Министерство образования и науки Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего профессионального образования
«Севастопольский государственный университет»

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА ЯЗЫКЕ JAVA
В СРЕДЕ ECLIPSE

Методические указания
к выполнению лабораторной работы
для студентов, обучающихся по направлению
09.03.02 “Информационные системы и технологии”
очной и заочной форм обучения

Севастополь
2015

УДК 004.42 (075.8)

Основы программирования на языке Java в среде Eclipse: методические указания к лабораторной работе №1 по дисциплине “Платформа Java” для студентов направления 09.03.02 “Информационные системы и технологии”/ Сост. **С.А. Кузнецов, А.Л. Овчинников —** Севастополь: Изд-во СевГУ, 2015. — 26 с.

Цель указаний: оказание помощи студентам направления 09.03.02 “Информационные системы и технологии” при выполнении лабораторной работы №1 по дисциплине “Платформа Java”.

Методические указания составлены в соответствии с требованиями программы дисциплины «Платформа Java» для студентов дневной и заочной формы обучения направления 09.03.02 “Информационные системы и технологии” и утверждены на заседании кафедры «Информационные системы» протоколом № 1 от 31 августа 2015 года.

Допущено учебно-методическим центром СевГУ в качестве методических указаний.

Рецензент: Кожаев Е.А., канд. техн. наук, доцент кафедры кибернетики и вычислительной техники.

СОДЕРЖАНИЕ

1. ЦЕЛЬ РАБОТЫ

В ходе выполнения данной лабораторной работы необходимо ознакомиться с функциональными возможностями среды разработки Eclipse, основами языка Java, приобрести практические навыки создания консольных приложений на языке Java, с возможностью доступа к файлам.

2. ПОСТАНОВКА ЗАДАЧИ

2.1. В соответствии с вариантом задания, представленным в п. 4 необходимо разработать программу на языке Java, выполняющую требуемые действия (см. таблицу 4.1). Предусмотреть ввод входных данных с клавиатуры (по умолчанию) или из файла (при запуске с параметром *–i <filename>).* Предусмотреть вывод результатов на экран (по умолчанию) или в файл (при запуске с параметром *–o <filename>*). Предусмотреть возможность запуска с 2 параметрами: *–i <filename1> –o <filename2>*.Предусмотреть обработку ошибок с использованием операторов языка Java: *try, catch* и *finally*.

2.2. Ознакомившись со средствами отладки программ в среде Eclipse выполнить отладку разработанной программы.

2.3. Проверить правильность работы программы на тестовых примерах корректных и не корректных входных данных.

3. ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

3.1. Общие положения

Java — объектно-ориентированный язык программирования, разработанный компанией Sun Microsystems (в последствии приобретённой компанией Oracle).

Программы на Java транслируются в байт-код, выполняемый виртуальной машиной Java (JVM) — программой, обрабатывающей байтовый код и передающей инструкции оборудованию как интерпретатор.

Java Development Kit (сокращенно JDK) — бесплатно распространяемый компанией Oracle Corporation (ранее Sun Microsystems) комплект разработчика приложений на языке Java, включающий в себя компилятор Java (javac), стандартные библиотеки классов Java, примеры, документацию, различные утилиты и исполнительную систему Java (JRE). В состав JDK не входит интегрированная среда разработки на Java, поэтому разработчик, использующий только JDK, вынужден использовать внешний текстовый редактор и компилировать свои программы, используя утилиты командной строки.

Eclipse – это популярная IDE с открытым исходным кодом для Java-разработки. Она решает основные задачи, такие как компиляция кода и настройка среды отладки, позволяя сосредоточиться на написании и тестировании программ. Кроме того, Eclipse можно использовать для организации файлов исходного кода в проекты, компиляции и тестирования этих проектов и для хранения файлов проектов в любом количестве хранилищ исходного кода.

Чтобы использовать Eclipse для Java-разработки, нужно предварительно установить JDK.

3.2. Настройка среды разработки Java

Установка JDK.

Чтобы загрузить и установить JDK , выполните следующие действия:

1. Зайдите на страницу загрузок программного обеспечения для разработчиков Java компании Oracle (http://www.oracle.com/technetwork/java/javase/downloads/index.html) и нажмите кнопку Java Platform (JDK), чтобы вызвать страницу загрузки последней версии JDK.
2. Нажмите кнопку Download.
3. Выберите нужную платформу операционной системы.
4. Вам будет предложено ввести имя пользователя и пароль. Введите их, если у вас есть учетная запись, зарегистрируйтесь, если ее нет, или нажмите кнопку Continue, чтобы пропустить этот шаг и перейти к загрузке.
5. По запросу сохраните файл на жестком диске.
6. Когда загрузка будет завершена, запустите программу установки. Установите JDK на жесткий диск в соответствии с рекомендациями программы установки.

Теперь на вашем компьютере есть среда Java. Далее необходимо установить Eclipse IDE.

Установка Eclipse IDE.

Чтобы загрузить и установить Eclipse, выполните следующие действия:

1. Откройте страницу загрузок Eclipse (http://www.eclipse.org/downloads/).
2. Выберите Eclipse IDE for Java Developers.
3. В разделе Download Links справа выберите свою платформу.
4. Выберите зеркало для скачивания, а затем сохраните файл на жестком диске.
5. Распакуйте содержимое архива в необходимую папку (например, C:\Program Files\Eclipse).

3.3. Основные понятия о среде Eclipse

Среда разработки Eclipse является модульной средой разработки на основе программного ядра и интерфейсов для написания и подключения дополнительных модулей (плагинов, plugin). Таким образом, среда Eclipse может быть использована не только для написания приложений на языке Java, но также и для других целей, в зависимости от установленных плагинов.

