength(2);
10          System.out.println("buffer after = " + sb);
11          System.out.println("charAt(1) after = " + sb.charAt(1));
12      }
13  }
```

Вот вывод, полученный при запуске этой программы (рисунок 34).

```
run:
buffer before = Hello
charAt(1) before = e
buffer after = Hi
charAt(1) after = i
BUILD SUCCESSFUL (total time: 0 seconds)
```

**Рисунок 34. – Другой результат работы StringBuilder**

### Метод append

Метод **append** класса StringBuilder обычно вызывается неявно при использовании оператора + в выражениях со строками. Для каждого параметра вызывается метод **String.valueOf**, и его результат добавляется к текущему объекту StringBuilder. К тому же при каждом вызове метод append возвращает ссылку на объект StringBuilder, с которым он был вызван. Это позволяет выстраивать в цепочку последовательные вызовы метода, как это показано в очередном примере.

```
1   public class SampleApp
2   {
3       public static void main(String... args)
4       {
5           String s;
6           int a = 42;
7           StringBuilder sb = new StringBuilder(40);
8           s = sb.append("a = ").append(a).append("!").toString();
9           System.out.println(s);
10      }
11  }
```

Вот вывод этого примера (рисунок 35):

```
run:
a = 42!
BUILD SUCCESSFUL (total time: 0 seconds)
```

**Рисунок 35. – Результат работы append()**

### Метод insert

Метод **insert** идентичен методу **append** в том смысле, что для каждого возможного типа данных существует своя совмещенная версия этого метода. Правда, в отличие от **append**, он не добавляет символы, возвращаемые методом **String.valueOf**, в конец объекта StringBuilder, а вставляет их в определенное место в буфере, задаваемое первым его параметром. В очередном нашем примере строка "there" вставляется между "hello" и "world!":

```
1  public class SampleApp
2  {
3      public static void main(String... args)
4      {
5          StringBuilder sb = new StringBuilder("hello world !");
6          sb.insert(6,"there ");
7          System.out.println(sb);
8      }
9  }
```

При запуске эта программа выводит следующую строку (рисунок 36):

```
run:
hello there world !
BUILD SUCCESSFUL (total time: 0 seconds)
```

**Рисунок 36. – Результат работы insert()**

### Пул строк

В Java есть **пул строк**. Туда неявно попадают все литералы (строки в коде, оформленные через двойные кавычки). Также есть возможность положить строку в пул явно с помощью метода intern(). Например, классы javareflection интенсивно используют этот метод, и все имена полей и методов класса попадают в пул. Соответственно, то же самое происходит при использовании стандартной java сериализации, которая неявно использует reflection. Также обычно библиотеки, работающие с XML, кладут в пул названия элементов и атрибутов документов.

Пул располагается в PermGen и хранит строки с помощью слабых ссылок. Таким образом, при вызове Full GC, если ваша куча больше не ссылается на строки, находящиеся в пуле, то сборщик мусора их очищает. Однако не стоит увлекаться складыванием в пул всего подряд: чем больше в вашем пуле находится строк, тем дольше будет операция на проверку находится ли строка уже там или нет.

Говоря о пуле строк, стоит упомянуть, почему некоторые библиотеки при объявлении констант используют вышеописанный метод intern:

```
public static final SOME_CONSTANT = "SOME_VALUE".intern();
```

Данная конструкция может показаться абсурдной неискушенному программисту. И правда, зачем класть эту строку в пул, если она и так константа и присутствует в единственном экземпляре. И вообще, это же литерал, он и так должен неявно попадать в пул. Однако цель, с которой это делается в библиотеках, – совсем не пулинг. Дело в том, что если бы эта константа была объявлена в библиотеке просто как

```
public static final SOME_CONSTANT = "SOME_VALUE";
```

и мы использовали ее в своем коде, то при компилировании в байткод эта строка бы «закешировалась». И если бы мы потом подложили jar с новой версией библиотеки и не перекомпилировали наши исходники, то наш код изменения в данной константе просто бы не заметил.

## ЗАДАНИЕ К ЛАБОРАТОРНОЙ РАБОТЕ

1. Создать консольный интерфейс для динамического создания объектов из лабораторной работы № 3.
2. В интерфейсе должны быть предусмотрены команды для создания различных типов объектов и ввода всех их параметров, а также команда просмотра списка уже созданных объектов.
3. В конструкторах и сеттерах объектов предусмотреть появление исключения в случае попытки записать в объект некорректное значение, создать обработчик данных исключений с корректными сообщениями пользователю.
4. Предусмотреть наличие нескольких (не менее 3) пользовательских исключений с их использованием и обработкой.
5. Дополнительно выполнить 2 задания из таблицы 3, первое согласно индивидуальному варианту, второе – следующее за ним.

Таблица 3. – Дополнительные задания к лабораторной работе № 5

| Номер варианта | Задание |
|---|---|
| 1 | 2 |
| 1 | Дан текст. Найти сколько он содержит букв а, б, с |
| 2 | Дан текст. Определить, содержит ли он символы, отличные от букв и пробела |
| 3 | Дан текст. Если в тексте нет символа *, то оставить этот текст без изменения, иначе каждую из малых латинских букв, предшествующих первому вхождению символа *, заменить на цифру |
| 4 | Дан текст. Если в тексте нет символа +, то оставить текст без изменения, иначе каждую из цифр, предшествующую первому вхождению символа +, заменить символом — |

Окончание таблицы 3

| 1 | 2 |
|---|---|
| 5 | Дан текст. Если в нем нет малых латинских букв, то оставить его без изменения, иначе каждый из символов, следующих за первой группой малых латинских букв, заменить точкой |
| 6 | Дан текст. Выяснить, является ли этот текст идентификатором переменной |
| 7 | Дан текст. Выяснить, является ли этот текст десятичной записью целого числа |
| 8 | Дан текст. Группы символов, разделенные пробелами (одним или несколькими) и не содержащие пробелов внутри себя, будем называть словами. Опередить количество слов в тексте |
| 9 | Дан текст. Определить, сколько раз в тексте встречается каждая буква русского алфавита |
| 10 | Дан текст. Определить, сколько раз в нем встречается заданное слово |
| 11 | Дан текст. Заменить в тексте заданное слово на символ * |
| 12 | Дан текст. Заменить в тексте заданное(1) слово на заданное(2) слово |
| 13 | Дан текст. Убрать из него заданную букву |
| 14 | Дан текст. Выполнить простейшее шифрование текста. Каждый символ, код которого есть x, заменить символом с кодом 256–x |
| 15 | Дан текст, содержащий слова, разделенные несколькими пробелами. Получить текст, содержащий слова, разделенные одним пробелом |
| 16 | Дан текст. Выяснить, встречается ли в данном тексте заданная группа букв |
| 17 | Дан текст. Каждую строку текста записать в зеркальном виде |
| 18 | Дан текст. Определить сколько раз в тексте встречается каждая буква английского алфавита |
| 19 | Дан текст. Выяснить, сколько раз в нем встречается заданное слово |
| 20 | Дан текст. Выяснить процентное содержание в тексте заданной буквы |
| 21 | Дан текст. Выяснить процентное содержание в тексте заданного слова |
| 22 | Дан текст. Выяснить процентное содержание в тексте пробелов |
| 23 | Дан текст. Заменить в тексте все прописные буквы строчными |
| 24 | Дан текст. Заменить в тексте все строчные буквы прописными |
| 25 | Дан текст. Заменить в тексте все буквы строчными |
| 26 | Дан текст. Заменить в тексте все прописные буквы строчными |
| 27 | Считая заданный тест одной очень большой сточкой, записать ее зеркальное представление |
| 28 | Дан русский текст, записанный так, что каждый символ русского языка заменен на созвучный символ английского алфавита («транслитерация» текста). Получить текст на русском языке |
| 29 | Дан текст. Выполнить простейшее шифрование текста. Каждый символ, код которого есть x, заменить символом с кодом = F(x). Выбрать F(x) самостоятельно |
| 30 | Дан русский текст. Выполнить «транслитерацию» текста. Каждый символ русского языка заменить на созвучные символы английского алфавита |

## КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что будет, если в static блоке кода возникнет исключительная ситуация?

2. Какие виды исключений в Java вы знаете, чем они отличаются?

3. Назовите несколько классов из вершины иерархии исключений в Java.

4. Что такое Error? В каком случае используется Error. Приведите пример Error'a.

5. Какая конструкция используется в Java для обработки исключений?

6. Возможно ли использование блока try–finally (без catch)?

7. Всегда ли исполняется блок finally?

8. Предположим, есть блок try–finally. В блоке try возникло исключение, и выполнение переместилось в блок finally. В блоке finally тоже возникло исключение. Какое из двух исключений «выпадет» из блока try–finally? Что случится со вторым исключением?

9. Могли бы вы придумать ситуацию, когда блок finally не будет выполнен?

10. Предположим, есть метод, который может выбросить IOException и FileNotFoundException. В какой последовательности должны идти блоки catch? Сколько блоков catch будет выполнено?

11. Предположим, вам необходимо создать свой собственный класс Exception. Какими мотивами вы будете руководствоваться при выборе типа исключения: checked/unchecked?

12. Какие есть способы создания объекта String?

13. Напишите метод удаления данного символа из строки.

14. Как мы можем перевести строку в верхний регистр или нижний регистр?

15. Как сравнить две строки в Java?

16. Как преобразовать строку в символ и обратно?

17. Как преобразовать строку в массив байт и обратно?

18. Можем ли мы использовать строку в конструкции switch.

## СОДЕРЖАНИЕ ОТЧЕТА

1. Ф.И.О., группа, название лабораторной работы.
2. Цель работы.
3. Описание проделанной работы.

4. Результаты выполнения лабораторной работы.

5. Выводы.

Отчет и исходные коды запаковать в архив с названием по следующему шаблону:

Lab5–«группа, аббревиатура на латинице»–«Фамилия на латинице».

Пример: Lab5–13VS–Ivanov.zip.

### Порядок защиты лабораторной работы

1. Предоставление отчета о выполнении лабораторной работы.
2. Демонстрация программы.
3. Выполнение дополнительного задания.
4. Ответы на вопросы по теме текущей лабораторной работы.

## Лабораторная работа № 6. Обобщенные классы в Java, коллекции

**Цель работы.** Изучить методы работы с Java-generics. Изучить устройство и принципы работы с коллекциями в Java.

### Теоретическая часть

#### Введение в обобщенные классы

Java 5 (JDK 1.5) ввел принцип обобщений или параметризованных типов. Данная лабораторная работа знакомит с принципами обобщений и показывает примеры их использования. Далее рассматривается, как обобщения на самом деле реализованы в Java, и несколько проблем с применением обобщений.

#### Проблема безопасности типов

Java – строго типизированный язык. При программировании на Java во время компиляции надо знать, передается ли неверный тип параметра методу. Например, если определить

```
Dog aDog = aBookReference; // ошибка
```

где aBookReference – ссылка типа Book, не связанная с Dog, вы получите ошибку компиляции.

Однако, к сожалению, при появлении Java это не осуществлялось полностью в библиотеке Коллекции. Так, например, можно написать:

```
Vector vec = new Vector();
vec.add("hello");
vec.add(new Dog());
…
```

Не контролируется то, какой тип объекта помещается в Vector. Рассмотрим пример в следующем фрагменте кода:

```
import java.util.ArrayList;
import java.util.Iterator;

public class Test
{
    public static void main(String[] args)
    {
        ArrayList list = new ArrayList();
        populateNumbers(list);

        int total = 0;
        Iterator iter = list.iterator();
        while(iter.hasNext())
        {
            total += ((Integer) (iter.next())).intValue();
        }

        System.out.println(total);
    }

    private static void populateNumbers(ArrayList list)
    {
        list.add(new Integer(1));
        list.add(new Integer(2));
    }
}
```

В программе выше создается ArrayList, заполняется некоторыми целыми значениями Integer, а затем значения суммируются путем извлечения Integer из ArrayList.

Вывод из вышеприведенной программы – значение 3, как ожидалось.

Что, если изменить метод populateNumbers() следующим образом:

```
private static void populateNumbers(ArrayList list)
{
    list.add(new Integer(1));
    list.add(new Integer(2));
    list.add("hello");
}
```

Ошибок компиляции не будет. Однако программа не выполнится правильно. Выдастся следующая ошибка при выполнении:

```
Exception in thread "main" java.lang.ClassCastException:
  java.lang.String at com.agiledeveloper.Test.main(Test.java:17)…
```

До Java 5 в коллекциях не было безопасности типов.

**Что такое обобщения?**

В C++ есть такое прекрасное средство, как шаблоны. Шаблоны дают безопасность типов, в то же время, позволяя писать универсальный код, то есть не специфичный для какого-то конкретного типа. Хотя шаблоны C++ – очень мощный принцип, у него есть несколько недостатков. Во-первых, не все компиляторы хорошо поддерживают его. Во-вторых, он очень сложен в применении. В-третьих, его применение имеет ряд неприятных особенностей (так можно сказать о C++ в целом, но это другая история). Когда появился Java, обошлись без большинства сложных средств C++, таких как шаблоны и перегрузка оператора.

Наконец, в Java 5 было решено ввести обобщения. Хотя обобщения (возможность писать универсальный или обобщенный код, независимый от конкретного типа) в принципе похожи на шаблоны в C++, есть ряд отличий. Например, в отличие от C++, где генерируются разные классы для каждого параметризованного типа, в Java есть только один класс для каждого обобщенного типа, независимо от того, экземпляры скольких разных типов создаются посредством него.

Развитие обобщений в Java началось с проекта под названием GJ1 (обобщенный Java), начатого как расширение языка. Затем эту идею принял Процесс сообщества Java (JCP) в качестве Запроса спецификации Java (JSR) 142.

**Обобщенная безопасность типов**

Начнем с рассмотренного необобщенного примера, чтобы увидеть, как можно выиграть от обобщений. Переделаем вышеприведенный код, чтобы использовать обобщения. Измененный код показан ниже:

```
import java.util.ArrayList;
import java.util.Iterator;

public class Test
{
    public static void main(String[] args)
    {
        ArrayList<Integer> list = new ArrayList<Integer>();
        populateNumbers(list);

        int total = 0;
        for(Integer val : list)
        {
           total = total + val;
        }

        System.out.println(total);
    }

    private static void populateNumbers(ArrayList<Integer> list)
    {
        list.add(new Integer(1));
        list.add(new Integer(2));
        list.add("hello");
    }
}
```

Используется ArrayList <Integer> вместо ArrayList. Сейчас при компиляции кода выдается ошибка компиляции (рисунок 37).

```
Test.java:26: cannot find symbol
symbol  : method add(java.lang.String)
location: class java.util.ArrayList<java.lang.Integer>
        list.add("hello");
           ^
1 error
```

**Рисунок 37. – Ошибка компиляции**

Параметризованный тип ArrayList обеспечивает безопасность типов.

**Соглашения об именовании**

Во избежание путаницы между обобщенными параметрами и настоящими типами в коде надо придерживаться продуманного соглашения об именовании. Если мы придерживаемся продуманных соглашений Java и практик программирования, то наверняка не станем называть ваши классы одной буквой. Мы будем использовать смешанный регистр для имен классов, начиная с верхнего регистра. Ниже дано несколько соглашений, применяемых для обобщений:

– использовать букву E для элементов коллекции, как в определении:

```
public class PriorityQueue<E> {…}
```

– использовать буквы T, U, S и т.д. для универсальных типов.

**Написание обобщенных классов**

Синтаксис для написания обобщенного класса очень простой. Пример обобщенного класса:

```
public class Pair<E>
{
    private E obj1;
    private E obj2;

    public Pair(E element1, E element2)
    {
        obj1 = element1;
        obj2 = element2;
    }

    public E getFirstObject() { return obj1; }
    public E getSecondObject() { return obj2; }
}
```

Этот класс представляет собой пару значений некоторого обобщенного типа E. Рассмотрим несколько примеров использования данного класса:

```
// Правильное использование
Pair<Double> aPair = new Pair<Double>(new Double(1), new Double(2.2));
```

Если попытаться создать объект с типами, которые не соответствуют, выдастся ошибка компиляции. Рассмотрим следующий пример:

```
// Неправильное использование
Pair<Double> anotherPair = new Pair<Double>(new Integer(1), new Double(2.2));
```

Здесь предпринимается попытка отправить экземпляр Integer и экземпляр Double экземпляру Pair. Однако это дает ошибку компиляции.

## Обобщения и заменяемость

**Обобщения соблюдают принцип заменяемости Лискова**. Поясним на примере. Допустим, есть корзина фруктов. В нее можно добавить апельсины, бананы, виноград и т.д. Теперь создадим корзину бананов. В нее должно быть разрешено добавлять только бананы. Она должна запрещать добавление других типов фруктов. Банан является фруктом, т.е. банан наследуется от фрукта. Должна ли корзина бананов наследоваться от корзины фруктов, как показано на рисунке 38?

**Рисунок 38. – Примеры наследования**

Если бы корзина бананов наследовалась от корзины фруктов, то можно было бы заставить ссылку типа корзина фруктов ссылаться на экземпляр корзины бананов. Затем с помощью этой ссылки можно было бы добавить банан в корзину, но можно было бы добавить и апельсин. Тогда как добавление банана в корзину бананов правильно, добавление апельсина – нет. В лучшем случае это вызовет исключение времени выполнения. Однако код, использующий корзину фруктов, может не знать, как выйти из этой ситуации. Корзина бананов не заменяема там, где используется корзина фруктов.

Обобщения соблюдают этот принцип. Рассмотрим следующий пример:

```
Pair<Object> objectPair = new Pair<Integer>(new Integer(1), new Integer(2));
```

Этот код даст ошибку компиляции:

```
Error:  line (9) incompatible types found   :
com.agiledeveloper.Pair<java.lang.Integer> required:
    com.agiledeveloper.Pair<java.lang.Object>
```

Что, если мы хотим обрабатывать другой тип Pair как один тип? Это будет рассмотрено позже в разделе «Подстановочный знак».

Прежде чем оставить данную тему, посмотрим на одно странное поведение. Тогда как

```
Pair<Object> objectPair = new Pair<Integer>(new Integer(1), new Integer(2));
```

запрещено, однако следующее – разрешено:

```
Pair objectPair = new Pair<Integer>(new Integer(1), new Integer(2));
```

Pair без параметризованного типа является необобщенной формой класса Pair. Каждый обобщенный класс также имеет необобщенную форму, поэтому к нему можно обращаться из необобщенного кода. Это обеспечивает обратную совместимость с существующим кодом или кодом, не пере-

несенным для использования обобщений. В то время как такая совместимость имеет определенное преимущество, она может вызвать путаницу и проблемы с безопасностью типов.

**Обобщенные методы**

Наряду с классами, методы также могут быть параметризованными. Рассмотрим следующий пример кода:

```
public static <T> void filter(Collection<T> in, Collection<T> out)
{
    boolean flag = true;
    for(T obj : in)
    {
        if(flag)
        {
            out.add(obj);
        }
        flag = !flag;
    }
}
```

Метод filter() копирует чередующиеся элементы из коллекции in в коллекцию out. <T> перед void показывает, что метод является обобщенным методом с <T>, означающим параметризованный тип. Рассмотрим пример применения данного обобщенного метода:

```
ArrayList<Integer> lst1 = new ArrayList<Integer>();
lst1.add(1);
lst1.add(2);
lst1.add(3);

ArrayList<Integer> lst2 = new ArrayList<Integer>();
filter(lst1, lst2);
System.out.println(lst2.size());
```

ArrayList lst1 заполняется тремя значениями, а затем его содержимое фильтруется (копируется) в другой ArrayList lst2. Размер lst2 после вызова метода filter() равен 2. Теперь посмотрим на немного другой вызов:

```
ArrayList<Double> dblLst = new ArrayList<Double>();
filter(lst1, dblLst);
```

Здесь получаем ошибку компиляции:

```
Error:
line (34) <T>filter(java.util.Collection<T>,java.util.Collection<T>)
in com.agiledeveloper.Test cannot be applied to
(java.util.ArrayList<java.lang.Integer>,
java.util.ArrayList<java.lang.Double>)
```

Ошибка говорит, что невозможно отправить ArrayList разных типов этому методу. Хорошо, однако попробуем следующей фрагмент:

```
ArrayList<Integer> lst3 = new ArrayList<Integer>();
ArrayList lst = new ArrayList();
lst.add("hello");
filter(lst, lst3);
System.out.println(lst3.size());
```

Этот код компилируется без ошибок, и вызов lst3.size() возвращает 1. Почему он скомпилировался, и что здесь происходит? Компилятор старается обеспечить вызовы обобщенных методов, если это возможно. В данном случае, рассматривая lst3 как простой ArrayList, то есть без параметризованного типа (смотрите последний абзац в разделе «Обобщения и заменяемость» выше), он в состоянии вызвать метод filter.

Это может привести к ряду проблем. Добавим еще один оператор к примеру выше. При начале набора с клавиатуры интегрированная среда разработки (IDE) подсказывает код, как показано на рисунке 39.

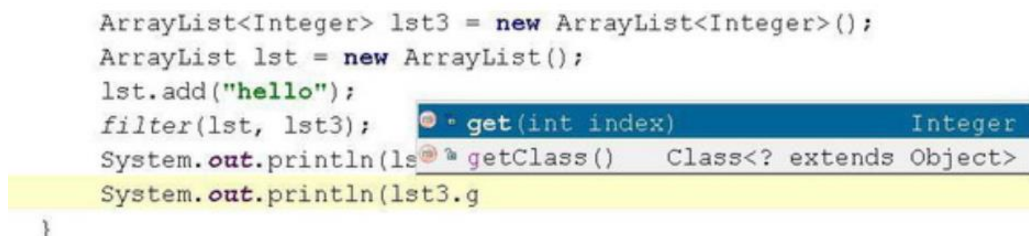


**Рисунок 39. – Использование метода get()**

Она говорит, что вызов метода get() принимает индекс и возвращает Integer. Ниже приведен готовый пример кода:

```
ArrayList<Integer> lst3 = new ArrayList<Integer>();
ArrayList lst = new ArrayList();
lst.add("hello");
filter(lst, lst3);
System.out.println(lst3.size());
System.out.println(lst3.get(0));
```

Что произойдет при выполнении этого кода? Этот кусок кода даст следующий вывод:

```
1
hello
```

Почему? Пока краткий ответ заключается в том, что хотя завершение кода предположило, что возвращается Integer, в реальности возвращаемый тип – Object. Поэтому строка "hello" сумела пройти без ошибки.

Что произойдет, если добавить следующий фрагмент кода:

```
for(Integer val: lst3)
{
    System.out.println(val);
}
```

Здесь запрашивается Integer из коллекции. Этот код сгенерирует ClassCastException. В то время как предполагается, что обобщения делают код безопасным по отношению к типам, данный пример показывает, как можно легко, умышленно или случайно, обойти это и, в лучшем случае, получить исключение времени выполнения или, в худшем случае, получить код, работающий неправильно.

**Верхние пределы**

Скажем, надо написать простой обобщенный метод для определения наибольшего из двух параметров. Прототип метода выглядел бы так:

```
public static <T> T max(T obj1, T obj2)
```

Использовали бы его, как показано ниже:

```
System.out.println(max(new Integer(1), new Integer(2)));
```

Как доделать реализацию метода max()? Попытаемся сделать это в следующем примере:

```
public static <T> T max(T obj1, T obj2)
{
    if (obj1 > obj2) // ошибка
    {
        return obj1;
    }
    return obj2;
}
```

Это не сработает. **Оператор** > не определен в ссылках. Как же тогда сравнить два объекта? Вспоминается интерфейс Comparable(сравнимый). По-

чему бы не использовать интерфейс Comparable для выполнения этой задачи, например:

```
public static <T> T max(T obj1, T obj2)
{
    // Не изящный код
    Comparable c1 = (Comparable) obj1;
    Comparable c2 = (Comparable) obj2;

    if (c1.compareTo(c2) > 0)
    {
        return obj1;
    }
    return obj2;
}
```

Несмотря на то, что этот код работает, есть две проблемы. Во-первых, он некрасив. Во-вторых, приходится учитывать случай, когда приведение к Comparable не удается. Поскольку так сильна зависимость от типа, реализующего этот интерфейс, почему бы не попросить компилятор навязать его? Именно это делают верхние пределы. Ниже приведен пример кода:

```
public static <T extends Comparable> T max(T obj1, T obj2)
{
    if (obj1.compareTo(obj2) > 0)
    {
        return obj1;
    }
    return obj2;
}
```

Компилятор проверит, чтобы убедиться, что параметризованный тип, заданный при вызове этого метода, реализует интерфейс Comparable. Если попытаться вызвать max() с экземплярами некоторого типа, не реализующего интерфейс Comparable, выдастся строгая ошибка компиляции.

**Подстановочный знак**

Перейдем к более интересным принципам обобщений. Рассмотрим следующий пример:

```
public abstract class Animal
{
    public void playWith(Collection<Animal> playGroup)
    {

    }
}

public class Dog extends Animal
{
    public void playWith(Collection<Animal> playGroup)
    {
    }
}
```

Класс Animal(животное) имеет метод playWith(), принимающий коллекцию Animal. Dog(собака), расширяющая Animal, переопределяет этот метод. Попробуем использовать класс Dog в примере:

```
Collection<Dog> dogs = new ArrayList<Dog>();

Dog aDog = new Dog();
aDog.playWith(dogs); //ошибка
```

Здесь создается экземпляр Dog и отправляется коллекция Dog его методу playWith(). Выдается ошибка компиляции:

```
Error:  line (29) cannot find symbol
method playWith(java.util.Collection<com.agiledeveloper.Dog>)
```

Причина состоит в том, что коллекцию Dog нельзя рассматривать как коллекцию Animal, которую ожидает метод playWith() (смотрите раздел «Обобщения и заменяемость» выше). Однако было бы логично иметь возможность отправить коллекцию Dog этому методу. Как это сделать? Здесь вступает в дело подстановочный знак или неизвестный тип.

Оба метода playMethod() (в Animal и Dog) изменяются следующим образом:

```
public void playWith(Collection<?> playGroup)
```

Collection не имеет тип Animal. Вместо этого она имеет неизвестный тип (?). Неизвестный тип – не Object, он просто неизвестный или неопределенный.

Теперь код

```
aDog.playWith(dogs);
```

компилируется без ошибок. Однако есть проблема. Также можно написать:

```
ArrayList<Integer> numbers = new ArrayList<Integer>();
aDog.playWith(numbers);
```

Изменение, сделанное, чтобы позволить отправить коллекцию Dog методу playWith(), теперь позволяет отправить и коллекцию Integer. Если разрешить это, получится странная собака. Как сказать, что компилятор должен разрешать коллекции Animal или коллекции любого типа, расширяющего

Animal, но не любую коллекцию других типов? Это позволяет осуществить применение верхних пределов, как показано ниже:

```
public void playWith(Collection<? extends Animal> playGroup)
```

Ограничение применения подстановочных знаков состоит в том, что разрешено извлекать элементы из Collection<?>, но нельзя добавлять элементы в такую коллекцию – компилятор не знает, с каким типом имеет дело.

**Нижние пределы**

Рассмотрим последний пример. Допустим, надо скопировать элементы из одной коллекции в другую. Ниже приведен код первой попытки сделать это:

```
public static <T> void copy(Collection<T> from, Collection<T> to) {…}
```

Попытаемся использовать данный метод:

```
ArrayList<Dog> dogList1 = new ArrayList<Dog>();
ArrayList<Dog> dogList2 = new ArrayList<Dog>();
//…
copy(dogList1, dogList2);
```

В этом коде копируются Dog из одного Dog ArrayList в другой. Так как Dog является Animal, Dog может находиться в Dog ArrayList и в Animal ArrayList. Следующий код копирует из Dog ArrayList в Animal ArrayList.

```
ArrayList<Animal> animalList = new ArrayList<Animal>();
copy(dogList1,  animalList);
```

Однако при компиляции этого кода выдается ошибка:

```
Error:
line (36) <T>copy(java.util.Collection<T>,java.util.Collection<T>)
in com.agiledeveloper.Test cannot be applied
to (java.util.ArrayList<com.agiledeveloper.Dog>,
java.util.ArrayList<com.agiledeveloper.Animal>)
```

Как заставить его работать? Здесь помогают нижние пределы. Второй аргумент Copy должен иметь тип T или любой тип, являющийся базовым типом T. Код выглядит так:

```
public static <T> void copy(Collection<T> from, Collection<? super T> to)
```

Здесь сказано, что принимаемый второй коллекцией тип является типом T или его супертипом.

**Непроверенное предупреждение**

Компилятор Java предупреждает, если не может проверить безопасность типов. Это происходит при смешении обобщенного и необобщенного кода (это плохая идея). Разработка приложений без проверки таких предупреждений является риском. Лучше рассматривать предупреждения как ошибки.

Рассмотрите следующий пример:

```
public class Test
{
    public static void foo1(Collection c)
    {
    }

    public static void foo2(Collection<Integer> c)
    {
    }

    public static void main(String[] args)
    {
        Collection<Integer> coll = new ArrayList<Integer>();
        foo1(coll);

        ArrayList lst = new ArrayList();
        foo2(lst);
    }
}
```

Имеется метод foo1, принимающий традиционный Collection(коллекция) в качестве параметра. Метод foo2, напротив, принимает обобщенную версию Collection. Объект традиционного ArrayList отправляется методу foo2. Поскольку ArrayList может содержать объекты разных типов, внутри метода foo2, компилятор не в состоянии гарантировать, что Collection<Integer> будет содержать только экземпляры Integer. В данном случае компилятор выдает предупреждение, показанное ниже:

```
Предупреждение: строка (22) [не проверено] обнаружено непроверенное преобразование:
           java.util.ArrayList требуется:
java.util.Collection<java.lang.Integer>
```

Хотя получить это предупреждение лучше, чем не быть предупрежденным о потенциальной проблеме, было бы лучше, если бы это была ошибка вместо предупреждения. Используйте флаг компиляции – Xlint, чтобы точно не упустить это предупреждение.

Есть еще одна проблема. В методе main отправляется обобщенный Collection из Integer методу foo1. Хотя компилятор не жалуется на это, это опасно. Что, если внутри метода foo1 в коллекцию добавляются объекты типов, отличных от Integer? Это нарушит безопасность типов.

Возникает вопрос, почему компилятор разрешил рассматривать обобщенный тип как традиционный тип. Причина в том, что на уровне байтового кода нет понятия обобщений.

**Ограничения**

В использовании обобщений есть ряд ограничений. Нельзя создать массив обобщенных коллекций. Любой массив коллекций шаблонов разрешен, но является опасным с позиции безопасности типов. Нельзя создать обобщение элементарного типа. Например, ArrayList<int> запрещен. Нельзя создать параметризованные статические поля внутри обобщенного класса или иметь статические методы с параметризованными типами в качестве параметров. Например, рассмотрите следующее:

```
class MyClass<T>
{
    private Collection<T> myCol1; // нормально
    private static Collection<T> myCol2; // ошибка
}
```

Внутри обобщенного класса нельзя создать экземпляр объекта или массива объектов параметризованного типа. Например, если имеется обобщенный класс MyClass<T> внутри метода этого класса, нельзя написать:

```
new T();
или

new T[10];
```

Можно сгенерировать исключение обобщенного типа, однако в блоке catch придется использовать конкретный тип вместо обобщенного.

Можно унаследовать класс от другого обобщенного класса, но нельзя унаследовать от параметрического типа. Например, тогда как

```
class MyClass2<T> extends MyClass<T>
{
}
```

нормально,

```
class MyClass2<T> extends T
{
}
```

нормальным не является.

Запрещено наследовать от двух экземпляров одного и того же обобщенного типа. Например, тогда как

```
class MyList implements MyCollection<Integer>
{
    //...
}
```

нормально,

```
class MyList implements MyCollection<Integer>, MyCollection<Double>
{
    //...
}
```

нормальным не является.

Какова причина этих ограничений? Они обусловлены способом реализации обобщений. Понимание механизмов реализации обобщений в Java дает понимание, откуда эти ограничения берутся и почему они существуют.

**Методы-мосты**

Иногда компилятору приходится добавлять классу так называемый метод-мост (bridge method), чтобы справиться с ситуациями, когда результат очистки типов в перегруженном методе подкласса не совпадает с тем, что получается при очистке в суперклассе. В этом случае генерируется метод, который использует очистку типов суперкласса, и этот метод вызывает соответствующий метод подкласса, выполняющий очистку. Конечно, методы-мосты появляются только на уровне байт-кода, невидимы для вас и недоступны для непосредственного вызова.

Несмотря на то, что методы-мосты – это не то, в чем вы обычно нуждаетесь и с чем имеете дело, все же полезно рассмотреть ситуацию, в которой они генерируются. Взгляните на следующую программу:

```
// Ситуация, в которой генерируется метод-мост,
class Gen<T>
{
    T ob; // объявить объект типа T

    Gen(T o) // Передать конструктору ссылку на // объект типа T.
    {
        ob = o;
    }

    // Возвращает ob.
    T getob () { return ob; }
}

// Подкласс Gen.
class Gen2 extends Gen<String>
{
    Gen2(String o)
    {
        super (o);
    }

    // String-ориентированная перегрузка getob().
    String getob()
    {
        System.out.print("Вызван String getob(): "); return ob;
    }
}

// Демонстрация ситуации, в которой необходим метод-мост,
class BridgeDemo
{
    public static void main(String args[])
    {
        // Создать объект Gen2 для Strings.
        Gen2 strOb2 = new Gen2("Обобщенный тест");
        System.out.println(strOb2.getob());
    }
)
```

В этой программе Gen2 расширяет Gen, но делает это с использованием специфичной String-версии Gen, как показывает следующее объявление:

```
class Gen2 extends Gen<String> {
```

Более того, внутри Gen2 метод getob() переопределен с типом возврата String:

```
//String-ориентированная перегрузка getob().
String getob()
{
    System.out.print("Вызван String getob(): ");
    return ob;
}
```

Все это совершенно допустимо. Единственная проблема в том, что из-за очистки типов ожидаемая форма getob() будет выглядеть так:

```
Object getobO { // ...
```

Чтобы справиться с этой проблемой, компилятор генерирует метод-мост с показанной выше сигнатурой, который вызывает String-версию. То есть,

если вы посмотрите на интерфейс класса Gen2 с помощью javap, то увидите следующие методы:

```
class Gen2 extends Gen
{
     Gen2(java.lang.String);
     java. lang. String getobO;
     java.lang.Object getobO; // метод-мост
}
```

Как видно, сюда включен метод-мост.

Последнее замечание о методах-мостах. Обратите внимание, что единственная разница между двумя методами getob() заключается в типе возврата. Обычно это вызывает ошибку, но поскольку это происходит не в исходном коде, проблема не возникает, и JVM успешно справляется с этой ситуацией.

## Классы-коллекции

### Понятие коллекции

Для хранения большого количества однотипных данных могут использоваться массивы, но они не всегда являются идеальным решением. Во-первых, длина массива задается заранее, и в случае, если количество элементов заранее неизвестно, придется либо выделять память «с запасом», либо предпринимать сложные действия по переопределению массива. Во-вторых, элементы массива имеют жестко заданное размещение в его ячейках, поэтому, например, удаление элемента из массива не является простой операцией.

В программировании давно и эффективно используются такие структуры данных, как стек, очередь, список, множество и т.д., объединенные общим названием коллекция. Коллекция – это группа элементов с операциями добавления, извлечения и поиска элемента. Механизм работы операций существенно различается в зависимости от типа коллекции. Например, элементы стека упорядочены в последовательность, добавление нового элемента может происходить только в конец этой последовательности, и получить можно только элемент, находящийся в конце (то есть, добавленный последним). Очередь, напротив, позволяет получить лишь первый элемент (элементы добавляются в один конец последовательности, а «забираются» с другого). Другие коллекции (например, список) позволяют получить элемент из любого места последовательности, а множество вообще не упорядочивает элементы и позволяет (помимо добавления и удаления) только узнать, содержится ли в нем данный элемент.

Язык Java предоставляет библиотеку стандартных коллекций, которые собраны в пакете java.util, поэтому нет необходимости программировать их самостоятельно.

При работе с коллекциями главное избегать ошибки начинающих — пользоваться наиболее универсальной коллекцией вместо той, которая необходима для решения задачи, например, списком вместо стека. Если логика работы программы такова, что данные должны храниться в стеке (появляться и обрабатываться в обратной последовательности), следует использовать именно стек. В этом случае вы не сможете нарушить логику обработки данных, обратившись напрямую к середине последовательности, а значит, шанс появления трудно обнаруживаемых ошибок резко уменьшается.

Чтобы выбрать коллекцию, которая лучше всего подходит условию задачи, необходимо знать особенности каждой из них. Эти знания являются обязательными для любого программиста, поскольку без применения тех или иных коллекций редко обходится любая современная задача.

Так как большинство коллекций параметризированы, то мы при описании интерфейсов и классов будем указывать T, что означает любой класс, которым вы параметризируете коллекцию.

На вершине иерархии в Java Collection Framework располагаются 2 интерфейса: Collection и Map. Эти интерфейсы разделяют все коллекции, входящие во фреймворк, на две части по типу хранения данных: простые последовательные наборы элементов и наборы пар «ключ-значение» (словари) (рисунок 40).

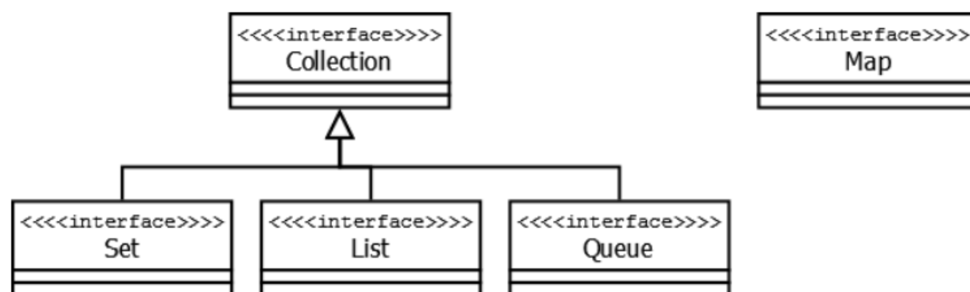


**Рисунок 40. – Интерфейсы, наследуемые от Collection**

**Collection** – этот интерфейс находится в составе JDK с версии 1.2 и определяет основные методы работы с простыми наборами элементов, которые будут общими для всех его реализаций (например, size(), isEmpty(), add(E e) и др.). Интерфейс был слегка доработан с приходом дженериков в Java 1.5. Также в версии Java 8 было добавлено несколько новых методов для работы с **лямбдами** (такие как stream(), parallelStream(), removeIf(Predicate<? super E> filter) и др.).

*Collection:*

- add(T e) – добавить элемент e;
- clear() – очистить;
- addAll(Collection col) – добавить все элементы другой коллекции со схожим типом данных;
- contains(Object o) – содержит ли коллекция элемент;
- isEmpty() – возвращает, пуста ли коллекция;
- remove(Object o) – удаляет элемент;
- removeAll(Collection col) – удаляет все элементы, которые есть в коллекции col;
- size() – возвращает количество элементов в коллекции;
- containsAll(Collection col) – возвращает, содержатся ли все элементы col в коллекции;
- toArray(T[] a) – возвращает массив, который содержит все элементы коллекции, на вход принимает массив, который будет заполнен ими;
- retainAll(Collection col) – удаляет все элементы, не принадлежащие col;
- toArray() – возвращает массив Object-ов, который содержит все элементы коллекции.

**Map**

Данный интерфейс также находится в составе JDK с версии 1.2 и предоставляет разработчику базовые методы для работы с данными вида «ключ-значение» (рисунок 41). Так же, как и Collection, он был дополнен дженериками в версии Java 1.5, и в версии Java 8 появились дополнительные методы для работы с лямбдами, а также методы, которые зачастую реализовались в логике приложения (getOrDefault(Object key, V defaultValue), putIfAbsent(K key, V value)).

*Map:*

- V get(KeyK) – возвращает объект, соответствующий указанному ключу, или значение null, если карта не содержит указанный ключ. Ключ может быть равен null;
- V put(KeyK, ValueV) – добавляет ключ и значение к карте. Если такой ключ уже имеется, то новый объект заменяет предыдущий, связанный с этим ключом. Этот метод возвращает предыдущее значение объекта или значение null, если ключ не содержался в карте ранее. Ключ может быть равен null, но значение должно быть отлично от null;

– void putAll(Map<? extendsK, ? extendsV> entries) – добавляет все элементы заданной карты к текущей;

– boolean containsKey(Objectkey) – возвращает значение true, если в карте имеется указанный ключ;

– boolean containsValue(Objectvalue) – возвращает значение true, если в карте имеется указанное значение;

– Set<Map.Entry<K, V>> entrySet() – возвращает представление карты в виде множества объектов Map.Entry, т.е. пар «ключ-значение». Из этого представления можно удалять элементы, при этом они удаляются и из карты, но добавлять их нельзя;

– Set<K> keySet() – возвращает представление карты в виде множества всех ключей. Из этого представления можно удалять элементы, при этом ключи и соответствующие им значения автоматически удаляются из карты, но добавлять новые элементы нельзя;

– Collection<V> values() – возвращает представление карты в виде множества всех значений. Из этого представления можно удалять элементы, при этом значения и соответствующие им ключи автоматически удаляются из карты, но добавлять новые элементы нельзя.

Подробнее про Map: [https://www.examclouds.com/ru/java/java-core-russian/map](https://www.examclouds.com/ru/java/java-core-russian/map).

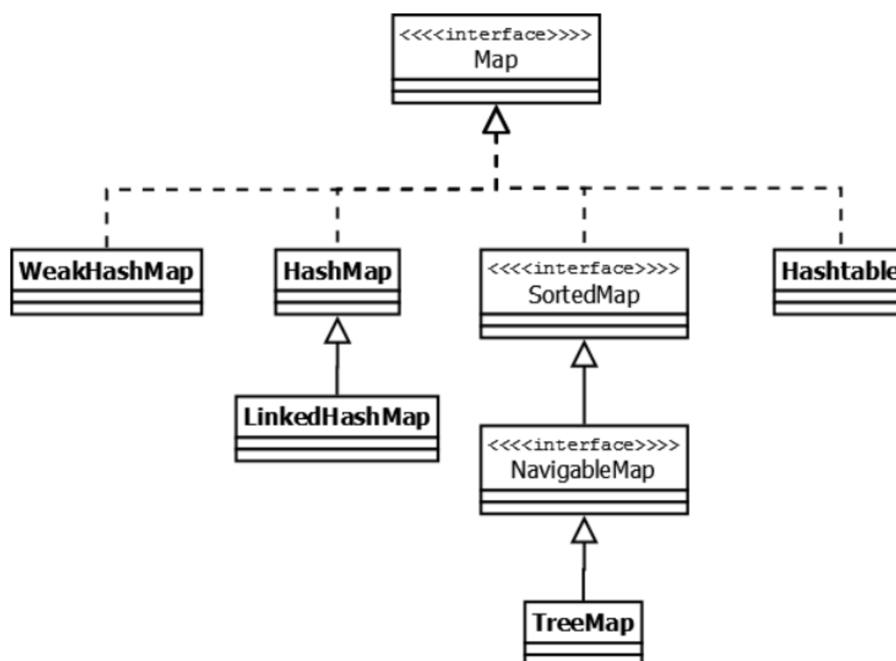


**Рисунок 41. – Наследники интерфейса Map**

**Hashtable** – реализация такой структуры данных, как хэш-таблица. Она не позволяет использовать null в качестве значения или ключа. Эта кол-

лекция была реализована раньше, чем Java Collection Framework, но впоследствии была включена в его состав. Как и другие коллекции из Java 1.0, Hashtable является синхронизированной (почти все методы помечены как synchronized). Из-за этой особенности у нее имеются существенные проблемы с производительностью и, начиная с Java 1.2, в большинстве случаев рекомендуется использовать другие реализации интерфейса Map ввиду отсутствия у них синхронизации.

**HashMap** – коллекция является альтернативой Hashtable. Двумя основными отличиями от Hashtable являются то, что HashMap не синхронизирована и HashMap позволяет использовать null в качестве как ключа, так и значения. Так же, как и Hashtable, данная коллекция не является упорядоченной: порядок хранения элементов зависит от хэш-функции. Добавление элемента выполняется за константное время O(1), но время удаления, получения зависит от распределения хэш-функции. В идеале является константным, но может быть и линейным O(n). Более подробную информацию о HashMap можно почитать здесь: http://habrahabr.ru/post/128017/.

**LinkedHashMap** — это упорядоченная реализация хэш-таблицы. Здесь, в отличии от HashMap, порядок итерирования равен порядку добавления элементов. Данная особенность достигается благодаря двунаправленным связям между элементами (аналогично LinkedList). Но это преимущество имеет также и недостаток – увеличение памяти, которое занимает коллекция. Более подробная информация изложена в этой статье: http://habrahabr.ru/post/129037/.

**TreeMap** – реализация Map, основанная на красно-черных деревьях. Как и LinkedHashMap, является упорядоченной. По умолчанию, коллекция сортируется по ключам с использованием принципа **"natural ordering"**, но это поведение может быть настроено под конкретную задачу при помощи объекта Comparator, который указывается в качестве параметра при создании объекта TreeMap.

**WeakHashMap** – реализация хэш-таблицы, которая организована с использованием weak references. Другими словами, Garbage Collector автоматически удалит элемент из коллекции при следующей сборке мусора, если на ключ этого элемента нет жестких ссылок.

**Интерфейс List**

Реализации этого интерфейса представляют собой упорядоченные коллекции (рисунок 42).

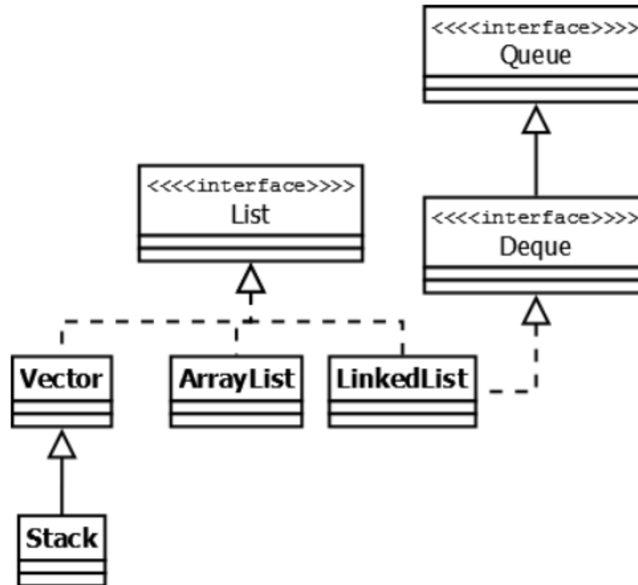


**Рисунок 42. – Интефейс List и его наследники**

Кроме того, разработчику предоставляется возможность доступа к элементам коллекции по индексу и по значению (так как реализации позволяют хранить дубликаты, результатом поиска по значению будет первое найденное вхождение).

*List:*

- get(int index) – получаем элемент по индексу;
- add(int index, т. e) – вставляет элемент в позицию;
- indexOf(Object obj) – возвращает первое вхождение элемента в список;
- lastIndexOf(Object obj) – возвращает последнее вхождение;
- set(int index, т. e) – заменяет элемент в позиции index;
- subList(int from, int to) – возвращает новый список, представляющий собой часть главного.

**Vector** – реализация динамического массива объектов. Позволяет хранить любые данные, включая null в качестве элемента. Vector появился в JDK версии Java 1.0, но, как и Hashtable, эту коллекцию не рекомендуется использовать, если не требуется достижения потокобезопасности. Потому как в Vector, в отличие от других реализаций List, все операции с данными являются синхронизированными. В качестве альтернативы часто применяется аналог – ArrayList.

**Stack** – данная коллекция является расширением коллекции Vector. Была добавлена в Java 1.0 как реализация стека LIFO (last-in-first-out). Является частично синхронизированной коллекцией (кроме метода добавления push()). После добавления в Java 1.6 интерфейса Deque, рекомендуется использовать именно реализации этого интерфейса, например, ArrayDeque.

**ArrayList** – как и Vector, является реализацией динамического массива объектов. Позволяет хранить любые данные, включая null в качестве элемента. Как можно догадаться из названия, его реализация основана на обычном массиве. Данную реализацию следует применять, если в процессе работы с коллекцией предплагается частое обращение к элементам по индексу. Из-за особенностей реализации поиндексное обращение к элементам выполняется за константное время O(1). Но данную коллекцию рекомендуется избегать, если требуется частое удаление/добавление элементов в середину коллекции. Подробный анализ и описание можно почитать здесь: http://habrahabr.ru/post/128269/.

**LinkedList** – еще одна реализация List. Позволяет хранить любые данные, включая null. Особенностью реализации данной коллекции является то, что в ее основе лежит двунаправленный связный список (каждый элемент имеет ссылку на предыдущий и следующий). Благодаря этому, добавление и удаление из середины, доступ по индексу, значению происходит за линейное время O(n), а из начала и конца – за константное O(1). Также, ввиду реализации, данную коллекцию можно использовать как стек или очередь. Для этого в ней реализованы соответсвующие методы. Статья с подробным анализом и описанием этой коллекции: http://habrahabr.ru/post/127864/.

**Интерфейс Set**

Интерфейс Set представляет собой неупорядоченную коллекцию, которая не может содержать дублирующиеся данные (рисунок 43).

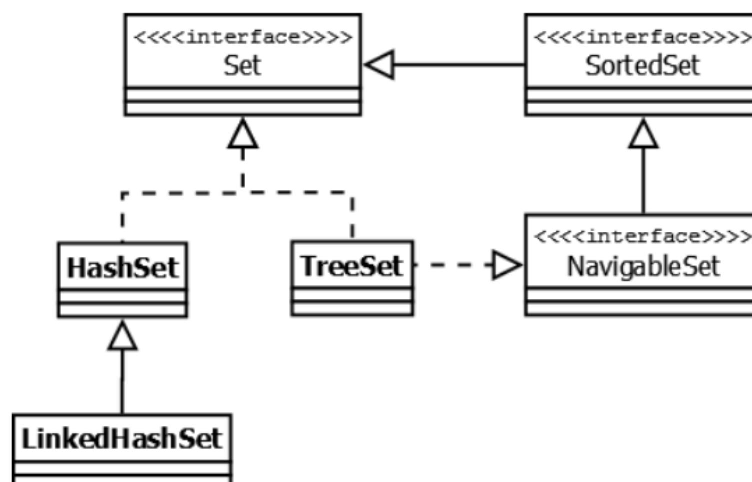


**Рисунок 43. – Интерфейс Set с наследниками**

**HashSet** – реализация интерфейса Set, базирующаяся на HashMap. Внутри использует объект HashMap для хранения данных. В качестве ключа

используется добавляемый элемент, а в качестве значения – объект-пустышка (new Object()). Из-за особенностей реализации порядок элементов не гарантируется при добавлении.

**LinkedHashSet** – отличается от HashSet только тем, что в основе лежит LinkedHashMap вместо HashSet. Благодаря этому отличию порядок элементов при обходе коллекции является идентичным порядку добавления элементов.

**TreeSet** – аналогично другим классам-реализациям интерфейса Set содержит в себе объект NavigableMap, что и обуславливает его поведение. Предоставляет возможность управлять порядком элементов в коллекции при помощи объекта Comparator, либо сохраняет элементы с использованием "natural ordering".

Set содержит все операции интерфейса Collection, но некоторые из них имеют другой смысл. Например, операция add добавляет только уникальные элементы, зато операции поиска элемента происходят быстрее, чем в списке.

**Интерфейс Queue**

Интерфейс Queue представлен на рисунке 44.

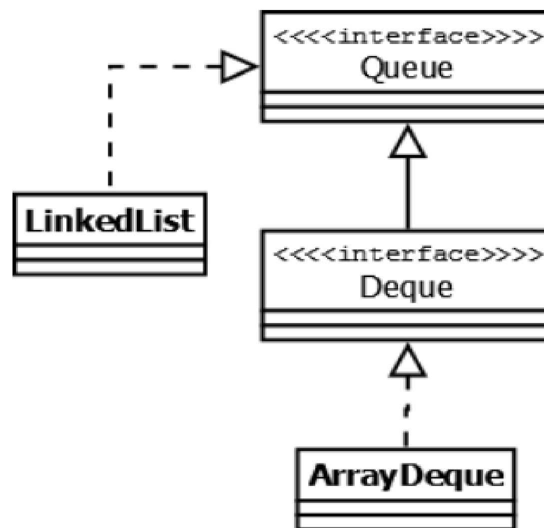


**Рисунок 44. – Интерфейс Queue с наследниками**

Этот интерфейс описывает коллекции с предопределенным способом вставки и извлечения элементов, а именно – очереди FIFO (first-in-first-out). Помимо методов, определенных в интерфейсе Collection, определяет дополнительные методы для извлечения и добавления элементов в очередь. Большинство реализаций данного интерфейса находится в пакете java.util.concurrent и подробно рассмотрены тут: http://habrahabr.ru/company/luxoft/blog/157273/.

*Queue:*

– poll() – возвращает первый элемент и удаляет его;

– peek() – возвращает первый элемент очереди, не удаляя его;
– offer(т. e) – добавляет элемент в конец очереди.

**Сравнение скорости работы**

Java Collections Framework содержит большое количество различных структур данных, доступных в JDK «из коробки», которые в большинстве случаев покрывают все потребности при реализации логики приложения. Сравнение временных характеристик основных коллекций, которые зачастую используются в разработке приложений, приведено в таблице 4.

Таблица 4. – Временная сложность выполнения операций в различных коллекциях

| | Временная сложность | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | *Среднее* | | | | *Худшее* | | | |
| | Индекс | Поиск | Вставка | Удаление | Индекс | Поиск | Вставка | Удаление |
| ArrayList | O(1) | O(n) | O(n) | O(n) | O(1) | O(n) | O(n) | O(n) |
| Vector | O(1) | O(n) | O(n) | O(n) | O(1) | O(n) | O(n) | O(n) |
| LinkedList | O(n) | O(n) | O(1) | O(1) | O(n) | O(n) | O(1) | O(1) |
| Hashtable | n/a | O(1) | O(1) | O(1) | n/a | O(n) | O(n) | O(n) |
| HashMap | n/a | O(1) | O(1) | O(1) | n/a | O(n) | O(n) | O(n) |
| LinkedHashMap | n/a | O(1) | O(1) | O(1) | n/a | O(n) | O(n) | O(n) |
| TreeMap | n/a | O(log(n)) | O(log(n)) | O(log(n)) | n/a | O(log(n)) | O(log(n)) | O(log(n)) |
| HashSet | n/a | O(1) | O(1) | O(1) | n/a | O(n) | O(n) | O(n) |
| LinkedHashSet | n/a | O(1) | O(1) | O(1) | n/a | O(n) | O(n) | O(n) |
| TreeSet | n/a | O(log(n)) | O(log(n)) | O(log(n)) | n/a | O(log(n)) | O(log(n)) | O(log(n)) |

**Итераторы**

Идея паттерна Iterator заключается в том, что к коллекции привязывается объект, который поможет обойти в некотором порядке все элементы коллекции. Рассмотрим интерфейс Iterator, реализующий данный подход:

– next() – возвращает следующий элемент коллекции, при этом итератор переходит на следующий элемент;
– hasNext() – возвращает, существует ли следующий элемент;
– remove() – удаляет текущий элемент.

Интерфейс Collection имеет метод iterator(), который возвращает итератор текущей коллекции. Поэтому обход произвольной коллекции можно реализовать, например, следующим образом:

```
1 Collection<Integer> c;
2 Iterator<Integer> iterator = c.iterator();
3 while (iterator.hasNext()) {
4     System.out.print(iterator.next());
5 }
```

Более интересный итератор применяется в случае, если мы имеем дело с упорядоченной коллекцией, т.е. коллекцией, которая реализует интерфейс List. Тогда в качестве итератора мы можем использовать ListIterator, который содержит следующие методы:

– previous() – возвращает предыдущий элемент из коллекции, при этом итератор переходит на предыдущий элемент;
– hasPrevious() – возвращает, существует ли предыдущий элемент;
– add(т. е) – добавляет новый элемент перед текущим элементом;
– set(т. е) – заменяет текущий элемент;
– nextIndex() – возвращает индекс следующего элемента;
– previousIndex() – возвращает индекс предыдущего элемента.

Для того чтобы получить ListIterator, необходимо вызвать listIterator() у объекта, который реализует интерфейс List.

## ЗАДАНИЕ К ЛАБОРАТОРНОЙ РАБОТЕ

1. Для хранения списков объектов из лабораторной работы № 4 использовать коллекции на базе интерфейсов List, Set, Map (задействовать все коллекции).
2. Реализовать простой класс со статическими generic-методами для вывода информации об объектах.
3. Реализовать простой класс с методами, использующими маски для поиска объектов в коллекциях.
4. Реализовать собственный generic-класс для хранения объектов. Добавить в данный класс методы для заполнения коллекций и методы для вывода коллекций.

## КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Есть ли взаимосвязь простых типов и generic-типов?
2. Что такое стирание (очистка) типов в generic-классах?
3. В каких случаях образуются методы-мосты?
4. Что такое маски (wildcards)?
5. Назовите основные интерфейсы коллекций и их имплементации.
6. Чем отличается ArrayList от LinkedList? В каких случаях лучше использовать первый, а в каких – второй?
7. Чем отличается HashMap от Hashtable?
8. Чем отличается ArrayList от Vector?

9. Особенности интерфейса Set.

10. Каким образом можно отсортировать коллекцию?

11. Как правильно удалить элемент из коллекции при итерации в цикле?

12. Как правильно удалить элемент из ArrayList (или другой коллекции) при поиске этого элемента в цикле?

## СОДЕРЖАНИЕ ОТЧЕТА

1. Ф.И.О., группа, название лабораторной работы.
2. Цель работы.
3. Описание проделанной работы.
4. Результаты выполнения лабораторной работы.
5. Выводы.

Отчет и исходные коды запаковать в архив с названием по следующему шаблону:

Lab6–«группа, аббревиатура на латинице»–«Фамилия на латинице».

Пример: Lab6–13VS–Ivanov.zip.

### Порядок защиты лабораторной работы

1. Предоставление отчета о выполнении лабораторной работы.
2. Демонстрация программы.
3. Выполнение дополнительного задания.
4. Ответы на вопросы по теме текущей лабораторной работы.

## Лабораторная работа № 7. Потоки ввода/вывода

**Цель работы.** Изучить методы работы потоками данных в Java. Изучить механизм сериализации данных и работу с файлами.

### Теоретическая часть

#### Введение в потоки ввода/вывода

Отличительной чертой многих языков программирования является работа с файлами и потоками. В Java основной функционал работы с потоками сосредоточен в классах из пакета java.io.

Ключевым понятием здесь является понятие **потока**, хотя понятие «поток» в программировании довольно перегружено и может обозначать мно-

жество различных концепций. В данном случае применительно к работе с файлами и вводом/выводом мы будем говорить о потоке (stream) как об абстракции, которая используется для чтения или записи информации (файлов, сокетов, текста консоли и т.д.).

Понятие потока оказалось настолько удобным и облегчающим программирование ввода/вывода, что в Java предусмотрена возможность создания потоков, направляющих символы или байты не на внешнее устройство, а в массив или из массива, т.е. связывающих программу с областью оперативной памяти. Более того, можно создать поток, связанный со строкой типа string, находящейся, опять-таки, в оперативной памяти. Кроме того, можно создать канал (pipe) обмена информацией между подпроцессами.

Объект, из которого можно считать данные, называется **потоком ввода**, а объект, в который можно записывать данные, – **потоком вывода**. Например, если надо считать содержание файла, то применяется поток ввода, а если надо записать в файл – то поток вывода.

В Java существует 2 типа потоков данных (рисунок 45):

– символьные потоки (**character-streams**, последовательности 16-битовых символов Unicode), содержащие символы;

– байтовые потоки (**byte-streams**), содержащие 8-ми битную информацию.

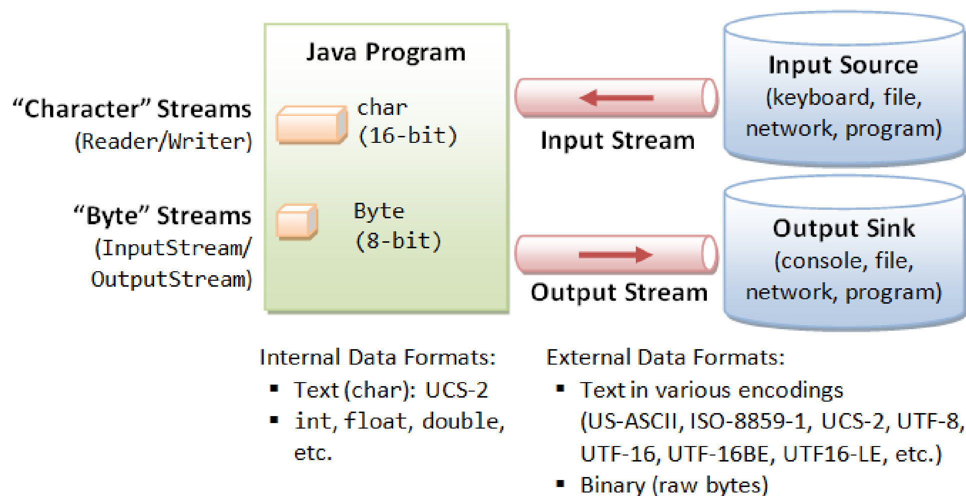


**Рисунок 45. – Потоки данных Java**

**Классы для работы с потоками**

В Java есть целых четыре иерархии классов для создания, преобразования и слияния потоков. Во главе иерархии четыре класса, непосредственно расширяющих класс Object:

– **Reader** – абстрактный класс, в котором собраны самые общие методы **символьного ввода**;

– **Writer** – абстрактный класс, в котором собраны самые общие методы **символьного вывода**;

– **InputStream** – абстрактный класс с общими методами **байтового ввода**;

– **OutputStream** – абстрактный класс с общими методами **байтового вывода**.

Все остальные классы, работающие с потоками, являются наследниками этих абстрактных классов.

**Класс InputStream**

Класс InputStream (public abstract class InputStream implements Closeable) является базовым для всех классов, управляющих байтовыми потоками ввода. Рассмотрим его основные методы:

– **int available() throws IOException** – возвращает количество байт, доступных для чтения в потоке;

– **void close() throws IOException** – закрывает поток;

– **int read() throws IOException** – возвращает целочисленное представление следующего байта в потоке. Когда в потоке не останется доступных для чтения байт, данный метод возвратит число –1;

– **int read(byte[] buffer) throws IOException** – считывает байты из потока в массив buffer. После чтения возвращает число считанных байт. Если ни одного байта не было считано, то возвращается число –1;

– **int read(byte[] buffer, int offset, int length) throws IOException** – считывает некоторое количество байт, равное length, из потока в массив buffer. При этом считанные байты помещаются в массиве, начиная со смещения offset, то есть с элемента buffer[offset]. Метод возвращает число успешно прочитанных байт;

– **long skip(long number) throws IOException** – пропускает в потоке при чтении некоторое количество байт, которое равно number;

– **boolean markSupported()** – возвращает true, если данный поток поддерживает операции mark/reset;

– **void mark(int readlimit)** – ставит метку в текущей позиции входного потока, которую можно будет использовать до тех пор, пока из потока не будет прочитано readlimit байт;

– **void reset() throws IOException** – возвращает указатель потока на установленную ранее метку.

Иерархия классов байтового потока ввода представлена на рисунке 46.

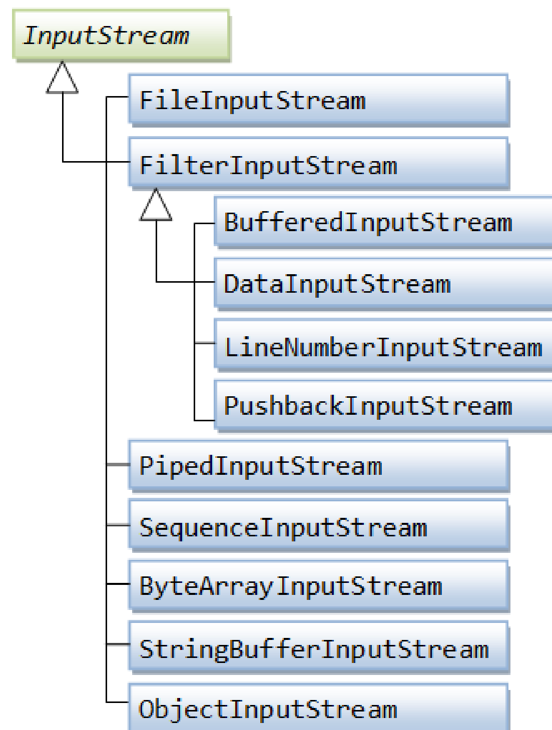


**Рисунок 46. – Иерархия наследников InputStream**

**Класс OutputStream**

Класс OutputStream (public abstract class OutputStream implements Closeable, Flushable) является базовым классом для всех классов, которые работают с бинарными потоками записи. Свою функциональность он реализует через следующие методы:

– **int close() throws IOException** – закрывает поток;

– **void flush() throws IOException** – очищает буфер вывода, записывая все его содержимое;

– **void write(int b) throws IOException** – записывает в выходной поток один байт, который представлен целочисленным параметром b;

– **void write(byte[] buffer) throws IOException** – записывает в выходной поток массив байтов buffer;

– **void write(byte[] buffer, int offset, int length) throws IOException** – записывает в выходной поток некоторое число байт, равное length, из массива buffer, начиная со смещения offset, то есть с элемента buffer[offset].

На рисунке 47 представлена иерархия классов байтового потока вывода.

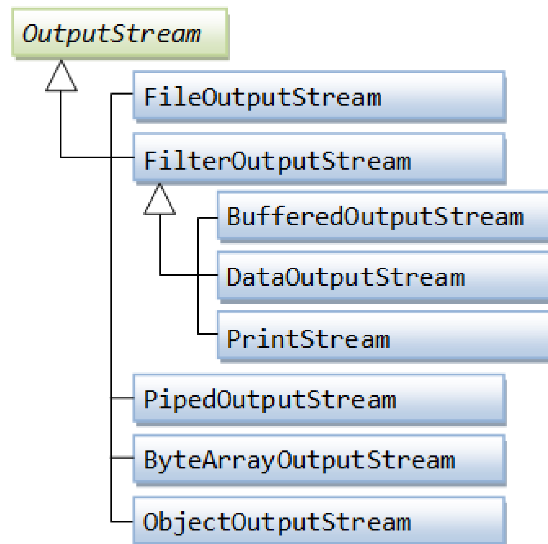


**Рисунок 47. – Иерархия наследников OutputStream**

Пример копирования файлов с помощью байтовых потоков данных:

```
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;

public class CopyBytes
{
    public static void main(String[] args) throws IOException
    {
        FileInputStream in = null;
        FileOutputStream out = null;

        try
        {
            in = new FileInputStream("xanadu.txt");
            out = new FileOutputStream("outagain.txt");

            int c;

            while ((c = in.read()) != -1)
            {
                out.write(c);
            }

        }
        finally
        {
            if (in != null)
                in.close();

            if (out != null)
                out.close();
        }
    }
}
```

На рисунке 48 представлена схема работы приложения.

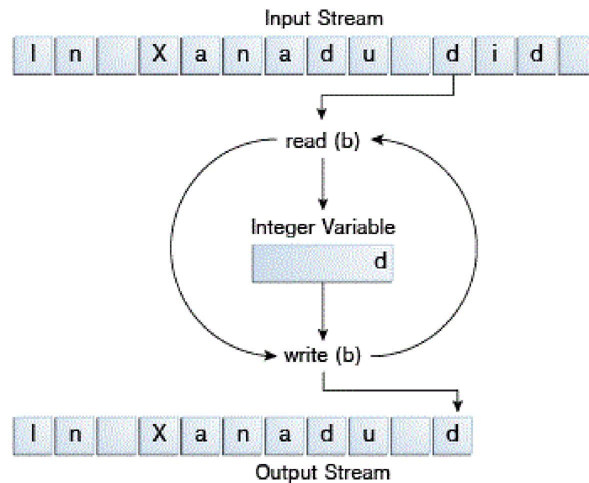


**Рисунок 48. – Схема процесса работы потоков**

**Абстрактный класс Reader**

Абстрактный класс Reader (public abstract class Reader implements Readable, Closeable) предоставляет функционал для чтения текстовой информации. Рассмотрим его основные методы:

– **void close() throws IOException** – закрывает поток ввода;

– **int read() throws IOException** – возвращает целочисленное представление следующего символа в потоке. Если таких символов нет, и достигнут конец файла, то возвращается число –1;

– **int read(char[] buffer) throws IOException** – считывает в массив buffer из потока символы, количество которых равно длине массива buffer. Возвращает количество успешно считанных символов. При достижении конца файла возвращает –1;

– **int read(CharBuffer buffer) throws IOException** – считывает в объект **CharBuffer** из потока символы. Возвращает количество успешно считанных символов. При достижении конца файла возвращает –1;

– **absract int read(char[] buffer, int offset, int count) throws IOException** – считывает в массив buffer, начиная со смещения offset, из потока символы, количество которых равно count;

– **long skip(long count) throws IOException** – пропускает количество символов, равное count. Возвращает число успешно пропущенных символов;

– **boolean markSupported()** – возвращает true, если данный поток поддерживает операции **mark/reset**;

– **void mark(int readAheadlimit) throws IOException** – помечает текущее положение потока, параметр указывает количество символов, которые могут быть прочитаны до тех пор, пока метка не станет недействительной;

– **void reset() throws IOException** – возвращает поток в положение, указанное меткой;

– **boolean ready() throws IOException** – возвращает true, если в потоке есть данные, доступные для чтения.

Иерархия классов символьного потока ввода представлена на рисунке 49.

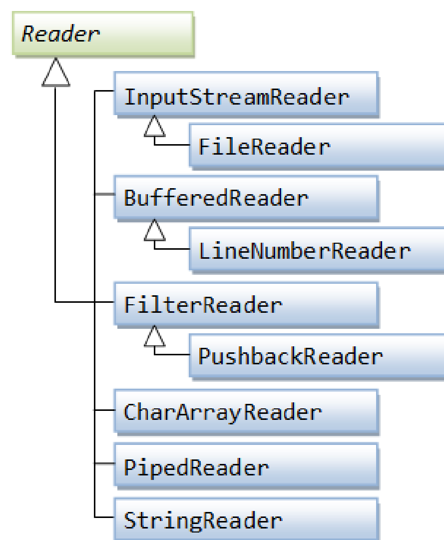


**Рисунок 49. – Иерархия наследников класса Reader**

**Абстрактный класс Writer**

Класс Writer (public abstract class Writer implements Appendable, Closeable, Flushable) определяет функционал для всех символьных потоков вывода. Его основные методы:

– **Writer append(char c) throws IOException** – добавляет в конец выходного потока символ c. Возвращает объект **Writer**;

– **Writer append(CharSequence chars) throws IOException** – добавляет в конец выходного потока набор символов chars. Возвращает объект **Writer**;

– **Writer append(CharSequence csq, int start, int end) throws IOException** – добавляет в конец выходного потока последовательность символов (от start до end) из набора символов. Возвращает объект **Writer**;

– **abstract void close() throws IOException** – закрывает поток;

– **abstract void flush() throws IOException** – очищает буферы потока. Используйте **flush** для сбрасывания буферизованных данных в поток;

– **void write(int c) throws IOException** – записывает в поток один символ, который имеет целочисленное представление;

– **void write(char[] buffer) throws IOException** – записывает в поток массив символов;

– **absract void write(char[] buffer, int off, int len) throws IOException** – записывает в поток только несколько символов из массива buffer. Причем количество символов равно len, а отбор символов из массива начинается с индекса off;

– **void write(String str) throws IOException** – записывает в поток строку;

– **void write(String str, int off, int len) throws IOException** – записывает в поток из строки некоторое количество символов, которое равно len, причем отбор символов из строки начинается с индекса off;

Иерархия классов символьного потока вывода представлена на рисунке 50.

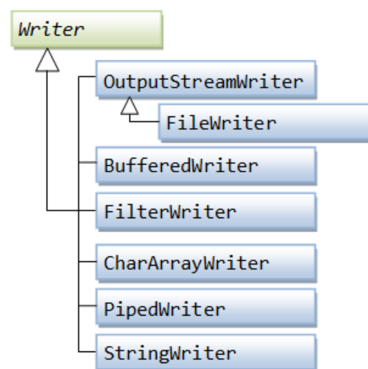


**Рисунок 50. – Иерархия наследников класса Writer**

Пример программы копирования файлов с помощью символьных потоков данных:

```
1  import java.io.FileReader;
2  import java.io.FileWriter;
3  import java.io.BufferedReader;
4  import java.io.PrintWriter;
5  import java.io.IOException;
6
7  public class CopyLines
8  {
9      public static void main(String[] args) throws IOException
10     {
11
12         BufferedReader inputStream = null;
13         PrintWriter outputStream = null;
14
15         try
16         {
17             inputStream = new BufferedReader(new FileReader("xanadu.txt"));
18             outputStream = new PrintWriter(new FileWriter("characteroutput.txt"));
19
20             String l;
21
22             while ((l = inputStream.readLine()) != null)
23             {
24                 outputStream.println(l);
25             }
26         }
27         finally
28         {
29             if (inputStream != null)
30                 inputStream.close();
31
32             if (outputStream != null)
33                 outputStream.close();
34         }
35     }
36 }
```

**Закрытие потоков**

При завершении работы с потоком его надо закрыть с помощью метода **close**(), который определен в интерфейсе **Closeable**. Метод **close** имеет следующее определение:

**void close() throws IOException**.

Этот интерфейс уже реализуется в классах **InputStream**, **OutputStream**, **Reader** и **Writer**, а через них и во всех классах потоков.

При закрытии потока освобождаются все выделенные для него ресурсы, например, файл. В случае, если поток окажется не закрыт, может происходить утечка памяти.

Есть два способа закрытия файла. Первый, традиционный, заключается в использовании блока **try..catch..finally**. Например, считаем данные из файла:

```
1   import java.io.*;
2
3   public class FilesApp
4   {
5       public static void main(String[] args)
6       {
7           FileInputStream fin = null;
8
9           try
10          {
11              fin = new FileInputStream("C://SomeDir//Hello.txt");
12
13              int i = -1;
14
15              while((i=fin.read()) != -1)
16              {
17                  System.out.print((char)i);
18              }
19          }
20          catch(IOException ex)
21          {
22              System.out.println(ex.getMessage());
23          }
24          finally
25          {
26              try
27              {
28                  if (fin!=null) fin.close();
29              }
30              catch(IOException ex)
31              {
32                  System.out.println(ex.getMessage());
33              }
34          }
35      }
36  }
```

Поскольку при открытии или считывании файла может произойти ошибка ввода/вывода, то код считывания помещается в блок **try**. И чтобы быть уверенным, что поток в любом случае закроется, даже если при работе с ним возникнет ошибка, вызов метода **close**() помещается в блок **finally**. И так как метод **close**() также в случае ошибки может генерировать исключение **IOException**, то его вызов тоже помещается во вложенный блок **try..catch**.

Начиная с Java 7, можно использовать еще один способ, который автоматически вызывает метод **close**. Этот способ заключается в использовании конструкции **try-with-resources** (try-с-ресурсами). Данная конструкция работает с объектами, которые реализуют интерфейс **AutoCloseable**. Так как все классы потоков реализуют интерфейс **Closeable**, который в свою очередь наследуется от **AutoCloseable**, то их также можно использовать в данной конструкции.

Итак, перепишем предыдущий пример с использованием конструкции **try-with-resources**:

```
1  import java.io.*;
2
3  public class FilesApp
4  {
5      public static void main(String[] args)
6      {
7          try(FileInputStream fin = new FileInputStream("C://SomeDir//Hello.txt"))
8          {
9              int i=-1;
10
11             while((i=fin.read())!=-1)
12             {
13                 System.out.print((char)i);
14             }
15         }
16         catch(IOException ex)
17         {
18             System.out.println(ex.getMessage());
19         }
20     }
21 }
```

Синтаксис конструкции следующий: **try(название_класса имя_переменной = конструктор_класса).** Данная конструкция также не исключает использования блоков catch.

После окончания работы в блоке try у ресурса (в данном случае у объекта **FileInputStream**) автоматически вызывается метод **close**().

Если нам надо использовать несколько потоков, которые после выполнения надо закрыть, то мы можем указать объекты потоков через точку с запятой:

```
1  try(FileInputStream fin=new FileInputStream("C://SomeDir//Hello.txt");
2      FileOutputStream fos = new FileOutputStream("C://SomeDir//Hello2.txt"))
3  {
4      //..................
5  }
```

**Обзор основных классов байтовых потоков данных**

Обзор классов, реализующих потоковые интерфейсы, представлен на рисунке 51.

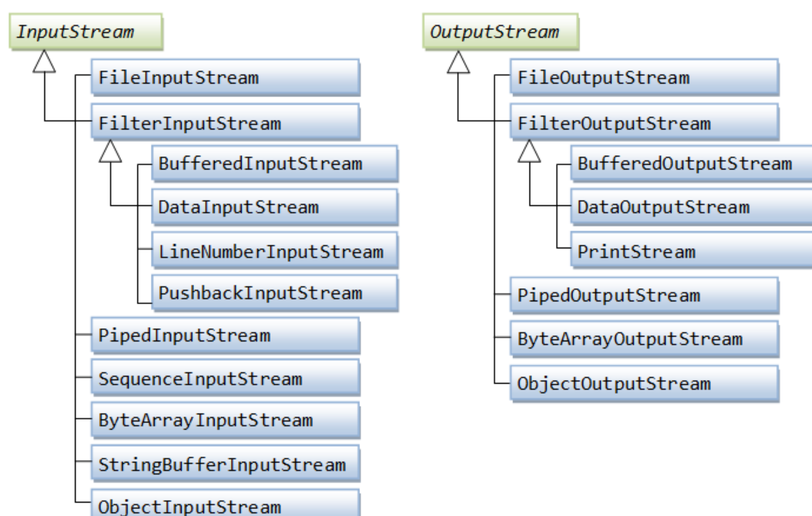


**Рисунок 51. – Обзор классов, реализующих потоковые интерфейсы**

Описание классов-наследников потоков приведено в таблице 5.

Таблица 5. – Описание классов-наследников потоков

| Поточный класс | Значение |
|---|---|
| 1 | 2 |
| **BufferedInputStream** | Класс для буферизации ввода |
| **BufferedOutputStream** | Класс для буферизации вывода |
| **ByteArrayInputStream** | Поток, читающий из массива байт |
| **ByteArrayOutputStream** | Поток, пишущий в массив байт |
| **DataInputStream** | Поток ввода, который содержит методы для чтения данных стандартных типов Java |
| **DataOutputStream** | Поток вывода, который содержит методы для записи данных стандартных типов Java |
| **FileInputStream** | Поток, читающий байты из файла |
| **FileOutputStream** | Поток, пишущий байты в файл |
| **FilterInputStream** | Реализует InputStream (шаблон адаптер) |
| **FilterOutputStream** | Реализует OutputStream (шаблон адаптер) |
| **InputStream** | Абстрактный класс, который описывает поточный ввод |
| **LineNumberInputStream** | Расширяет функциональность InputStream тем, что дополнительно производит подсчет, сколько строк было считано из потока |
| **OutputStream** | Абстрактный класс, который описывает поточный вывод |
| **ObjectInputStream** | Поток сериализации объектов |
| **ObjectOutputStream** | Поток десериализации объектов |
| **PipedInputStream** | Поток, читающий данные из канала |
| **PipedOutputStream** | Поток, пишущий данные в канал |
| **PrintStream** | Используется для конвертации и записи строк в байтовый поток |
| **PushbackInputStream** | Фильтр позволяет вернуть во входной поток считанные из него данные |

Окончание таблицы 5

| 1 | 2 |
|---|---|
| **SequenceInputStream** | Считывает данные из других двух и более входных потоков |
| **StringBufferInputStream** | Производит считывание данных, получаемых преобразованием символов строки в байты |

## Чтение и запись файлов

### Классы FileInputStream и FileOutputStream. Чтение файлов и класс FileInputStream

Для считывания данных из файла предназначен класс **FileInputStream**, который является наследником класса **InputStream** и поэтому реализует все его методы.

Для создания объекта **FileInputStream** мы можем использовать ряд конструкторов. Наиболее используемая версия конструктора в качестве параметра принимает путь к считываемому файлу:

**FileInputStream(String fileName) throws FileNotFoundException**.

Если файл не может быть открыт, например, по указанному пути такого файла не существует, то генерируется исключение **FileNotFoundException**.

Считаем данные из файла и выведем на консоль в следующем примере:

```
import java.io.*;

public class FilesApp
{
    public static void main(String[] args)
    {
        try(FileInputStream fin=new FileInputStream("C://SomeDir//note.txt"))
        {
            System.out.println("Размер файла: " + fin.available() + " байт(а)");

            int i=-1;

            while((i=fin.read())!=-1)
            {
                System.out.print((char)i);
            }
        }
        catch(IOException ex)
        {
            System.out.println(ex.getMessage());
        }
    }
}
```

Подобным образом можно считать данные в массив и затем производить с ним манипуляции:

```
byte[] buffer = new byte[fin.available()];

// считаем файл в буфер
fin.read(buffer, 0, fin.available());

System.out.println("Содержимое файла:");

for(int i=0; i<buffer.length;i++)
{
    System.out.print((char)buffer[i]);
}
```

**Запись файлов и класс FileOutputStream**

Класс **FileOutputStream** предназначен для записи байтов в файл. Он является производным от класса **OutputStream**, поэтому наследует всю его функциональность.

Например, запишем в файл строку:

```
1  import java.io.*;
2
3  public class FilesApp
4  {
5      public static void main(String[] args)
6      {
7          String text = "Hello world!"; // строка для записи
8
9          try(FileOutputStream fos=new FileOutputStream("C://SomeDir//notes.txt"))
10         {
11             // перевод строки в байты
12             byte[] buffer = text.getBytes();
13
14             fos.write(buffer, 0, buffer.length);
15         }
16         catch(IOException ex)
17         {
18             System.out.println(ex.getMessage());
19         }
20     }
21 }
```

Для создания объекта **FileOutputStream** используется конструктор, принимающий в качестве параметра путь к файлу для записи. Так как здесь записываем строку, то ее надо сначала перевести в массив байтов. И с помощью метода **write** строка записывается в файл.

Необязательно записывать весь массив байтов. Используя перегрузку метода write(), можно записать и одиночный байт:

**fos.write(buffer[0]); // запись первого байта**.

Совместим оба класса и выполним чтение из одного и запись в другой файл:

```
1  import java.io.*;
2
3  public class FilesApp
4  {
5      public static void main(String[] args)
6      {
7          try(FileInputStream fin=new FileInputStream("C://SomeDir//notes.txt");
8              FileOutputStream fos=new FileOutputStream("C://SomeDir//notes_new.txt"))
9          {
10             byte[] buffer = new byte[fin.available()];
11             // считываем буфер
12             fin.read(buffer, 0, buffer.length);
13             // записываем из буфера в файл
14             fos.write(buffer, 0, buffer.length);
15         }
16         catch(IOException ex)
17         {
18             System.out.println(ex.getMessage());
19         }
20     }
21 }
```

Классы **FileInputStream** и **FileOutputStream** предназначены прежде всего для записи двоичных файлов. И хотя они также могут использоваться для работы с текстовыми файлами, но все же для этой задачи больше подходят другие классы.

**Классы ByteArrayInputStream и ByteArrayOutputStream**

Для работы с массивами байтов – их чтения и записи – используются классы ByteArrayInputStream и ByteArrayOutputStream.

**Чтение массива байтов и класс ByteArrayInputStream**

Класс ByteArrayInputStream представляет входной поток, использующий в качестве источника данных массив байтов. Он имеет следующие конструкторы:

**ByteArrayInputStream(byte[] buf)**;

**ByteArrayInputStream(byte[] buf, int offset, int length)**.

В качестве параметров конструкторы используют массив байтов buf, из которого производится считывание, смещение относительно начала массива offset и количество считываемых символов length.

Считаем массив байтов и выведем его на экран:

```
1  import java.io.*;
2
3  public class FilesApp
4  {
5      public static void main(String[] args)
6      {
7          byte[] array1 = new byte[]{1, 3, 5, 7};
8          ByteArrayInputStream byteStream1 = new ByteArrayInputStream(array1);
9          int b;
10
11         while((b=byteStream1.read())!=-1)
12         {
13             System.out.println(b);
14         }
15
16         String text = "Hello world!";
17         byte[] array2 = text.getBytes();
18
19         ByteArrayInputStream byteStream2 = new ByteArrayInputStream(array2, 0, 5);
20         int c;
21
22         while((c=byteStream2.read())!=-1)
23         {
24             System.out.println((char)c);
25         }
26     }
27 }
```

В отличие от других классов потоков, для закрытия объекта ByteArrayInputStream не требуется вызывать метод close.

**Запись массива байт и класс ByteArrayOutputStream**

Класс ByteArrayOutputStream представляет поток вывода, использующий массив байтов в качестве места вывода.

Чтобы создать объект данного класса, мы можем использовать один из его конструкторов:

**ByteArrayOutputStream()**;

**ByteArrayOutputStream(int size)**.

Первая версия создает массив для хранения байтов длиной в 32 байта, а вторая версия создает массив длиной size.

Рассмотрим применение класса:

```
import java.io.*;

public class FilesApp
{
    public static void main(String[] args)
    {
        ByteArrayOutputStream baos = new ByteArrayOutputStream();
        String text = "Hello Wolrd!";
        byte[] buffer = text.getBytes();

        try
        {
            baos.write(buffer);
        }
        catch(Exception ex)
        {
            System.out.println(ex.getMessage());
        }

        // превращаем массив байтов в строку
        System.out.println(baos.toString());

        // получаем массив байтов и выводим по символьно
        byte[] array = baos.toByteArray();

        for(byte b : array)
        {
            System.out.print((char)b);
        }

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

Как и в других потоках вывода, в классе ByteArrayOutputStream определен метод write, который записывает в поток некоторые данные. В данном случае мы записываем в поток массив байтов. Этот массив байтов записывается в объекте ByteArrayOutputStream в защищенное поле buf, которое представляет также массив байтов (protected byte[] buf). Так как метод write может сгенерировать исключение, то вызов этого метода помещается в блок try..catch. Используя методы toString() и toByteArray(), можно получить массив байтов buf в виде текста или непосредственно в виде массива байт. С помощью метода writeTo мы можем вывести массив байт в другой поток. Данный метод в качестве параметра принимает объект OutputStream, в который производится запись массива байт:

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
String text = "Hello Wolrd!";
byte[] buffer = text.getBytes();

try
{
    baos.write(buffer);
}
catch(Exception ex)
{
    System.out.println(ex.getMessage());
}

try(FileOutputStream fos = new FileOutputStream("hello.txt"))
{
    baos.writeTo(fos);
}
catch(IOException e)
{
    System.out.println(e.getMessage());
}
```

Как и для объектов ByteArrayInputStream, для ByteArrayOutputStream не надо явным образом закрывать поток с помощью метода close.

## Буферизуемые потоки

### Классы BufferedInputStream и BufferedOuputStream

Для оптимизации операций ввода-вывода используются буферизуемые потоки. Эти потоки добавляют к стандартным специальный буфер в памяти, с помощью которого повышается производительность при чтении и записи потоков.

### Класс BufferedInputStream

Класс BufferedInputStream накапливает вводимые данные в специальном буфере без постоянного обращения к устройству ввода, например:

```
import java.io.*;

public class FilesApp
{
    public static void main(String[] args)
    {
        String text = "Hello world!";
        byte[] buffer = text.getBytes();

        ByteArrayInputStream in = new ByteArrayInputStream(buffer);

        try(BufferedInputStream bis = new BufferedInputStream(in))
        {
            int c;

            while((c=bis.read())!=-1)
            {
                System.out.print((char)c);
            }
        }
        catch(Exception e)
        {
            System.out.println(e.getMessage());
        }

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

Класс BufferedInputStream в конструкторе принимает объект InputStream. В данном случае таким объектом является экземпляр класса ByteArrayInputStream.

Как и все потоки ввода, BufferedInputStream обладает методом read(), который считывает данные. И здесь мы считываем с помощью метода read каждый байт из массива buffer.

Фактически все то же самое можно было сделать и с помощью одного ByteArrayInputStream, не прибегая к буферизированному потоку. Класс BufferedInputStream просто оптимизирует производительность при работе с потоком ByteArrayInputStream.

### Класс BufferedOutputStream

Класс BufferedOutputStream аналогично создает буфер для потоков вывода. Этот буфер накапливает выводимые байты без постоянного обра-

щения к устройству. И когда буфер заполнен, производится запись данных. Рассмотрим на примере:

```
import java.io.*;

public class FilesApp
{
    public static void main(String[] args)
    {
        String text = "Hello world!"; // строка для записи

        try(FileOutputStream out=new FileOutputStream("notes.txt");
            BufferedOutputStream bos = new BufferedOutputStream(out))
        {
            // перевод строки в байты
            byte[] buffer = text.getBytes();
            bos.write(buffer, 0, buffer.length);
        }
        catch(IOException ex)
        {
            System.out.println(ex.getMessage());
        }
    }
}
```

Класс BufferedOutputStream в конструкторе принимает в качестве параметра объект OutputStream – в данном случае это файловый поток вывода FileOutputStream. И также производится запись в файл. Опять же BufferedOutputStream не добавляет много новой функциональности, он просто оптимизирует действие потока вывода.

### Класс PrintStream

Класс PrintStream – это именно тот класс, который используется для вывода на консоль. Когда мы выводим на консоль некоторую информацию с помощью вызова System.out.println(), то тем самым мы задействуем **PrintStream**, так как переменная out в классе System как раз и представляет объект класса PrintStream, а метод println() – это метод класса PrintStream.

Но PrintStream полезен не только для вывода на консоль. Мы можем использовать данный класс для записи информации в поток вывода. Например, запишем информацию в файл:

```
import java.io.*;

public class FilesApp
{
    public static void main(String[] args)
    {
        String text = "Привет мир!"; // строка для записи

        try(FileOutputStream fos=new FileOutputStream("C://SomeDir//notes3.txt");
            PrintStream printStream = new PrintStream(fos))
        {
            printStream.println(text);
            System.out.println("Запись в файл произведена");
        }
        catch(IOException ex)
        {
            System.out.println(ex.getMessage());
        }
    }
}
```

В данном случае применяется форма конструктора PrintStream, которая в качестве параметра принимает поток вывода: PrintStream (OutputStream out). Кроме того, мы могли бы использовать ряд других форм конструктора, например, указывая названия файла для записи: PrintStream (string filename).

В качестве потока вывода используется объект FileOutputStream. С помощью метода println() производится запись информации в выходной поток – то есть в объект FileOutputStream. В случае с выводом на консоль с помощью System.out.println() в качестве потока вывода выступает консоль.

Для вывода информации в выходной поток PrintStream использует следующие методы:

- **println()** – вывод строковой информации с переводом строки;
- **print()** – вывод строковой информации без перевода строки;
- **printf()** – форматированный вывод.

Кроме того, как и любой поток вывода и наследник класса OutputStream, он имеет метод write, например:

```
1   import java.io.*;
2
3   public class FilesApp
4   {
5       public static void main(String[] args)
6       {
7           String s1 = "Привет мир!";
8           String s2="Hello World!";
9
10          try(PrintStream printStream = new PrintStream("C://SomeDir//notes3.txt"))
11          {
12              printStream.println(s1);
13              int i=2;
14              printStream.printf("Квадрат числа %d равен %d \n", i, i*i);
15              byte[] s2_toBytes = s2.getBytes();
16              printStream.write(s2_toBytes);
17              printStream.print("Конец");
18              System.out.println("Запись в файл произведена");
19          }
20          catch(IOException ex)
21          {
22              System.out.println(ex.getMessage());
23          }
24      }
25  }
```

### Классы DataOutputStream и DataInputStream

Классы DataOutputStream и DataInputStream позволяют записывать и считывать данные примитивных типов.

### Запись данных и DataOutputStream

Класс DataOutputStream представляет поток вывода и предназначен для записи данных примитивных типов, таких, как int, double и т.д. Для записи каждого из примитивных типов предназначен свой метод:

- **writeBoolean(boolean v)** – записывает в поток булевое однобайтовое значение;

– **writeByte(int v)** – записывает в поток 1 байт, который представлен в виде целочисленного значения;

– **writeChar(int v)** – записывает 2-байтовое значение char;

– **writeDouble(double v)** – записывает в поток 8-байтовое значение double;

– **writeFloat(float v)** – записывает в поток 4-байтовое значение float;

– **writeInt(int v)** – записывает в поток целочисленное значение int;

– **writeLong(long v)** – записывает в поток значение long;

– **writeShort(int v)** – записывает в поток значение short;

– **writeUTF(String str)** – записывает в поток строку в кодировке UTF–8.

**Считывание данных и DataInputStream**

Класс DataInputStream действует противоположным образом – он считывает из потока данные примитивных типов. Соответственно для каждого примитивного типа определен свой метод для считывания:

– **boolean readBoolean()** – считывает из потока булевое однобайтовое значение;

– **byte readByte()** – считывает из потока 1 байт;

– **char readChar()** – считывает из потока значение char;

– **double readDouble()** – считывает из потока 8-байтовое значение double;

– **float readFloat()** – считывает из потока 4-байтовое значение float;

– **int readInt()** – считывает из потока целочисленное значение int;

– **long readLong()** – считывает из потока значение long;

– **short readShort()** – считывает значение short;

– **String readUTF()** – считывает из потока строку в кодировке UTF–8;

– **int skipBytes(int n)** – пропускает при чтении из потока n байтов.

Рассмотрим применение классов на примере:

```
1  import java.io.*;
2
3  class Person
4  {
5      public String name;
6      public int age;
7      public double height;
8      public boolean married;
9
10     public Person(String n, int a, double h, boolean m)
11     {
12         this.name=n;
13         this.height=h;
14         this.age=a;
15         this.married=m;
16     }
17 }
```

```
18
19  public class FilesApp
20  {
21      public static void main(String[] args)
22      {
23          Person tom = new Person("Tom", 35, 1.75, true);
24
25          // запись в файл
26          try(DataOutputStream dos = new DataOutputStream(new FileOutputStream("data.bin")))
27          {
28             // записываем значения
29              dos.writeUTF(tom.name);
30              dos.writeInt(tom.age);
31              dos.writeDouble(tom.height);
32              dos.writeBoolean(tom.married);
33              System.out.println("Запись в файл произведена");
34          }
35          catch(IOException ex)
36          {
37              System.out.println(ex.getMessage());
38          }
39
40          // обратное считывание из файла
41          try(DataInputStream dos = new DataInputStream(new FileInputStream("data.bin")))
42          {
43              // записываем значения
44              String name = dos.readUTF();
45              int age = dos.readInt();
46              double height = dos.readDouble();
47              boolean married = dos.readBoolean();
48              System.out.printf("Человека зовут: %s , его возраст: %d , его рост: %f метров, женат/замужем: %b",
49                      name, age, height, married);
50          }
51          catch(IOException ex)
52          {
53              System.out.println(ex.getMessage());
54          }
55      }
56  }
```

Здесь мы последовательно записываем в файл данные объекта Person.

Объект DataOutputStream в конструкторе принимает поток вывода: DataOutputStream (OutputStream out). В данном случае в качестве потока вывода используется объект FileOutputStream, поэтому вывод будет происходить в файл. И с помощью выше рассмотренных методов типа writeUTF() производится запись значений в бинарный файл.

Затем происходит чтение ранее записанных данных. Объект DataInputStream в конструкторе принимает поток для чтения: DataInputStream(InputStream in). Здесь таким потоком выступает объект FileInputStream.

Классы символьных потоков представлены в таблице 6.

Таблица 6. – Список классов, реализующих потоки ввода/вывода

| Поточный класс | Значение |
|---|---|
| 1 | 2 |
| BufferedReader | Буферизированный символьный поток ввода |
| BufferedWriter | Буферизированный символьный поток вывода |
| CharArrayReader | Поток ввода, которой читает из символьного массива |
| CharArrayWrite | Выходной поток, который записывает в символьный массив |
| FileReader | Поток ввода, который читает из файла |
| FileWriter | Выходной поток, который записывает в файл |

Окончание таблицы 6

| 1 | 2 |
|---|---|
| FilterReader | Отфильтрованный поток ввода |
| FilterWriter | Отфильтрованный поток вывода |
| InputStreamReader | Поток ввода, который переводит байты в символы |
| LineNumberReader | Поток ввода, который считает строки |
| OutputStreamWriter | Поток ввода, который переводит символы в байты |
| PipedReader | Канал ввода |
| PipedWriter | Канал вывода |
| PrintWriter | Поток вывода, который поддерживает print() и println() |
| PushbackReader | Поток ввода, возвращающий символы в поток ввода |
| Reader | Абстрактный класс, который описывает символьный поток ввода |
| StringReader | Поток ввода, который читает из строки |
| StringWriter | Поток вывода, который записывает в строку |
| Writer | Абстрактный класс, который описывает символьный поток вывода |

**Чтение и запись текстовых файлов. FileReader и FileWriter**

Хотя с помощью ранее рассмотренных классов можно записывать текст в файлы, однако все же их возможностей для полноценной работы с текстовыми файлами недостаточно. И для этой цели служат совсем другие классы, которые являются наследниками абстрактных классов Reader и Writer.

**Запись файлов. Класс FileWriter**

Класс FileWriter является производным от класса Writer. Он используется для записи текстовых файлов.

Чтобы создать объект FileWriter, можно использовать один из следующих конструкторов:

**FileWriter(File file)**;

**FileWriter(File file, boolean append)**;

**FileWriter(FileDescriptor fd)**;

**FileWriter(String fileName)**;

**FileWriter(String fileName, boolean append)**.

Так, в конструктор передается либо путь к файлу в виде строки, либо объект File, который ссылается на конкретный текстовый файл. Параметр append указывает, должны ли данные дозаписываться в конец файла (если параметр равен true), либо файл должен перезаписываться.

Запишем в файл какой-нибудь текст, например:

```
import java.io.*;

public class FilesApp
{
    public static void main(String[] args)
    {
        try(FileWriter writer = new FileWriter("C:\\SomeDir\\notes3.txt", false))
        {
           // запись всей строки
            String text = "Мама мыла раму, раму мыла мама";
            writer.write(text);
            // запись по символам
            writer.append('\n');
            writer.append('E');
        }
        catch(IOException ex)
        {
            System.out.println(ex.getMessage());
        }
    }
}
```

В конструкторе использовался параметр append со значением false – то есть файл будет перезаписываться. Затем с помощью методов, определенных в базовом классе Writer, производится запись данных.

**Чтение файлов. Класс FileReader**

Класс FileReader наследуется от абстрактного класса Reader и предоставляет функциональность для чтения текстовых файлов.