Рассмотрим основные понятия, применяемые в среде Eclipse:

Рабочая область (Workspace) – это фактически директория на диске, в которой хранятся различные файлы настроек, относящиеся к конкретному пользователю среды Eclipse. В любой момент времени пользователь работает в определенной рабочей области.

Проект (Project) – это логическое понятие, представляющее собой определенную единицу работы пользователя, способ группировки исходных текстов программы, набора библиотек, а также настроек компилятора. Для создания программ на языке Java используется проект с типом Java.

Представление (панель View) – это одно из окон в рабочей области, предназначенное для выполнения специфических задач при работе над проектом или проектами. Например, представление Package Explorer используется для просмотра иерархии пакетов в Java проекте, а представление Outline – для просмотра структуры класса, открытого в редакторе. Новые представления могут быть отображены с использованием меню Window->Show View. Как и во многих современных средах разработки представления можно помещать в любом месте оконного интерфейса программы, закреплять их, а также переводить в режим быстрого просмотра (Fast View). Режим быстрого просмотра включается выбором соответствующего пункта контекстного меню, которое появляется при нажатии правой клавиши мыши на заголовке соответствующего представления. Режим быстрого просмотра удобен тогда, когда большая часть времени проводится в работе с редактором исходного кода, а представления нужно изредка ненадолго открывать.

Перспектива (проекция, Perspective) – это способ группировки представлений для выполнения различных задач над одним и тем же проектом или набором проектов. Например, по умолчанию для Java-проектов используется перспектива с одноименным названием, удобная для редактирования исходного кода, а перспектива Debug используется в режиме отладки программ и содержит набор представлений, характерных для данного вида роботы. Пользователь может создавать свои собственные перспективы, а также модифицировать существующие.

Рабочий набор (Working Set) – этот элемент позволяет скрывать неиспользуемые ресурсы от пользователя, когда тот должен работать только над частью открытого проекта или проектов. Использование рабочих наборов позволяет повысить производительность труда за счет сокрытия лишней информации, что позволяет лучше сосредоточиться над решением определенных задач. Создать рабочий набор можно с помощью меню в представлении Package Explorer (иконка справа с изображением стрелки, направленной вниз). С помощью этого меню можно также снять любой рабочий набор, чтобы увидеть все доступные ресурсы.

Добавление программных элементов в проект. Выполняется при помощи контекстного меню, вызываемого в представлении Package Explorer (естественно, это только один из множества способов, но он наиболее распространен).

Компиляция программ. Выполняется по умолчанию автоматически (это поведение можно изменить в меню Build). Таким образом, пользователю нет необходимости заботиться о компиляции приложения перед его запуском.

Запуск программы. Для запуска необходимо создать соответствующую конфигурацию запуска, которая включает в себя как имя класса, содержащего точку входа, так и различные параметры командной строки, а также виртуальной машины. Конфигурацию можно создать вручную с помощью меню Run->Run..., но гораздо

проще создать ее автоматически, при помощи контекстного меню, вызываемого в редакторе (Run As->Java Application).

Внимание! Учтите, что класс должен содержать точку входа:

public static void main(String[] args)

для того, чтобы его можно было запустить. Точка входа генерирует заготовку метода main() для запуска класса как приложения Java.

После создания конфигурации можно запускать приложение с помощью пиктограммы Run в панели инструментов (отладка запускается с помощью пиктограммы Debug, находящейся рядом). Весь вывод приложения будет поступать в представление Console.

Редактирование исходного кода. Eclipse предоставляет богатые средства для редактирования исходного кода, которые включают в себя подсветку синтаксиса, автодополнение кода, автогенерацию кода, а также множество других полезных функций. Автодополнение кода вызывается нажатием сочетания клавиш Ctrl+Пробел и позволяет быстрее писать код, используя только первые буквы набираемого выражения.

Автогенерация кода – это возможность автоматической генерации целых блоков кода из коротких аббревиатур. Для использования необходимо написать соответствующую аббревиатуру и нажать Ctrl+Пробел (после возможно придется выбрать один из вариантов и нажать Enter для выбора).

Например:

main → public static void main(String[] args)

sysout → System.out.println()

Список всех аббревиатур можно просмотреть в настройках программы, кроме того, можно добавлять свои аббревиатуры и генерируемый по ним код.

Автогенерация кода в Eclipse учитывает также контекст применения и пытается подставить соответствующие имена переменных, полей и методов в генерируемый код.

Меню Source позволяет автоматически генерировать целые последовательности кода, решающие определенные широко распространенные задачи, например создание конструкторов, методов доступа, переопределение методов суперкласса и т.д. Вызывается меню сочетанием клавиш Alt+Shift+S (либо через навигацию по системе меню).

Меню Refactor содержит целый ряд действий для выполнения рефакторинга (переработки) исходного кода. Рефакторинг – это процесс изменения исходного кода без изменения функциональности программы. Применяется для улучшения дизайна, а также удобочитаемости программ. Примером рефакторинга является переименование любых элементов кода в пределах всего проекта (или нескольких проектов), выделение блоков исходного кода в методы, выделение интерфейсов из классов и т.д. Вызывается меню рефакторинга с помощью сочетания клавиш Alt+Shift+T (или через навигацию по системе меню). Обратите внимание, что данное меню является контекстно-зависимым.

Автоматическое исправление ошибок – это средство, позволяющее пользователю выполнять некоторые распространенные действия по устранению ошибок в программах (синтаксических и лексических) в автоматическом режиме. Каждый раз, когда редактор указывает на ошибку, есть возможность выполнить одно

из автоматических действий по ее исправлению, для этого необходимо установить курсор на источник ошибки и нажать Ctrl+1 (либо кликнуть на иконку ошибки в соответствующей строке). Например, когда в выражении используется необъявленная до этого переменная, есть возможность автоматически добавить ее объявление, а также провести ряд других действий. Помимо исправления ошибок данная функциональность может быть использована и в других случаях, например для инвертирования условий в условных операторах или переименования идентификатора, все зависит от контекста применения. Рассмотренные выше возможности среды Eclipse требуют детального самостоятельного изучения в процессе выполнения лабораторной работы.

3.4. Основы языка Java

Программы на Java - это набор пробелов, комментариев, разделителей, ключевых слов, идентификаторов, литеральных констант, операторов.

3.4.1. Комментарии

Java поддерживает три вида комментариев

Вид комментраия	**Описание комментариев**
/*Комментарий */	Многострочный комментарий
// Комментарий	Комментарий, располагающийся до конца текущей строки
/** Документация*/	Документирующий комментарий, обрабатывается при помощи утилиты javadoc, которая генерирует документацию в формате HTML

3.4.2. Разделители

Разделители - это символы, используемые для группировки и упорядочения текста программы. В языке Java к ним относятся следующие конструкции [1]:

() {} [] ; , .

Разделители {} используются для ограничения тела классов, методов и других конструкций. Разделители () используются в определении методов и для группировки выражений. Разделитель ‘;’ используется для ограничения ввода операторов, а символ . для доступа к переменным и методам класса. Разделитель ‘,’ чаще всего используется при передаче нескольких параметров методу класса. При организации доступа к элементам массива соответствующие индексы указываются с использованием разделителей []. Если необходимо обратиться к многомерному массиву, то для разделения индексов используется , (запятая).

3.4.3.Идентификаторы

С помощью идентификаторов осуществляется именование классов, методов и переменных. *Имя* или *идентификатор* в языке Java представляет собой неограниченную последовательность букв и цифр в формате **UNICODE**, причем первым символом должна быть буква. Язык различает прописные и строчные буквы. Имена классов начинаются с прописной буквы; если имя содержит несколько слов, то каждое слово начинается с прописной буквы.

Имена методов и переменных начинаются со строчной буквы; если имя содержит несколько слов, то каждое следующее слово начинается со строчной буквы.

Имена констант записываются полностью прописными буквами; если имя состоит из нескольких слов, то между ними ставится знак подчеркивания.

3.4.4. Ключевые слова

Следующие слова зарезервированы в языке Java как ключевые, и не могут использоваться в качестве идентификаторов:

abstract	**default**	**genetic**	**native**	**return**	**try**
boolean	**do**	**goto**	**new**	**short**	**var**
break	**double**	**if**	**null**	**static**	**void**
byte	**else**	**implements**	**operator**	**super**	**volatile**
case	**extends**	**import**	**outer**	**switch**	**while**
catch	**final**	**inner**	**package**	**synchronized**	
char	**finally**	**instanceof**	**private**	**this**	
class	**float**	**int**	**protected**	**throw**	
const	**for**	**interface**	**public**	**throws**	
continue	**future**	**long**	**rest**	**transient**	

Ключевые значения, определенные в Java: **false, true**.

3.4.5.Литералы

Константы в Java задаются их литеральным представлением. Литералы бывают следующих видов:

- целочисленный литерал;
- литерал значения с плавающей запятой (плавающей точкой);
- логический литерал;
- символьный литерал;
- строковый литерал;
- литерал для пустого объекта.

В таблице 1 приведены правила создания литералов

Таблица 1. Интерпретация литералов в Java

Тип литерала	**Форма записи**	**Пример**
Обычное целое(int)	Число	1234
Длинное целое(long)	числоl или числоL	12345678900L
Шестнадцатеричное целое	0xчисло	0xFF
Восьмеричное целое	0число	077
Вещественное(double)	число.число или .число	12.34
Вещественное(float)	числоf или числоF	12.34F
Вещественное (double)	числоd или числоD	12.34D
Знаковое	+число или -число	-6
Вещественное в	числоечисло или	1.2E3

экспоненциальной форме	числоЕчисло	
Один символ	'символ'	'j'
Строка символов	"символ"	"abcxyz"
Пустая строка	""	""
Символ «забоя»	\b	\b
Символ табуляции	\t	\t
Перевод строки	\n	\n
Перевод страницы	\f	\f
Возврат каретки	\r	\r
Двойные кавычки	\"	\"
Одинарные кавычки	\’	\’
Обратная косая	\\	\\
Символ	\uNNNN	\u0013
Булево значение	true или false	true

3.4.6. Переменные

Переменная представляет собой программную единицу для хранения соответствующего ей значения. Для объявления переменной синтаксис Java требует использования следующей

конструкции:

тип идентификатор [, индентификатор];

где *тип*- определяет тип данных переменной, т.е. диапазон значений, которые могут хранится в переменной; идентификатор - используется для именования переменной и служит связующим звеном при организации доступа к значениям переменной [1].

3.5. Типы данных, операции, массивы

Java представляет собой язык со строгой типизацией. Это означает, что каждая переменная и каждое выражение имеет тип данных, который должен быть известен во время компиляции программы. Типы ограничивают значения, которые может иметь переменная или результат исполнения выражения. Типы также ограничивают операции, которые исполняются над величинами и определяют значения операции. Строгая типизация позволяет обнаружить ошибку в момент компиляции программы.

Типы данных в Java делятся на две категории:

- *примитивные*, или *простые*, типы данных;
- с*сылочные* типы данных.