Для создания объекта FileReader мы можем использовать один из его конструкторов:

**FileReader(String fileName)**;

**FileReader(File file)**;

**FileReader(FileDescriptor fd)**.

А используя методы, определенные в базовом классе Reader, произвести чтение файла:

```
import java.io.*;

public class FilesApp
{
    public static void main(String[] args)
    {
        try(FileReader reader = new FileReader("C:\\SomeDir\\notes3.txt"))
        {
            // читаем посимвольно
            int c;

            while((c=reader.read())!=-1)
            {
                System.out.print((char)c);
            }
        }
        catch(IOException ex)
        {
            System.out.println(ex.getMessage());
        }
    }
}
```

## Буферизируемые символьные потоки

**BufferedReader и BufferedWriter. Чтение текста и BufferedReader**

Класс **BufferedReader** считывает текст из символьного потока ввода, буферизируя прочитанные символы. Использование буфера призвано увеличить производительность чтения данных из потока.

Класс BufferedReader имеет следующие конструкторы:

**BufferedReader(Reader in)**;

**BufferedReader(Reader in, int sz)**.

Второй конструктор, кроме потока ввода, из которого производится чтение, также определяет размер буфера, в который будут считываться символы.

Так как BufferedReader наследуется от класса Reader, то он может использовать все те методы для чтения из потока, которые определены в Reader. И также BufferedReader определяет свой собственный метод readLine(), который позволяет считывать из потока построчно.

Рассмотрим применение BufferedReader:

```
1  import java.io.*;
2
3  public class FilesApp
4  {
5      public static void main(String[] args)
6      {
7          try(BufferedReader br = new BufferedReader (new FileReader("C:\\SomeDir\\notes3.txt")))
8          {
9              // чтение посимвольно
10             int c;
11             while((c=br.read())!=-1)
12             {
13                 System.out.print((char)c);
14             }
15         }
16         catch(IOException ex)
17         {
18             System.out.println(ex.getMessage());
19         }
20     }
21 }
```

Также можно считать текст построчно:

```
1  try(BufferedReader br = new BufferedReader(new FileReader("C:\\SomeDir\\notes3.txt")))
2  {
3      //чтение построчно
4      String s;
5      while((s=br.readLine())!=null)
6      {
7          System.out.println(s);
8      }
9  }
10 catch(IOException ex)
11 {
12     System.out.println(ex.getMessage());
13 }
```

**Чтение текста и BufferedWriter**

Класс **BufferedWriter** записывает текст в поток, предварительно буферизируя записываемые символы, тем самым снижая количество обращений к физическому носителю для записи данных.

Класс BufferedWriter имеет следующие конструкторы:

**BufferedWriter(Writer out)**;

**BufferedWriter(Writer out, int sz)**.

В качестве параметра он принимает поток вывода, в который надо осуществить запись. Второй параметр указывает на размер буфера.

Например, осуществим запись в файл:

```
1  import java.io.*;
2
3  public class FilesApp
4  {
5      public static void main(String[] args)
6      {
7          try(BufferedWriter bw = new BufferedWriter(new FileWriter("C:\\SomeDir\\notes4.txt")))
8          {
9              String text = "Привет мир!";
10             bw.write(text);
11         }
12         catch(IOException ex)
13         {
14             System.out.println(ex.getMessage());
15         }
16     }
17 }
```

**Предопределенные потоки**

Все программы Java автоматически импортируют пакет java.lang. Этот пакет определяет класс с именем System, инкапсулирующий некоторые аспекты исполнительной среды Java.

Класс System содержит также три предопределенные поточные переменные **in**, **out** и **err**. Эти поля объявлены в System со спецификаторами public и static. Объект System.out называют потоком стандартного вывода. По умолчанию с ним связана консоль. На объект System.in ссылаются как на стандартный ввод, который по умолчанию связан с клавиатурой. К объекту System.err обращаются как к стандартному потоку ошибок, который по умолчанию также связан с консолью. Однако эти потоки могут быть переназначены на любое совместимое устройство ввода/вывода.

**Упаковка (wrapping) потоков**

Потокам можно придать новые свойства, заключив один поток в оболочку другого потока.

Класс BufferedReader может быть применен для более *эффективного чтения* символов, массивов и строк.

```
BufferedReader in = new BufferedReader(new FileReader("foo.in"));
```

Классы **BufferedWriter** и **PrintWriter** могут быть использованы для более *эффективной записи* символов, массивов, строк и других типов данных.

```
BufferedWriter out = new BufferedWriter(new FileWriter("foo.out"));

PrintWriter out = new PrintWriter(new BufferedWriter(new FileWriter("foo.out")));
```

**Сериализация объектов**

Сериализация представляет процесс записи состояния объекта в поток, соответственно, процесс извлечения или восстановления состояния объекта из потока называется десериализацией. Сериализация очень удобна, когда идет работа со сложными объектами.

Процесс сериализации заключается в сериализации каждого поля объекта, но только в том случае, если это поле не имеет спецификатора **static** или **transient**. Поля, помеченные ими, не могут быть предметом сериализации.

**Интерфейс Serializable**

Сразу надо сказать, что сериализовать можно только те объекты, которые реализуют интерфейс Serializable. Этот интерфейс не определяет никаких методов, просто он служит указателем системе, что объект, реализующий его, может быть сериализован.

При попытке сериализовать объект, не реализующий этот интерфейс, будет брошено java.io.NotSerializableException.

После того как объект был сериализован (превращен в последовательность байт), его можно восстановить, при этом восстановление можно проводить на любой машине (вне зависимости от того, где проводилась сериализация).

При десериализации поле со спецификатором transient получает значение по умолчанию, соответствующее его типу, а поле со спецификатором static получает значение по умолчанию в случае отсутствия в области видимости объектов своего типа, а при их наличии получает значение, которое определено для существующего объекта.

При использовании Serializable десериализация происходит следующим образом: под объект выделяется память, после чего его поля заполняются значениями из потока. *Конструктор* объекта при этом *не вызывается*.

Сериализуемый объект может хранить ссылки на другие объекты, которые в свою очередь также могут хранить ссылки на другие объекты. И все ссылки тоже должны быть восстановлены при десериализации.

Важно, что если несколько ссылок указывают на один и тот же объект, то в восстановленных объектах эти ссылки также указывали на один и тот же объект.

Чтобы сериализованный объект не был записан дважды, механизм сериализации некоторым образом для себя помечает, что объект уже записан в граф, и когда в очередной раз попадется ссылка на него, она будет указывать на уже сериализованный объект.

Такой механизм необходим, чтобы иметь возможность записывать связанные объекты, которые могут содержать перекрестные ссылки. В таких случаях необходимо отслеживать, был ли объект уже сериализован, то есть нужно ли его записывать, или достаточно указать ссылку на него.

Если класс содержит в качестве полей другие объекты, то эти объекты также будут сериализовываться, и поэтому тоже должны быть сериализуемы. В свою очередь, сериализуемы должны быть и все объекты, содержащиеся в этих сериализуемых объектах и т.д.

Полный путь ссылок объекта по всем объектным ссылкам, имеющимся у него и у всех объектов, на которые у него имеются ссылки, и т.д., называется **графом исходного объекта**.

При десериализации производного класса, наследуемого от несериализуемого класса, вызывается конструктор без параметров родительского несериализуемого класса. И если такого конструктора не будет, при десериализации возникнет ошибка java.io.InvalidClassException. Конструктор же дочернего объекта, того, который мы десериализуем, не вызывается.

В процессе десериализации поля несериализуемых классов (родительских классов, не реализующих интерфейс Serializable) инициируются вызовом конструктора без параметров. Такой конструктор должен быть доступен из сериализуемого их подкласса. Поля сериализуемого класса будут восстановлены из потока.

Попытка десериализации объекта, класс которого к этому времени был изменен, приведет к возникновению **InvalidClassException**. Для отслеживания таких ситуаций каждому классу присваивается его идентификатор (ID) версии. Он представляет собой число long (длина 64 бита), полученное при помощи хэш-функции. Для его вычисления используются имена классов, всех реализуемых интерфейсов, всех методов и полей класса. При десериализации объекта, идентификаторы класса и идентификатор, взятый из потока сравниваются.

**Сериализация. Класс ObjectOutputStream**

Для сериализации объектов в поток используется класс **ObjectOutputStream**. Он записывает данные в поток. Для создания объекта ObjectOutputStream в конструктор передается поток, в который производится запись:

**ObjectOutputStream(OutputStream out)**.

Для записи данных ObjectOutputStream использует ряд методов, среди которых можно выделить следующие:

- **void close()** – закрывает поток;
- **void flush()** – очищает буфер и сбрасывает его содержимое в выходной поток;
- **void write(byte[] buf)** – записывает в поток массив байтов;
- **void write(int val)** – записывает в поток один младший байт из val;
- **void writeBoolean(boolean val)** – записывает в поток значение boolean;
- **void writeByte(int val)** – записывает в поток один младший байт из val;
- **void writeChar(int val)** – записывает в поток значение типа char, представленное целочисленным значением;
- **void writeDouble(double val)** – записывает в поток значение типа double;
- **void writeFloat(float val)** – записывает в поток значение типа float;
- **void writeInt(int val)** – записывает целочисленное значение int;
- **void writeLong(long val)** – записывает значение типа long;
- **void writeShort(int val)** – записывает значение типа short;
- **void writeUTF(String str)** – записывает в поток строку в кодировке UTF–8;
- **void writeObject(Object obj)** – записывает в поток отдельный объект.

Эти методы охватывают весь спектр данных, которые можно сериализовать. Например, сохраним в файл один объект класса Person:

```
1  import java.io.*;
2
3  class Person implements Serializable
4  {
5      public String name;
6      public int age;
7      public double height;
8      public boolean married;
9
10     Person(String n, int a, double h, boolean m)
11     {
12         name=n;
13         age=a;
14         height=h;
15         married=m;
16     }
17 }
18
19 public class FilesApp
20 {
21     public static void main(String[] args)
22     {
23         try(ObjectOutputStream oos = new ObjectOutputStream(new FileOutputStream("person.dat")))
24         {
25             Person p = new Person("Джон", 33, 178, true);
26             oos.writeObject(p);
27         }
28         catch(Exception ex)
29         {
30             System.out.println(ex.getMessage());
31         }
32     }
33 }
```

**Десериализация. Класс ObjectInputStream**

Класс ObjectInputStream отвечает за обратный процесс – чтение ранее сериализованных данных из потока. В конструкторе он принимает ссылку на поток ввода.

## ЗАДАНИЕ К ЛАБОРАТОРНОЙ РАБОТЕ

1. Сохранить коллекции объектов из ЛР № 5 в файл. Путь к файлу вводит пользователь.

2. Пользователь повторно вводит путь к файлу – из файла читаются объекты и выводятся на экран в виде списка.

3. Обработать все ошибки работы с файлами и каталогами.

## КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие существуют виды потоков ввода/вывода?
2. Назовите основные предки потоков ввода/вывода.
3. Что общего и чем отличаются следующие потоки: InputStream, OutputStream, Reader, Writer?
4. В каких пакетах лежат классы-потоки?

5. Что вы знаете о классах-надстройках?

6. Какой класс-надстройка позволяет читать данные из входного байтового потока в формате примитивных типов данных?

7. Какой класс-надстройка позволяет ускорить чтение/запись за счет использования буфера?

8. Какие классы дополнительно позволяют преобразовать байтовые потоки в символьные и обратно?

9. Какой класс предназначен для работы с элементами файловой системы (ЭФС)?

10. Какой символ дополнительно является разделителем при указании пути к ЭФС?

11. Как выбрать все ЭФС определенного каталога по критерию (например, с определенным расширением)? (дополнительно)

12. Что вы знаете об интерфейсе FileFilter? (дополнительно)

13. Что такое сериализация?

14. Какие условия «благополучной» сериализации объекта?

15. Какие классы позволяют архивировать объекты? (дополнительно)

## СОДЕРЖАНИЕ ОТЧЕТА

1. Ф.И.О., группа, название лабораторной работы.
2. Цель работы.
3. Описание проделанной работы.
4. Результаты выполнения лабораторной работы.
5. Выводы.

Отчет и исходные коды запаковать в архив с названием по следующему шаблону:

Lab7–«группа, аббревиатура на латинице»–«Фамилия на латинице».

Пример: Lab7–13VS–Ivanov.zip.

## Порядок защиты лабораторной работы

1. Предоставление отчета о выполнении лабораторной работы.
2. Демонстрация программы.
3. Выполнение дополнительного задания.
4. Ответы на вопросы по теме текущей лабораторной работы.

## Лабораторная работа № 8. Построение кросс-платформенных графических интерфейсов

**Цель работы.** Изучение возможностей по созданию графических интерфейсов на языке Java, знакомство с типовыми компонентами для построения графических интерфейсов.

### Теоретическая часть

Средство GUI Builder в среде IDE позволяет создавать профессиональные графические интерфейсы без наличия знаний о работе с диспетчерами компоновки. Проектирование форм можно выполнять путем простого размещения компонентов формы в требуемых позициях.

Подробные сведения о визуальных средствах поддержки GUI Builder приведены в разделе Визуальные средства поддержки в GUI Builder.

**Создание проекта**

Поскольку разработка Java в среде IDE всегда выполняется в рамках проектов, сначала необходимо создать проект ContactEditor, в котором будут сохраняться исходные файлы и другие файлы проекта. Проект среды IDE представляет собой группу исходных файлов Java и связанные с ними метаданные, включая файлы свойств проекта, сценарий сборки Ant, управляющий параметрами сборки и выполнения, а также файл project.xml, отображающий целевые элементы Ant для команд среды IDE. Несмотря на то, что приложения на Java часто составляются из нескольких проектов среды IDE, в учебных целях предлагается собрать простое приложение, размещаемое в одном проекте.

Для создания проекта приложения ContactEditor выполните действия, описанные ниже:

1. Выберите команду **Файл** > **Новый проект**. Также можно щелкнуть значок **Новый проект** на панели инструментов среды IDE.

2. На панели **Категории** выберите узел Java и на панели **Проекты** выберите приложение Java. Нажмите кнопку **Далее**.

3. Введите ContactEditor в поле **Имя проекта** и укажите местоположение проекта.

4. Не устанавливайте флажок **Использовать отдельную папку для хранения библиотек**.

5. Убедитесь, что флажок **Установить как главный проект** выбран и очистите поле **Создать главный класс**.

6. Нажмите кнопку **Завершить**.

Среда IDE создаст в системе папку ContactEditor по указанному пути. Эта папка содержит все файлы, связанные с проектом, включая сценарий Ant, папки для хранения исходных файлов и тестов, а также папку с метаданными проекта. Для просмотра структуры проекта используйте окно **Файлы** в среде IDE.

**Создание контейнера JFrame**

По завершении создания приложения можно заметить, что папка с исходными файлами в окне **Проекты** содержит пустой узел <default package>. Для продолжения процесса создания интерфейса необходимо создать контейнер Java, в который будут помещены другие требуемые элементы графического интерфейса. В этом действии будет выполнено создание контейнера с использованием компонента JFrame и размещение контейнера в новом пакете.

Добавление контейнера JFrame:

1. В окне **Проекты** щелкните правой кнопкой мыши узел ContactEditor и выберите **Создать** > **Форма JFrame**. Также форму JFrame можно найти, выбрав **Создать** > **Другое** > **Формы графического интерфейса Swing** > **Форма JFrame**.
2. Введите ContactEditorUI в поле имени класса.
3. Введите my.numberaddition в поле пакета.
4. Нажмите кнопку **Завершить**.

Среда IDE создаст форму ContactEditorUI и класс ContactEditorUI в рамках приложения ContactEditorUI.java и откроет форму ContactEditorUI в средстве GUI Builder. Обратите внимание, что пакет my.contacteditor сохраняется вместо пакета по умолчанию.

**Знакомство со средством GUI Builder**

Теперь, по завершении настройки нового проекта для приложения, необходимо ознакомиться с интерфейсом GUI Builder (рисунок 52).

При добавлении контейнера JFrame на вкладке редактора IDE открывается новая форма ContactEditorUI, чья панель инструментов содержит кнопки, как показано на рисунке 52. Форма ContactEditor, открытая в режиме проектирования GUI Builder, и три дополнительных окна, автоматически выводимые по краям экрана среды IDE, позволяют пользователю выполнять навигацию, а также упорядочивать и редактировать формы графического интерфейса при их построении.

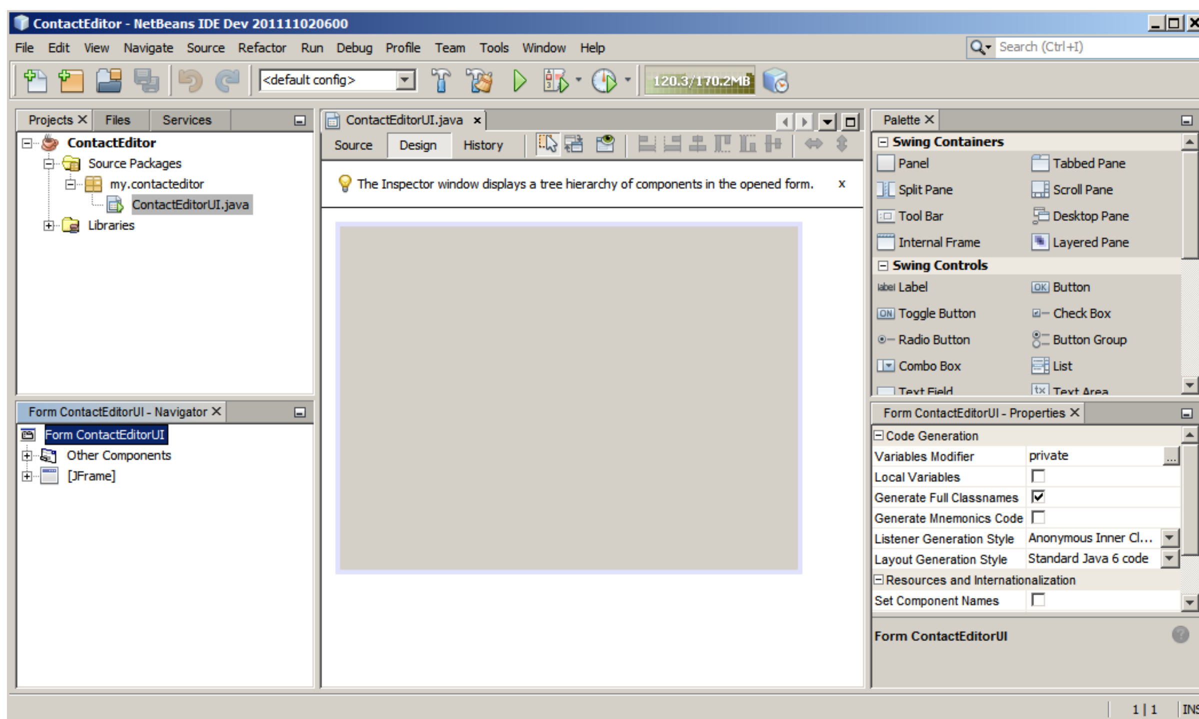


**Рисунок 52. – Интерфейс GUI**

Различные окна GUI Builder содержат следующие элементы:

1. **Область конструирования.** Основное окно конструктора графического интерфейса пользователя для создания и редактирования форм графического интерфейса пользователя Java. Кнопка **Исходный код** панели инструментов позволяет увидеть исходный код класса, кнопка **Проектирование** демонстрирует графическое изображение компонентов GUI, кнопка **Журнал** предоставляет доступ к истории локальных изменений файла. Дополнительные кнопки панели инструментов предоставляют быстрый доступ к часто используемым командам, например, переключение между режимами выбора и подключения, выравнивание компонентов, установка автоматического изменения размера для компонентов и предварительный просмотр форм.

2. **Навигатор.** Обеспечивает представление всех компонентов, как визуальное, так и не визуальное, в приложении в виде дерева иерархии. Кроме того, окно **Навигатор** предоставляет информацию визуальных средств поддержки о том, какие деревья в текущий момент изменяются средством GUI Builder, и позволяет группировать компоненты на доступных панелях.