К примитивным типам данных относятся логический тип данных **boolean**, и численные типы данных. Численные типы данных представляют собой:

- целочисленные типы данных - **byte, short, int, long, char**;
- типы данных с плавающей точкой - **float**, **double**.

К ссылочным типам данных относятся типы классов, интерфейсов и типы массивов. Также существует специальный нулевой тип данных **null**.

3.5.1. Примитивные типы данных и операции

Численные типы данных делятся на целочисленные типы данных и типы данных с плавающей точкой.

3.5.1.1.Целочисленные типы данных

Целочисленные типы данных бывают следующих видов:

Тип	**Размер**	**Диапазон**
byte	8 бит	-128…127
short	16 бит	-32768…32767
int	32 бита	-2147483648…2147483647
long	64 бита	-9223372036854775808...9223372036854775807
char	16 бит	0...65635

Символьный тип данных используется для представления 16 разрядных символов в формате **UNICODE**.

Поскольку в Java нет указателей, то реально размеры типов полезны только для определения минимумов и максимумов переменных.

Операции над целыми типами

Все операции, которые производятся над целыми числами, можно разделить на следующие группы.

Арифметические	**Операции сравнения**
Сложение + Вычитание Умножение * Деление / Взятие остатка от деления % Инкремент ++ Декремент - -	Больше >\ Меньше <\ Больше или равно >= Меньше или равно <= Равно = = Не равно !=

Побитовые операции	Сдвиги
Дополнение ~ Конъюнкция & Дизъюнкция | Исключающее ИЛИ ^	Сдвиг влево < < Сдвиг вправо > >\ Беззнаковый сдвиг вправо >>>

Если используется префиксный оператор (++i), то соответствующая операция приращения или уменьшения значения на единицу исполняется до возвращения результата всего выражения. Если используется постфиксный оператор (i++), то соответствующая операция приращения или уменьшения исполняется после получения результата всего выражения.

Конструкторы, методы и константы для работы с целочисленными значениями находятся в стандартных Java *классах-оболочках* **Integer**, **Long** и **Character**.

Встроенные в Java целочисленные операторы не сигнализируют о переполнении значений. Исключение составляют операторы деления (/) и нахождения остатка (\%), вызывающее исключение **ArithmeticException** в том случае, если знаменатель равен нулю.

3.5.1.2. Типы данных с плавающей точкой

Типы данных для чисел с плавающей точкой бывают следующих видов:

Тип	Размер	Диапазон значений по модулю
float	32 бита	.40239846e-45f...3.40282347e+38f
double	64 бита	4.94065645841246544e-324 ...1.79769313486231570e+308

К данным с плавающими точками применимы все арифметические операции и сравнения, перечисленные для целых типов. Операндами этих операций могут быть переменные и/или литералы. Целые и вещественные значения можно смешивать в операциях.

К обычным вещественным числам добавлены три значения:

- **POSITIVE_INFINITY** – положительная бесконечность;
- **NEGATIVE_INFINITY** – отрицательная бесконечность;
- **NaN** - "не число".

Конструкторы, методы и константы для работы со значениями с плавающей точкой находятся в стандартных Java классах-оболочках **Float**, **Double** и **Math**.

3.5.1.3. Логический тип данных

Логический (булевский) тип данных служит для представления логической величины, имеющей одно из двух возможных значений:

- true - истина
- false - ложь

В Java 0 (ноль) не является эквивалентом для значения "ложь".

Над логическими данными можно выполнять операции присваивания, сравнение на равенство и неравенство, а также логические операции.

Логические операции

Логическая операция	Правило выполнения
Отрицание !	Отрицание меняет значение истинности
Конъюнкция &	Конъюнкция истинна, только если оба операнда истинны
Дизъюнкция |	Дизъюнкция ложна, только если оба операнда ложны
Исключающее ИЛИ (каре)	Исключающее ИЛИ истинно, только если значения операндов различны
Сокращенная конъюнкция &&	Правый операнд вычисляется только в том случае, если левый операнд имеет значение true
Сокращенная дизъюнкция ||	Правый операнд вычисляется только в том случае, если левый операнд имеет значение false

Практически всегда в Java используются сокращенные логические операции.

3.5.2.Операции присваивания

Простая операция присваивания записывается знаком равенства, слева от которого стоит переменная, а справа выражение, совместимое с типом переменной. Результатом операции является приведенное значение правой части.

Кроме простой операции присваивания есть еще 11 составных операций присваивания:

+=, -=,*=,/=, %=, &=, |=, ^=,<<=, >>=, >>>=.

3.5.3.Условная операция

Условие: *условие ? оператор_в_случае_выполнения* : оператор_в_случае_невыполнения

3.5.4. Выражения

Из констант и переменных, операций над ними, вызовов методов и скобок составляются выражения. При вычислении выражения выполняются четыре правила:

1. Операции одного приоритета вычисляются справа налево. Исключение: операции присваивания вычисляются справа налево.
2. Левый операнд вычисляется раньше правого.
3. Операнды полностью вычисляются перед выполнением операции.
4. Перед выполнением составной операции присваивания значение левой части сохраняется для использования в правой части.

Пример. Int a=3,b=5;b+=a+=b+=7; В результате вычислений второго выражения b получит значение 27.

Приоритет операции

Операции перечислены в порядке убывания приоритета.

1. Постфиксные операции ++ и --.
2. Префиксные операции ++ и --, дополнение ~ и отрицание !.
3. Приведение типа (тип).
4. Умножение *, деление / и взятие остатка %.
5. Сложение + и вычитание - .
6. Сдвиги <<, >>, >>> .
7. Сравнения >, <, >=, <=.
8. Сравнения ==, !=.
9. Побитовая конъюнкция &.
10. Побитовое исключающее ИЛИ ^.
11. Побитовая дизъюнкция |.
12. Конъюнкция &&.
13. Дизъюнкция ||.
14. Условная операция ?:.
15. Присваивания =,+=, -=, *=, /=, %=, \&=, ^=, |=, <<=, >>=, >>>=.

3.5.5.Приведение типов

При проведении компиляции Java программ происходит проверка соответствия типов данных и предотвращение неверных операций присваивания. Все сомнительные программные конструкции приводят к выдаче компилятором сообщений об ошибке в приведении типов.

Java поддерживает **неявное приведение** значений целого типа данных в типы с плавающей точкой. Обратное преобразование некорректно и компилятор выдаст ошибку.

Явное приведение типов осуществляется при помощи оператора:

(тип) значение_другого_типа

Результат арифметической операции имеет тип **int**, кроме случая, когда один из операндов типа **long**. В этом случае результат будет типа **long**. Перед выполнением арифметической операции всегда происходит повышение типов **byte**,

short, **char**. Они преобразуются в тип **int**, а может быть, и в тип **long**, если другой операнд типа **long**.

Нельзя преобразовать тип данных с плавающей точкой в логический тип данных и наоборот. При преобразовании типа данных с большей точностью представления значений (например, **double**) к типу данных, поддерживающих меньшую точность представления

(например, **float**) возможна потеря данных.

3.5.6.Строки

Для представления строк в Java существует специальный тип данных(класс) **String**. В Java не существует перегрузка операторов, однако для строк сделано исключение - непосредственно при реализации языка программирования Java перегружен бинарный оператор сложения "+"(а также унарный оператор "+=") для строк, то есть реализована конкатенация строк.

Переменная, содержащая строку, инициализируется посредством присвоения ей строкового литерала, например:

String s="Ura!"

Конкатенация строк осуществляется при помощи оператора сложения. В Java для каждого класса определен метод **toString**(), предназначенный для преобразования объектов этого класса в строку. Для примитивных типов данных существуют классы-оболочки, реализующие преобразование типов. Для пустой ссылки **null** результатом работы метода **toString**() будет строка "null".

3.5.7 Массивы

Ссылочные типы делятся на массивы, классы и интерфейсы.

Массивом в Java называется упорядоченный набор элементов. В качестве элементов массива могут выступать как данные примитивных типов, так и ссылки на объекты, включая ссылки на другие массивы. Массивы, наряду с экземплярами классов, являются объектами.

Массив объявляется и создается при помощи выражения:

ТипЭлементов []*имяМассива* = **new** т*ипЭлементов* [*размерМассива*]

или, эквивалентно - в зависимости от выбора программиста:

ТипЭлементов имяМассива [] = **new** *типЭлементов* [*размерМассива*]

Итак, для создания массива необходимо выполнить описанные ниже действия:

1. Объявить массив - задать тип элементов, которые будут содержаться в массиве и присвоить ему имя.

2. Выделить для массива память - задать количество его элементов (используется оператор **new**).

3. Инициализировать массив, поместив в его элементы нужные данные.

Первый элемент массива имеет индекс 0, а последний *размерМассива*-1. Размер массива хранится в поле **length** данного объекта.

При обращении к элементу массива проверяется, соответствует ли индекс этого элемента допустимому диапазону значений - в противном случае возбуждается исключение **IndexOutOfBound**s.

Доступ к элементам массива имеет вид:

массив[*индекс*]

Доступ к полям массива имеет вид:

массив.поле

Доступ к полям и методам экземпляров класса, являющихся элементами массива, имеет вид:

массив[*индекс*].*поле*

массив[*индекс*].*метод(параметрыМетода*)

Можно создавать массив, элементы которого являются массивами. Объявление и создание многомерных массивов имеет вид:

типЭлемента [][] ...*имяМассива* = **new** *типЭлементов*} [размерМассива1] [*размерМассива2*]...

Например:

int [][][] = **new** ia[3][3][4]

При создании многомерного массива обязательно требуется указывать слева первый размер. Другие размеры для вложенных массивов можно указывать позже при помощи оператора **new**.

Инициализирующие значения массива задаются в фигурных скобках сразу же после объявления массива, например:

int [] simple={2,3,5,7}

int [][] B= {{1,0,0\},\{1,1,0\},\{1,2,1}}

3.6. Использование аргументов командной строки

Иногда будет требоваться передать определенную информацию программе во время ее запуска. Для этого используют аргументы, командной строки метода main (). Аргумент командной строки — это информация, которую во время запуска программы задают в командной строке непосредственно после ее имени. Доступ к аргументам командной строки внутри Java-программы не представляет сложности — они хранятся в виде строк в массиве String, переданного методу main (). Первый аргумент командной строки хранится в элементе массива args [0], второй — в элементе args [0] и т.д. Например, следующая программа отображает все аргументы командной строки, с которыми она вызывается.

```
// Отображение всех аргументов командной строки.
class CommandLine {
public static void main(String args[]) {
        (int i=0; args.length; i++)
                System.out.println("args[" + i + "] : " + args[i]);
        }
}
```

Попытайтесь выполнить эту программу, введя следующую строку:

Java CommandLine this is a test 100 -1

В результате отобразится следующий вывод:

```
args[0] this
args[1] is
args[2] a
args[3] test
args[4] 100
args[5] -1
```

Все аргументы командной строки передаются как строки. Численные значения нужно преобразовать в их внутренние представления.