3. **Палитра.** Настраиваемый список доступных компонентов с вкладками для компонентов JFC/Swing, AWT и JavaBeans, а также диспетчеров компоновки. Кроме того, существует возможность создания, удаления и изменения порядка следования категорий, отображаемых в окне **Палитра**, при помощи средства настройки.

4. **Окно Свойства.** Отображение свойств компонента, выбранного в конструкторе графического интерфейса пользователя, окне навигатора, окне **Проекты** или окне **Файлы**.

При нажатии кнопки **Исходный код** среда IDE отображает в редакторе исходный код приложения на языке Java. Область кода, которая была автоматически создана средством GUI Builder, будет выделена серыми областями (если их выбрать, их цвет поменяется на синий), которые называются **Защитными блоками**. Код в защищенных областях невозможно изменить в представлении **Source**. Функция редактирования доступна в этом представлении только для кода на белом фоне окна редактора. При необходимости изменения кода в защищенном блоке щелкните кнопку **Design** для возврата в окно GUI Builder редактора среды IDE, предоставляющее возможность изменения формы. При сохранении изменений среда IDE обновляет исходный код файла.

*Примечание.* Для опытных разработчиков доступен диспетчер палитры, что позволяет добавлять нестандартные компоненты из файлов JAR, библиотек или других проектов к палитре. Для добавления пользовательских компонентов при помощи диспетчера палитры выберите **Сервис** > **Палитра** > **Компоненты Swing/AWT**.

**Ключевые понятия**

GUI Builder в среде IDE разрешает основные проблемы, возникающие при создании графического интерфейса Java, путем рационализации процесса создания графических интерфейсов, освобождая разработчиков от необходимости изучения особенностей диспетчеров компоновки Swing. Это выполняется путем расширения возможностей конструктора графического интерфейса пользователя IDE NetBeans для поддержки простой парадигмы **Произвольная структура** с простыми правилами компоновки, понятными и простыми в использовании. В процессе проектирования формы GUI Builder предоставляет визуальные средства поддержки, предлагая оптимальное расположение и выравнивание компонентов. GUI Builder способствует переносу пользовательских решений по разработке в функциональный пользовательский интерфейс, реализуемый при помощи диспетчера компоновки GroupLayout и других средств Swing. Благодаря динамической модели размещения компонентов, поведение графического интерфейса в GUI Builder во время выполнения соответствует ожидаемому, что позволяет вносить корректировки без изменения установленных взаимосвязей между компонентами.

При каждом изменении размеров форм, переключении локалей или применении нового общего стиля графический интерфейс автоматически изменяется в соответствии с новой настройкой вставок и смещений стиля.

**Свободное проектирование**

В GUI Builder среды IDE можно создавать формы путем простого размещения компонентов в требуемых позициях, как при использовании абсолютного позиционирования. GUI Builder автоматически определяет необходимые атрибуты и создает код. Отсутствует необходимость в настройке вставок, привязок, заполнителей и др. вручную.

**Автоматическое размещение компонентов (привязка)**

В процессе добавления компонентов в форму GUI Builder предоставляет визуальные средства поддержки, позволяющие размещать компоненты на основе общего стиля операционной системы. GUI Builder содержит встроенные подсказки и другие визуальные средства поддержки относительно требуемого расположения компонентов в форме, позволяющие выполнять автоматическую привязку компонентов к различным позициям направляющих. Подсказки выводятся на основе позиции компонента в форме, при этом обеспечивается гибкость при выравнивании компонентов, и соответствующий новый общий стиль отображается во время выполнения.

**Визуальные средства поддержки в GUI Builder**

Средство GUI Builder предоставляет визуальные средства поддержки для обеспечения привязки компонентов и установки отношений между ними. Эти индикаторы способствуют быстрому определению различных отношений при позиционировании и поведения при привязке компонентов, влияющих на внешний вид и работу графического интерфейса. Это ускоряет процесс проектирования графического интерфейса и позволяет быстро создавать профессиональные функционирующие визуальные интерфейсы.

**Добавление компонентов. Основы**

Несмотря на то, что GUI Builder в среде IDE упрощает процесс создания графического интерфейса Java, часто важно схематически изобразить требуемое расположение элементов интерфейса до их размещения в форме. Многие разработчики интерфейсов считают этот метод наиболее эффективным, однако в учебных целях рекомендуется просмотреть результат построения формы, перейдя к разделу Предварительный просмотр графического интерфейса.

После добавления компонента JFrame как контейнера формы верхнего уровня следует добавить несколько панелей JPanel, которые позволят объединить компоненты пользовательского интерфейса в кластеры с использованием границ с заголовками.

**Добавление панели JPanel**

Панель JPanel можно добавить следующим образом:

1. В окне **Палитра** выберите компонент **Панель** из категории **Контейнеры Swing**, нажав и отпустив кнопку мыши.

2. Переместите курсор в левый верхний угол формы GUI Builder. При расположении компонента рядом с верхней или левой границами контейнера появляются горизонтальные и вертикальные направляющие, обозначающие предпочтительные поля. Щелкните в пространстве формы для расположения панели JPanel в позиции курсора мыши. Компонент JPanel, который появляется в форме ContactEditorUI, подсвечен оранжевым, чтобы показать, что он выбран. После того, как кнопка мыши будет отпущена, появятся маленькие индикаторы, которые показывают привязки компонентов. А соответствующий узел JPanel отобразит окно **Навигатор**, как изображено на рисунке 53.

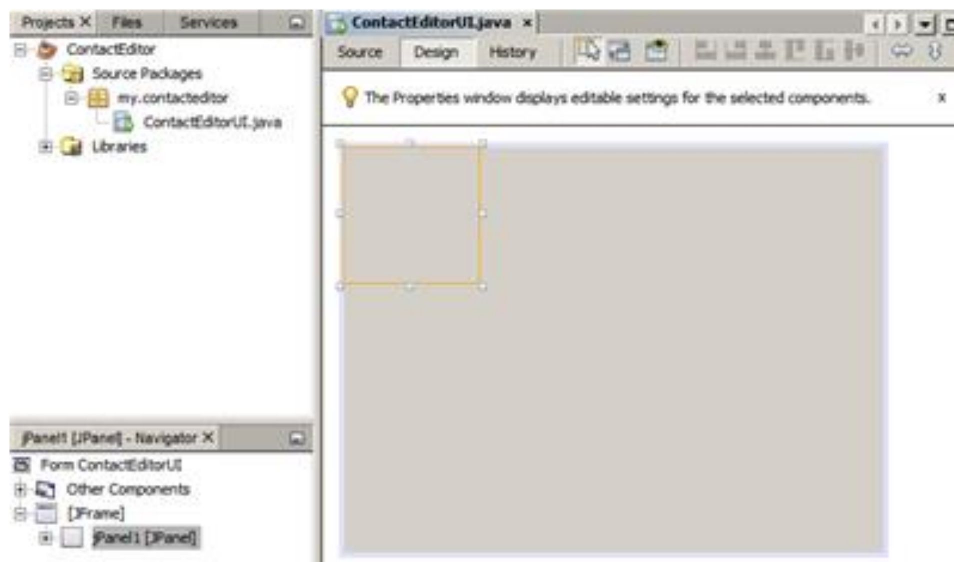


**Рисунок 53. – Компонент JPanel**

Теперь следует изменить размеры панели JPanel, чтобы подготовить пространство для размещения других компонентов. Однако сначала обратим внимание на еще одну функцию визуализации GUI Builder. Отмените выбор добавленной панели JPanel. Поскольку граница с заголовком еще не добавлена, панель исчезнет. Однако обратите внимание на то, что при

наведении курсора на панель JPanel ее края становятся светло-серыми, что позволяет четко определить позицию этого компонента. Теперь щелкните в пределах границ панели для ее повторного выбора, в результате появятся метки-манипуляторы и индикаторы привязки.

**Изменение размера панели JPanel**

Изменить размер панели JPanel можно следующим образом:

1. Выберите только что добавленную панель JPanel. По периметру компонента появятся небольшие квадратные метки-манипуляторы.

2. Щелкните метку-манипулятор на правой границе панели JPanel и, не отпуская кнопки мыши, перемещайте метку до тех пор, пока рядом с границей не появится пунктирная направляющая.

3. Отпустите кнопку мыши для фиксации измененного размера компонента.

4. Теперь компонент JPanel расширен и охватывает пространство между левым и правым полем контейнера с учетом рекомендуемого смещения, как изображено на рисунке 54.

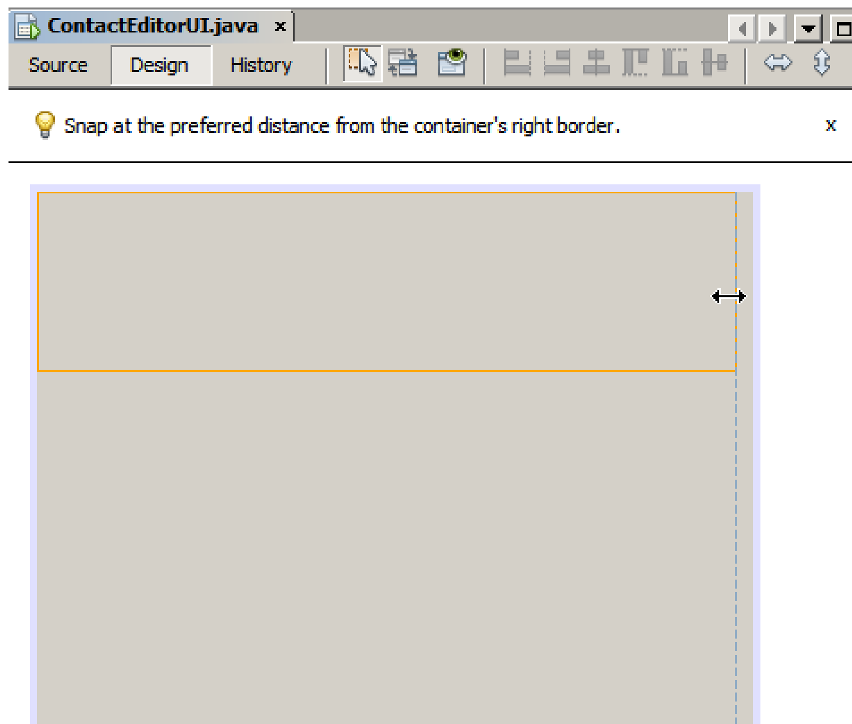


**Рисунок 54. – Установка компонента**

Теперь, после добавления панели, содержащей сведения об имени пользовательского интерфейса, необходимо повторить этот процесс для добавления еще одного компонента с данными об адресе электронной почты

непосредственно под первым компонентом. Повторите две предыдущие процедуры, как изображено на рисунках 55 и 56, при этом обратите внимание на предлагаемое размещение компонентов в GUI Builder. Следует отметить, что предложенный вертикальный интервал между двумя панелями JPanel намного меньше, чем пространство до границ формы. После добавления второй панели JPanel следует изменить ее размеры так, чтобы она заполнила оставшееся пространство формы (по вертикали) (см. рисунки 55, 56).

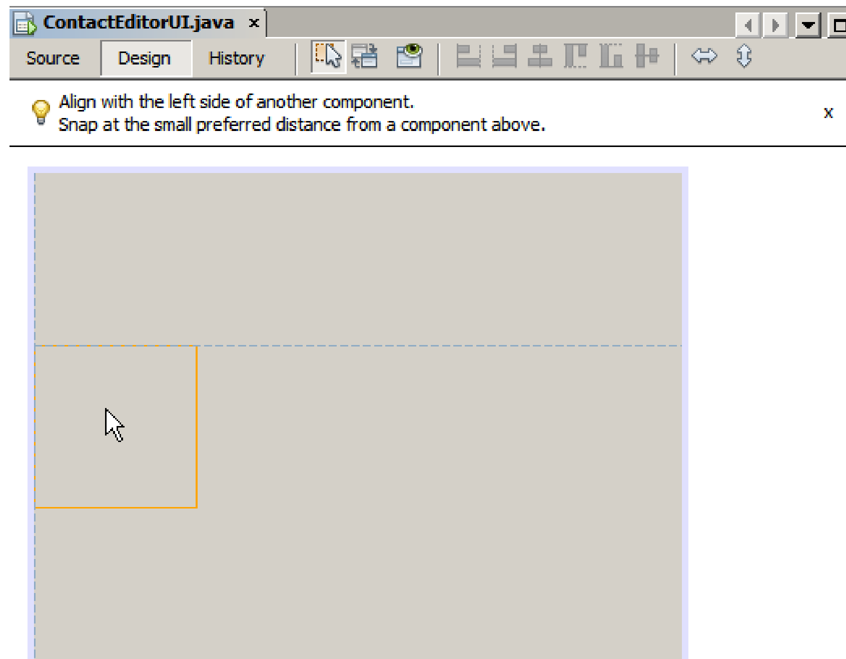


**Рисунок 55. – Выравнивание по левой стороне**

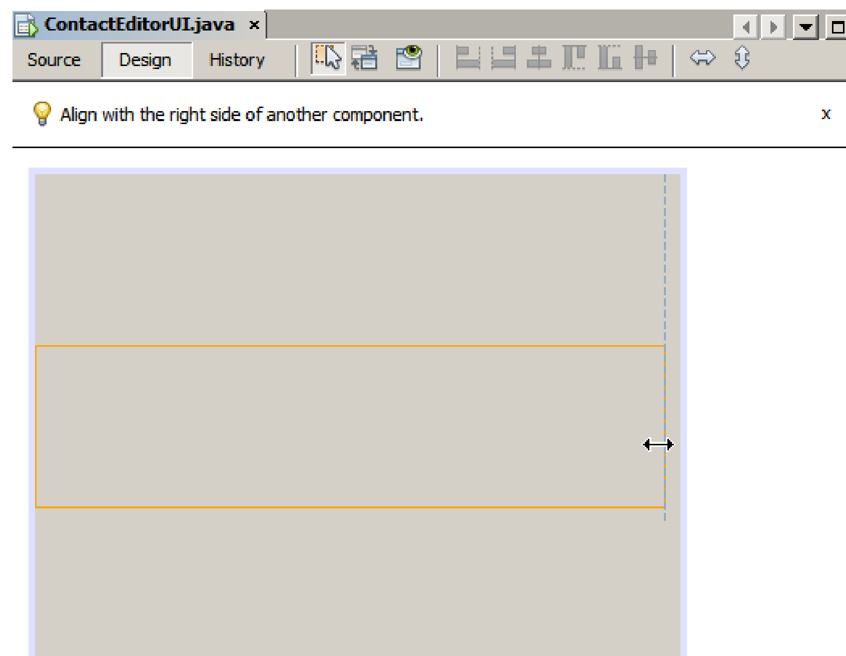


**Рисунок 56. – Выравнивание по вертикали**

Поскольку функции в верхних и нижних разделах графического интерфейса должны быть визуально различимы, в каждую панель JPanel необходимо добавить границу и заголовок. Это действие можно выполнить при помощи окна **Свойства** или с использованием всплывающего меню.

Для добавления границ с заголовком в панель JPanels выполните действия, указанные ниже:

1. Выберите верхнюю панель JPanel в GUI Builder.
2. В окне **Свойства** нажмите кнопку с многоточием **(...)** рядом со свойством **Граница**.
3. В появившемся редакторе границ JPanel выберите узел **Граница с заголовком** на панели **Доступные границы**.
4. На панели **Свойства**, расположенной ниже, введите Name как значение свойства **Заголовок**.
5. Нажмите кнопку с многоточием **(...)** рядом со свойством **Шрифт**, выберите **Жирный** в поле **Стиль шрифта** и введите **12** в поле **Размер**. Нажмите кнопку **ОК** для выхода из диалоговых окон.
6. Выберите нижнюю панель **JPanel** и повторите действия 2–5, но на этот раз щелкните правой кнопкой мыши панель **JPanel** и откройте окно **Properties** из контекстного меню. Введите значение **E-mail** в поле свойства **Заголовок**.

К обоим компонентами JPanel будут добавлены границы с заголовками (рисунок 57).

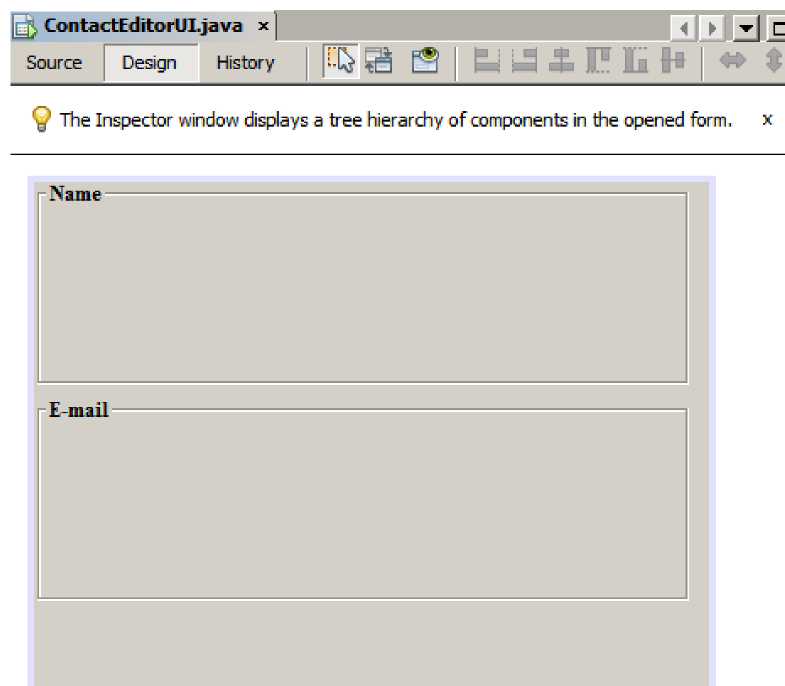


**Рисунок 57. – Добавление границ с заголовком**

**Добавление отдельных компонентов к форме**

Теперь добавим компоненты, которые будут представлять фактическую контактную информацию списка контактов. Для этого необходимо добавить четыре текстовых поля **JTextField**, в которых будет содержаться контактная информация, и четыре компонента **JLabel** для описания содержимого этих полей. При выполнении этого действия обратите внимание на горизонтальные и вертикальные направляющие, выводимые GUI Builder, которые отображают предпочтительное расстояние между компонентами согласно общему стилю операционной системы. Таким образом обеспечивается соответствие между создаваемым графическим интерфейсом и общим стилем операционной системы во время выполнения.

**Добавление компонента JLabel в форму**

Добавить компонент JLabel в форму можно по алгоритму:

1. В окне **Палитра** выберите компонент **Label** (Метка) из категории **Элементы управления Swing**.

2. Переместите курсор на панель **JPanel Name**, созданную ранее. После появления направляющих, указывающих на размещение компонента **JLabel** в верхнем левом углу панели **JPanel** с небольшим полем у верхней и левой границ, щелкните кнопкой мыши для подтверждения расположения нового компонента. К форме будет добавлен компонент **JLabel**, а в окне **Инспектор** появится соответствующий узел.

Перед переходом к следующему действию необходимо отредактировать отображаемый текст в только что добавленном компоненте **JLabe**l. Несмотря на то, что этот текст можно изменить в любое время, проще всего это сделать при добавлении компонента.

**Редактирование отображаемого текста компонента JLabel:**

1. Дважды щелкните компонент **JLabel** для выбора отображаемого текста.

2. Введите **First Name:** и нажмите **ENTER**.

Будет выведено новое имя **JLabel**, и ширина компонента будет изменена в соответствии с новым текстом.

Теперь следует добавить текстовое поле **JTextField**, на примере которого можно рассмотреть функцию выравнивания по базовой линии в GUI Builder.

**Добавление компонента JLabel в форму:**

1. В окне **Палитра** выберите компонент **Text field** (текстовое поле) из категории **Элементы управления Swing**.

2. Переместите курсор непосредственно к правому краю только что добавленного компонента **JLabel First Name:**. При появлении горизонтальной направляющей, указывающей на выравнивание базовой линии поля **JTextField** по базовой линии компонента **JLabel**, и вертикальной направляющей, определяющей интервал между этими двумя компонентами, щелкните кнопкой мыши для подтверждения позиции **JTextField**.