3.7. Обработка ошибок с помощью исключений

Исключение Java представляет собой объект, который описывает исключительную (то есть ошибочную) ситуацию, возникающую в части программного кода. Когда такая ситуация возникает, создается объект, представляющий исключение, который возбуждается в методе, вызвавшем ошибку. Этот метод может либо обработать исключение самостоятельно, либо пропустить его. В обоих случаях, в некоторой точке исключение перехватывается и обрабатывается. Исключения могут генерироваться системой времени выполнения Java, либо они могут быть сгенерированы вручную вашим кодом.

Обработка исключений Java управляется пятью ключевыми словами: try, catch, throw, throws и finally. Если кратко, они работают следующим образом. Операторы программы, которые вы хотите отслеживать на предмет исключений, помещаются в блок try. Если исключение возникает в блоке try, оно возбуждается. Ваш код может перехватить исключение (используя catch) и обработать его некоторым осмысленным способом. Сгенерированные системой исключения автоматически возбуждаются системой времени выполнения Java. Чтобы вручную возбудить исключение, используется ключевое слово throw. Любое исключение, которое возбуждается внутри метода, должно быть специфицировано в его определении ключевым словом throws. Любой код, который в обязательном порядке должен быть выполнен после завершения блока try, помещается в блок finally.

Типы исключений

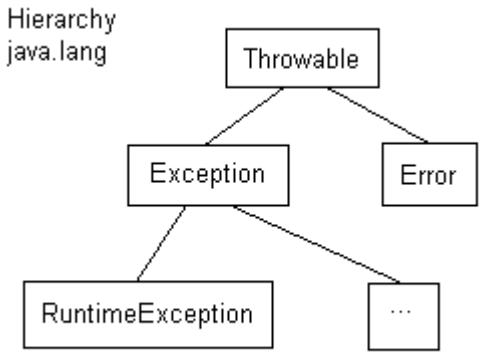

Рисунок 1 – Иерархия классов

Все типы исключений являются подклассами встроенного класса Throwable. То есть Throwable расположен на вершине иерархии классов исключений. Немедленно под Throwable находятся два подкласса, которые разделяют все исключения на две отдельные ветви. Одну ветвь возглавляет Exception. Этот класс используется для исключительных условий, которые пользовательская программа должна перехватывать Это также класс, от которого вы будете наследовать свои подклассы при создании собственных типов исключений. У класса Exception имеется важный подкласс RuntimeException, исключения этого типа автоматически определяются для всех программ и включает такие вещи, как деление на ноль и ошибочная индексация массивов.

Другая ветвь начинается с класса Error, определяющего исключения, вызов которых не ожидается при нормальном выполнении программы. Исключения типа Error используются системой времени выполнения Java для обработки ошибок, происходящих внутри самого окружения.

Причины возникновения исключений

Исключение генерируется по одной из трех причин:

1. Виртуальной машиной Java было обнаружено ошибочное состояние. Такие состояния возникают потому, что:
   - формула оценки выражения нарушает нормальную семантику языка Java, например, целое число делится на ноль.
   - ошибка происходит при загрузке или компоновке части программы
   - некий ограниченный ресурс превышен, например, такой как использование слишком большого количества памяти

Эти исключения не генерируются в произвольном месте программы, а генерируются там, где они определяются как возможный результат формулы оценки выражения или выполнения оператора.

2. Был выполнен оператор throw.
3. Асинхронное исключение произошло потому, что:
   - был вызван метод stop класса Thread
   - в виртуальной машине произошла внутренняя ошибка

Самостоятельное возбуждений исключений

Как и многие объекты в Java, объект исключения создается, используя new, который резервирует область памяти и вызывает конструктор. Есть два конструктора для всех стандартных исключений: первый - конструктор по умолчанию, и второй принимает строковый аргумент – некая поясняющая информация:

```
if(t == null)
  throw new NullPointerException("t = null");
```

Непосредственно выброс осуществляется с помощью ключевого слова throw. Объект, в результате, “возвращается” из метода, даже если метод обычно не возвращает этот тип объекта. Простой способ представлять себе обработку исключений, как альтернативный механизм возврата, хотя для этого он не предназначен.

Различные классы исключений соответствуют различным типом ошибок. Информация об ошибке представлена и внутри объекта исключения, и выбранным типом исключения. (Часто используется только информация о типе объекта исключения и ничего значащего не хранится в объекте исключения.)

Повторное выбрасывание исключений

Иногда в процессе обработки исключения возникает необходимость повторно выбросить исключение. Так как ссылка на текущее исключение доступна, можно просто вновь выбросить эту ссылку:

```
catch(Exception e) {
  System.err.println("An exception was thrown");
  throw e;
```

```
}
```

Повторное выбрасывание исключения является причиной того, что исключение переходит в обработчик следующего, более старшего контекста. Все остальные предложения *catch* для того же самого блока *try* игнорируются. Кроме того, все, что касается объекта исключения, сохраняется, так что обработчик старшего контекста, который поймает исключение этого специфического типа, может получить всю информацию из этого объекта.

Обработка исключения

Если метод выбросил исключение, он должен предполагать, что исключение будет “поймано” и устранено. Одно из преимуществ обработки исключений Java в том, что это позволяет разделить реализацию основной функциональности приложения от обработки ошибок.

Чтобы увидеть, как ловятся исключения, необходимо понять концепцию критического блока. Он является секцией кода, которая может произвести исключение и за которым следует код, обрабатывающий это исключение.

Блок try

Код, который необходимо проверять на возможность исключения необходимо поместить в блок try. Он называется блоком проверки, блок проверки – это обычный блок, которому предшествует ключевое слово try:

```
try {
  // Код, который может сгенерировать исключение
}
```

Блок обработки исключений

Конечно, выбрасывание исключения должно где-то заканчиваться. Это “место” - обработчик исключения, каждому типу исключения соответствует свой блок обработки. Обработчики исключений следуют сразу за блоком проверки и объявляются ключевым словом catch:

```
try {
  // Код, который может сгенерировать исключение
} catch(Type1 id1) {
  // Обработка исключения Type1
} catch(Type2 id2) {
  // Обработка исключения Type2
} catch(Type3 id3) {
  // Обработка исключения Type3
}

// и так далее...
```

Если исключение выброшено, механизм обработки исключений выбирает обработчик с таким аргументом, тип которого совпадает с типом исключения. Затем происходит вход в предложение *catch*, где происходит обработка исключения. Поиск

обработчика, после остановки на предложении *catch*, заканчивается. Выполняется только совпавшее предложение *catch*.

Обратите внимание, что внутри блока проверки несколько вызовов различных методов может генерировать одно и тоже исключение, но для обработки необходим только один обработчик.

Можно создать обработчик, ловящий любой тип исключения. Для этого необходимо перехватывать исключение базового типа – Exception (есть другие типы базовых исключений, но Exception – это базовый тип, которому принадлежит фактически вся программная активность):

```
catch(Exception e) {
  System.err.println("Caught an exception");
}
```

Выполнение очистки с помощью блока finally

Часто необходимо выполнять блоки кода, независимо от того, было ли выброшено исключение в блоке *try*, или нет. Это обычно относится к некоторым операциям, отличным от утилизации памяти (так как об этом заботится сборщик мусора). Для достижения этого эффекта используется предложение *finally* в конце списка всех обработчиков исключений. Полная картина секции обработки исключений выглядит так:

```
try {
  // Критическая область: Опасная активность,
  // при которой могут быть выброшены A, B или C
} catch(A a1) {
  // Обработчик ситуации A
} catch(B b1) {
  // Обработчик ситуации B
} catch(C c1) {
  // Обработчик ситуации C
} finally {
  // Действия, совершаемые всякий раз
}
```

Пример, демонстрирующий, использование finally:

```
//: c10:FinallyWorks.java
// Предложение finally выполняется всегда.

class ThreeException extends Exception {}

public class FinallyWorks {
  static int count = 0;
  public static void main(String[] args) {
    while(true) {
      try {
```

```
                    // Пост-инкремент, вначале равен нулю:
          if(count++ == 0)
            throw new ThreeException();
          System.out.println("No exception");
        } catch(ThreeException e) {
          System.err.println("ThreeException");
        } finally {
          System.err.println("In finally clause");
          if(count == 2) break; // выйти из "while"
        }
      }
    }
  } ///:~
```

Вот что получается на выходе:

```
ThreeException
In finally clause
No exception
In finally clause
```

Независимо от того, было выброшено исключение или нет, предложение finally выполняется всегда.

3.8. Работа с потоками ввода-вывода

Стандартные классы Java, инкапсулирующие работу с данными (оболочечные классы для числовых данных, классы String и StringBuffer, массивы и т.д.), не обеспечивают поддержку чтения этих данных из файлов или запись их в файл. Вместо этого используется весьма гибкая и современная технология использования *потоков* (Streams).

Поток представляет накапливающуюся последовательность данных, поступающих из какого-то источника. Порция данных может быть считана из потока, при этом она из потока изымается. В потоке действует принцип очереди – “первым вошёл, первым вышел”.

В качестве источника данных потока может быть использован как стационарный источник данных (файл, массив, строка), так и динамический – другой поток. При этом в ряде случаев выход одного потока может служить входом другого. Поток можно представить себе как трубу, через которую перекачиваются данные, причём часто в таких “трубах” происходит обработка данных. Например, поток шифрования шифрует данные, полученные на входе, и при считывании из потока передаёт их в таком виде на выход. А поток архивации сжимает по соответствующему алгоритму входные данные и передаёт их на выход. “Трубы” потоков можно соединять друг с другом – выход одного со входом другого. Для этого в качестве параметра конструктора потока задаётся имя переменной, связанной с потоком - источником данных для создаваемого потока.

Буферизуемые потоки имеют хранящийся в памяти промежуточный буфер, из которого считываются выходные данные потока. Наличие такого буфера позволяет повысить производительность операций ввода-вывода, а также осуществлять дополнительные операции – устанавливать метки (маркеры) для какого-либо элемента, находящегося в буфере потока и даже делать возврат считанных элементов в поток (в пределах буфера).

Абстрактный класс InputStream (“входной поток”) инкапсулирует модель входных потоков, позволяющих считывать из них данные. Абстрактный класс OutputStream (“выходной поток”) - модель выходных потоков, позволяющих записывать в них данные. Абстрактные методы этих классов реализованы в классах-потомках.

Методы класса InputStream:

Метод	**Что делает**
int available()	Текущее количество байт, доступных для чтения из потока.
int read()	Читает один байт из потока и возвращает его значение в виде целого, лежащего в диапазоне от 0 до 255. При достижении конца потока возвращает -1.
int read(byte[] b) int read(byte[] b, int offset, int count)	Пытается прочесть b.length байт из потока в массив b. Возвращает число реально прочитанных байт. После достижения конца потока последующие считывания возвращают -1.
long skip(long count)	Попытка пропускать (игнорировать) count байт из потока. count ≤0, пропуска байт нет. Возвращается реально пропущенное число байт. Если это значение ≤0, пропуска байт не было.
boolean markSupported()	Возвращает true в случае, когда поток поддерживает операции mark и reset, иначе – false.
void mark(int limit)	Ставит метку в текущей позиции начала потока. Используется для последующего вызова метода reset, с помощью которого считанные после установки метки данные возвращаются обратно в поток. Эту операцию поддерживают не все потоки. См. метод markSupported.
void reset()	Восстанавливает предшествующее состояние данных в начале потока, возвращая указатель начала потока на помеченный до того меткой элемент. То есть считанные после установки метки данные возвращаются обратно в поток. Попытка вызова при отсутствии метки или выходе её за пределы лимита приводит к возбуждению исключения IOException. Эту операцию поддерживают не все потоки. См. методы markSupported и mark.
void close()	Закрытие потока. Последующие попытки чтения из этого потока приводят к возбуждению исключения IOException.