Компонент **JTextField** размещается в форме в позиции, выровненной по базовой линии **JLabel**, как изображено на следующем рисунке. Обратите внимание на то, что компонент **JLabel** был немного смещен вниз с целью его выравнивания по базовой линии текстового поля, расположенной чуть выше. Узел, который представляет компонент, добавлен в окно **Навигатор**, как обычно (рисунок 58).

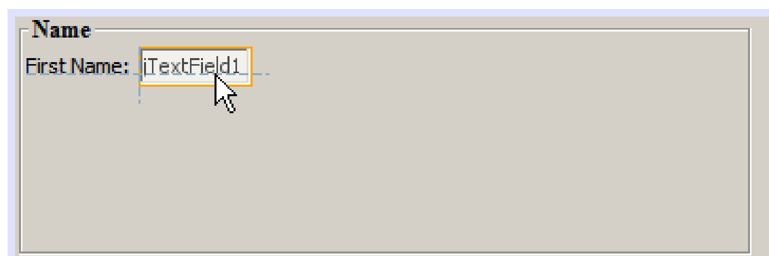


**Рисунок 58. – JTextField компонент**

Прежде чем продолжить, необходимо немедленно добавить дополнительный компонент **JLabel** и **JTextField** справа от уже добавленных компонентов, как изображено на рисунке ниже. Введите **Last Name:** в качестве отображаемого текста в компоненте **JLabel**, но пока не изменяйте текст заполнителя поля **JTextFields** (рисунок 59).

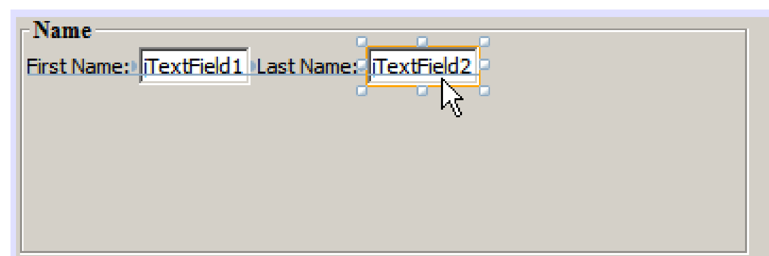


**Рисунок 59. – Установка нескольких компонентов**

**Изменение размера компонента JTextField:**

1. Выберите только что добавленный компонент **JTextField** справа от компонента **JLabel Last Name:**.

2. Перетащите метку-манипулятор правого края компонента **JTextField** к правой границе панели **JPanel**.

3. При появлении вертикальных направляющих, отображающих расстояние между текстовым полем и правым краем панели **JPanel**, отпустите кнопку мыши для фиксации изменения размеров поля **JTextField**.

Правая граница компонента **JTextField** будет выровнена по предложенной границе поля панели **JPanel**, как показано на рисунке 60.

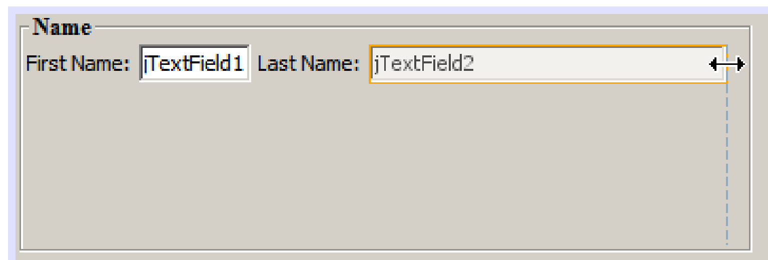


**Рисунок 60. – Выравнивание компонентов**

**Добавление нескольких компонентов в форму**

Теперь добавьте в форму компоненты **JLabel Title:** и **Nickname:**, описывающие два текстовых поля **JTextField**, которые также будут добавлены. Перетащите компоненты, удерживая нажатой клавишу SHIFT, чтобы быстрее добавить их на форму. При этом в GUI Builder снова появятся соответствующие горизонтальные и вертикальные направляющие, указывающие на предпочтительное размещение компонента.

Для добавления нескольких компонентов **JLabel** в форму выполните действия, указанные ниже:

1. В окне **Палитра** выберите компонент **Label** (метка) из категории **Элементы управления Swing**, нажав и отпустив кнопку мыши.

2. Переместите курсор в форме непосредственно под ранее добавленным компонентом **JLabel First Name:**. При появлении направляющих, указывающих на выравнивание левой границы нового компонента **JLabel** по границе компонента **JLabel**, расположенного выше, и при наличии небольшого пространства между этими компонентами, щелкните кнопкой мыши при нажатой клавише SHIFT для фиксации расположения первого компонента **JLabel**.

3. Не отпуская клавишу SHIFT, поместите другой компонент **JLabel** непосредственно справа от первого. Перед размещением второго компонента **JLabel** отпустите клавишу SHIFT. В случае удерживания клавиши SHIFT во время размещения второго компонента можно нажать клавишу ESC для отмены действия.

Компоненты **JLabel** будут добавлены к форме и образуют второй ряд, как показано на рисунке ниже. Узлы, представляющие каждый компонент, добавлены в окно Навигатор (рисунок 61).

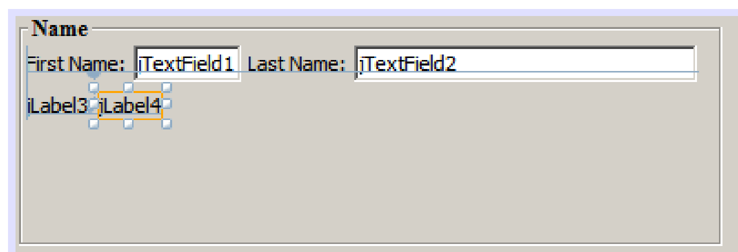


**Рисунок 61. – Установка компонентов в несколько рядов**

Перед следующим действием необходимо изменить имя компонента **JLabel**, что позволит проверить результаты выравнивания, которое будет произведено чуть позже.

Для редактирования отображаемого текста компонента **JLabel** выполните действия, указанные ниже:

1. Дважды щелкните компонент **JLabel** для выбора отображаемого текста.

2. Введите **Title:** и нажмите ENTER.

3. Повторите шаги 1 и 2, введя **Nickname:** в качестве имени второго свойства **Name** компонента **JLabel**.

Новые имена компонентов **JLabel** будут выведены в форме и смещены в результате изменения ширины текста, как изображено на рисунке 62.

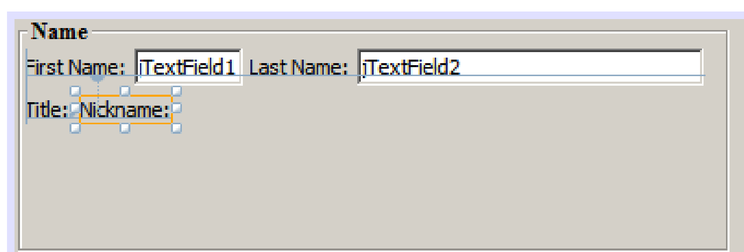


**Рисунок 62. – Задание имен компонентам**

**Пример создания простейшего калькулятора**

Для начала создается проект. Далее необходимо создать контейнер Java, в который будут помещены другие требуемые элементы графического интерфейса. В этом действии контейнер будет создан с помощью элемента **JFrame**. Контейнер будет помещен в новый пакет, который будет отображаться в узле **Source Packages**.

**Создание контейнера JFrame**

Для создания контейнера JFrame выполните следующие действия:

1. В окне **Проекты** щелкните правой кнопкой мыши узел **NumberAddition** и выберите **Создать** > **Другие**.

2. В диалоговом окне создания файла выберите категорию **Swing GUI Forms** и тип файла **JFrame Form**. Нажмите кнопку **Далее**.

3. Введите **NumberAdditionUI** в качестве имени класса.

4. Выберите пакет **my.numberaddition**.

5. Нажмите кнопку **Готово**.

Среда IDE создает форму NumberAdditionUI и класс NumberAdditionUI в приложении NumberAddition и открывает форму NumberAdditionUI в GUI Builder. Пакет my.numberaddition заменяет собой пакет по умолчанию.

### Добавление элементов: создание внешнего интерфейса

Далее с помощью окна **Palette** внешний интерфейс приложения заполняется панелью **JPanel**. После этого добавляются три элемента **JLabel** (текстовые подписи), три элемента **JTextField** (текстовые поля) и три элемента **JButton** (кнопки).

После перетаскивания и размещения указанных выше элементов элемент **JFrame** должен выглядеть так, как показано на рисунке 63.

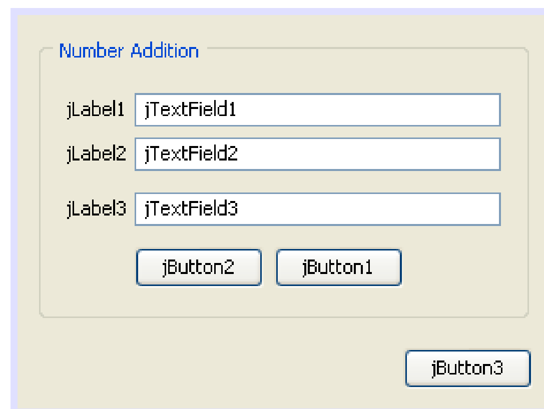


**Рисунок 63. – Предварительный результат размещения компонентов**

Если в правом верхнем углу среды IDE отсутствует окно **Palette** (Палитра), выберите **Window** (Окно) > **Palette** (Палитра). Выполните действия:

1. Для начала выберите панель из категории **Swing Containers** (Контейнеры Swing) в палитре и перетащите ее на **JFrame**.

2. Панель **JPanel** будет выделена. Перейдите к окну **Properties** и нажмите кнопку с многоточием **(...)** рядом с полем **Border** для выбора стиля границы.

3. В диалоговом окне **Border** выберите **TitledBorder** из списка и введите **Number Addition** в поле **Title**. Для сохранения изменений и закрытия диалогового окна нажмите кнопку **OK**.

4. Теперь на экране должен отображаться пустой элемент **JFrame** с заголовком Number Addition, как показано на рисунке 63. Согласно рисунку, добавьте к нему три метки **JLabel**, три текстовых поля **JTextField** и три кнопки **JButton**.

**Переименование элементов**

На этом этапе будет выполнено переименование элементов, которые были добавлены к элементу **JFrame**. Для этого:

1. Дважды щелкните **jLabel1** и измените **ntrcn** (свойство **text**) на **First Number**.

2. Дважды щелкните **jLabel2** и измените текст на **Second Number**.

3. Дважды щелкните **jLabel3** и измените текст на **Result**.

4. Удалите стандартный текст из **jTextField1**. Отображаемый текст можно преобразовать в редактируемый. Для этого щелкните правой кнопкой мыши текстовое поле и выберите «Редактировать текст» во всплывающем меню. При этом может потребоваться восстановить первоначальный размер поля **jTextField1**. Повторите это действие для полей **jTextField2** и **jTextField3**.

5. Измените отображаемый текст **jButton1** на **Clear**. Для изменения текста кнопки щелкните кнопку правой кнопкой мыши и выберите **Edit Text**. В качестве альтернативы можно щелкнуть кнопку, выдержать паузу и щелкнуть еще раз.

6. Измените отображаемый текст **jButton2** на **Add**.

7. Измените отображаемый текст **jButton3** на **Exit**.

Теперь готовый графический интерфейс должен выглядеть так, как показано на рисунке 64.

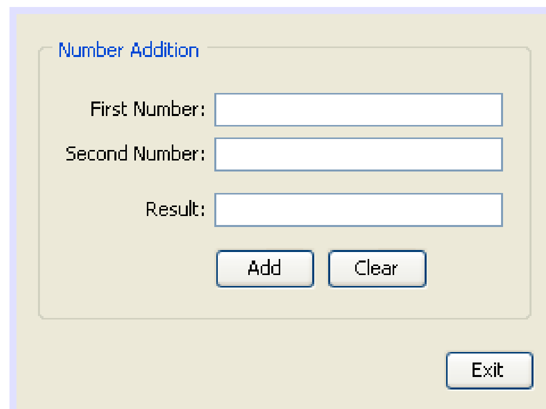


**Рисунок 64. – Окончательный вариант созданного интерфейса**

**Добавление функциональности**

В этом упражнении будет добавлена необходимая функциональность к кнопкам **Add**, **Clear** и **Exit**. Поля jTextField1 и TextField2 будут использоваться для ввода значений пользователем, а jTextField3 – для вывода результата работы программы. Создаваемая программа представляет собой простейший калькулятор. Итак, приступим.

**Добавление функциональности к кнопке Exit**

Для того чтобы кнопки стали функциональными, каждой из них необходимо присвоить обработчик событий, который будет отвечать за реагирование на события. В нашем случае требуется идентифицировать событие нажатия кнопки – путем щелчка мышью или с помощью клавиатуры. Поэтому будет использоваться интерфейс **ActionListener**, предназначенный для обработки событий **ActionEvent**.

Выполните следующие действия:

1. Щелкните правой кнопкой мыши кнопку **Exit**. Во всплывающем меню выберите **Events** (События) > **Action** (Действие) > **actionPerformed**. Учтите, что меню содержит множество других событий, на которые может реагировать программа. При выборе события actionPerformed среда IDE автоматически добавит прослушиватель ActionListener к кнопке Exit (Выход) и создаст метод обработчика для обработки метода прослушивателя **actionPerformed**.

2. В среде IDE автоматически открывается окно **Source Code**, где отображается место вставки действия, которое должно выполняться кнопкой при ее нажатии (с помощью мыши или клавиатуры). Окно Source Code должно содержать следующие строки:

```
private void jButton3ActionPerformed(java.awt.event.ActionEvent evt) {
   //TODO add your handling code here:
         }
```

3. Теперь добавим код действия, которое должна выполнять кнопка Exit. Замените строку TODO на **System.exit(0);**. Готовый код кнопки Exit должен выглядеть следующим образом:

```
private void jButton3ActionPerformed(java.awt.event.ActionEvent evt) {
   System.exit(0);
         }
```

**Добавление функциональности к кнопке Clear**

Выполните следующие действия:

1. Щелкните вкладку **Design** в верхней части рабочей области для возврата к экрану **Form Design**.

2. Щелкните правой кнопкой мыши кнопку Clear (jButton1). В появившемся меню выберите **Events** > **Action** > **actionPerformed**.

3. Нажатие кнопки Clear должно приводить к удалению всего текста из всех текстовых полей jTextField. Для этого следует добавить код, аналогич-

ный приведенному выше. Готовый исходный код должен выглядеть следующим образом:

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt){
 jTextField1.setText("");
      jTextField2.setText("");
      jTextField3.setText("");
 }
```

Этот код удаляет текст из всех трех полей JTextField, оставляя их пустыми.

**Добавление функциональности к кнопке Add**

Кнопка Add должна выполнять три действия:

1. Сначала она принимает данные, введенные пользователем в полях jTextField1 и jTextField2, и преобразовывает их из типа String в тип Float.
2. Дальше она выполняет сложение двух чисел.
3. И затем она преобразует сумму в тип String и поместит ее в jTextField3.

Далее:

1. Щелкните вкладку **Design** в верхней части рабочей области для возврата к экрану **Form Design**.
2. Щелкните правой кнопкой мыши кнопку Add (jButton2). Во всплывающем меню выберите **Events** > **Action** > actionPerformed.
3. Добавьте код действий, которые должна выполнять кнопка **Add**. Готовый исходный код должен выглядеть следующим образом:

```
private void jButton2ActionPerformed(java.awt.event.ActionEvent evt){
    // First we define float variables.
    float num1, num2, result;
    // We have to parse the text to a type float.
    num1 = Float.parseFloat(jTextField1.getText());
    num2 = Float.parseFloat(jTextField2.getText());
   // Now we can perform the addition.
    result = num1+num2;
    // We will now pass the value of result to jTextField3.
    // At the same time, we are going to
    // change the value of result from a float to a string.
    jTextField3.setText(String.valueOf(result));
}
```

Теперь программа полностью готова, и можно приступить к ее сборке и выполнению.

## ИНДИВИДУАЛЬНЫЕ ВАРИАНТЫ ЗАДАНИЙ

1. Конвертер валют. Пользователь вводит сумму, выбирает две валюты и получает эквивалент во второй валюте.

2. Бег. Пользователь указывает количество км за каждый день в течение некоторого периода времени. Программа находит общий пробег, среднее значение в день, наименьшее и наибольшее значения.

3. Средняя температура. Пользователь вводит данные о температуре за некоторый период времени. Программа находит среднее значение.

4. Создать калькулятор, аналогичный калькулятору Windows (вид: а) обычный; б) инженерный).

5. Создать приложение «Ежедневник». Возможности: создание записей двух видов (задача и заметка). Для задачи указывается срок выполнения. Заметка относится к одной из категорий. Список категорий можно изменять. Для заметки указываются метки, характеризующие тему заметки. Добавить поиск по меткам и содержанию заметок/задач в рамках выбранных категорий.

6. Создать приложение «Каталог книг». Пользователь добавляет книгу в каталог. Сам файл копируется из указанной директории в папку каталога. При добавлении указывается категория, статус (прочитана, прочитать, в процессе чтения (номер страницы)), рейтинг (если прочитана).

## ЗАДАНИЕ К ЛАБОРАТОРНОЙ РАБОТЕ

1. Ознакомиться с теоретической частью.

2. Реализовать приложение с графическим интерфейсом в соответствии с индивидуальным вариантом.

## КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Какие функции выполняет библиотека Swing?
2. Для чего используется контейнер JFrame?
3. Для чего используется компонент JPanel?
4. Назначение элементов JTextField и JLabel?
5. Назначение интерфейса ActionListener?

## СОДЕРЖАНИЕ ОТЧЕТА

1. Ф.И.О., группа, название лабораторной работы.
2. Цель работы.
3. Описание проделанной работы.
4. Результаты выполнения лабораторной работы.
5. Выводы.

Отчет и исходные коды запаковать в архив с названием по следующему шаблону:

Lab1–«группа, аббревиатура на латинице»–«Фамилия на латинице».

Пример: Lab8–13VS–Ivanov.zip.

### Порядок защиты лабораторной работы

1. Предоставление отчета о выполнении лабораторной работы.
2. Демонстрация программы.
3. Выполнение дополнительного задания.
4. Ответы на вопросы по теме текущей лабораторной работы.

## ЛИТЕРАТУРА

1. Объектно-ориентированный анализ и проектирование с примерами приложений / Г. Буч [и др.]. – 3-е изд. – М. : Вильямс, 2020. – 720 с. – (Серия «Объектные технологии»).

2. Эккель, Б. Философия Java / Б. Эккель. – 4-е изд. – СПб. : Питер 2009. – 1168 с. – (Серия «Классика computer science»).

3. Шилдт, Г. Полный справочник по Java. Java SE 6 Edition / Г. Шилдт. – 7-е изд. – М. : Вильямс, 2009. – 1040 с.

4. Морган, М. Java 2. Руководство разработчика : учеб. пособие / М. Морган. – М. : Вильямс, 2000. – 720 с.

5. Ноутон, П. Java 2 в подлиннике / П. Ноутон, Г. Шилдт. – СПб. : БХВ-Петербург, 2000. – 1072 с.

6. Эккель, Б. Философия Java / Б. Эккель. – СПб. : Питер, 2001. – 880 с.

7. Смирнов, Н. И. Java 2 : учеб. пособие / Н. И. Смирнов. – М. : Три Л, 2000. – 320 с.

8. Разработка Swing GUI в IDE NetBeans [Электронный ресурс]. – Режим доступа: https://netbeans.apache.org/kb/docs/java/quickstart-gui_ru.html. – Дата доступа: 01.12.2021.

9. Введение в разработку графического интерфейса [Электронный ресурс]. – Режим доступа: https://netbeans.apache.org/kb/docs/java/gui-functionality_ru.html. – Дата доступа: 01.12.2021.