Все методы класса InputStream, кроме markSupported и mark, возбуждают исключение IOException – оно возникает при ошибке чтения данных.

Методы класса OutputStream:

Метод	**Что делает**
void write(int b)	Записывает один байт в поток. Благодаря использованию типа int можно использовать в качестве параметра целочисленное выражение без приведения его к типу byte. Напомним, что в выражениях “короткие” целые типы автоматически преобразуются к типу int.
void write(byte[] b) void write(byte[] b, int offset, int count)	Записывает в поток массив байт. Если заданы параметры offset и count, записывается не весь массив, а count байт начиная с индекса offset.
void flush()	Форсирует вывод данных из выходного буфера и его очистку. Необходимость этой операции связана с тем, что в большинстве случаев данные уходят “во внешний мир” на запись не каждый раз после вызова write, а только после заполнения выходного буфера. Таким образом, если операции записи разделены паузой (большим интервалом времени), и требуется, чтобы уход данных из выходного буфера совершался своевременно, после последнего вызова оператора write, предшествующего паузе, надо вызывать оператор flush.
void close()	Закрытие потока. Последующие попытки записи в этот поток приводят к возбуждению исключения IOException.

Все методы этого класса возбуждают IOException в случае ошибки записи.

Все бинарные потоки наследуются от двух классов - InputStream и OutputStream. Нам важны два потомка - FileInputStream иFileOutputStream соответственно. Вот кусок текста программы, которая копирует содержимое файла *a.txt* в файл *b.txt*:

```
FileInputStream in = null;
FileOutputStream out = null;
try {
 in = new FileInputStream("a.txt");
 out = new FileOutputStream("b.txt");
 int c;
 while ((c = in.read()) != -1) { out.write(c); }
} finally {
 if (in != null) { in.close(); }
 if (out != null) { out.close(); }
}
```

Оба класса обладают удобными конструкторами с аргументом пути к файлу. Если *b.txt* не существует, то будет создан. Иначе - переписан. Не забывайте отлавливать исключения (или же указывать их в заголовках методов).

На практике работать с двоичным представлением данных при написании олимпиадных задач не приходится. Небольим шажком вперед послужат FileReader и FileWriter, которые очень похожи на бинарные потоки но работают с символами.

Новую возможность предоставляют BufferedReader и PrintWriter, которые можно рассматривать, как обертки для уже предыдущей пары классов. Эта возможность - построчное чтение и запись файла. Приведем все тот же пример копирования файла:

```
BufferedReader in = null;
PrintWriter out = null;
try {
 in = new BufferedReader(new FileReader("a.txt"));
 out = new PrintWriter(new FileWriter("b.txt"));
 String l;
 while ((l = in.readLine()) != null) { out.println(l); }
} finally { ... }
```

Класс PrintWriter очень близок к стандартным возможностям языка C. Методы **print()** и **println()** определены для всех стандартных типов (последний добавляет перевод строки в конец вывода), а метод **printf()** очень похож на соответсвующую процедуру языка C.

BufferedReader обладает также методом **readLine()** позволяющим выполнять чтение строк.

Начиная с версии Java **1.5**, в пакете java.util существует класс Scanner. Для каждого из базовых типов (а также классов длинной арифметики) имеется пара методов: **hasNextT()** говорит, можно ли далее прочесть элемент типа **T**, в то время как **nextT()** этот элемент пытается считать. Методы **hasNext()** и **next()** работают с отдельными словами.

С использованием класса Scaner возможно чтение как с калвиатуры так и из файла. Следующий пример демонстрирует чтение различных типао данных из файла:

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

public class ReadFileByScanner {

	public static void main(String[] args) {
//
		// Загружаем созданный нами текстовый файл
		//
		File file = new File("testfile.txt");
		try {
			//
			// Объявляем класс Scanner, инициализируем его с параметром file
```

```
                    // Создаем цикл, который будет считывать строки, пока не дойдем
                // до конца файла.
                //
                Scanner scanner = new Scanner(file);
                while (scanner.hasNextLine()) {
                    if (scanner.hasNextInt()) {
                        int i = scanner.nextInt();
                        System.out.println("Tun Integer: " + i);
                    } else if (scanner.hasNextDouble()) {
                        double d = scanner.nextDouble();
                        System.out.println("Tun double: " + d);
                    } else if (scanner.hasNextBoolean()) {
                        Boolean b = scanner.nextBoolean();
                        System.out.println("Tun boolean: " + b);
                    } else {
                        System.out.println("String: " + scanner.next());
                    }
                }
            } catch (FileNotFoundException e) {
                e.printStackTrace();
            }
        }
}
```

4. ВАРИАНТЫ ЗАДАНИЙ

Таблица 4.1 Варианты заданий

№	**Содержание задания**
1.	Написать программу “калькулятор” (операции: сложение, вычитание, умножение и деление). Входные данные поступают в формате “число операция число”.
2.	Написать программу — возведения числа m в степень числа n.
3.	Написать программу вычисления длины периметра и площади прямоугольного треугольника.
4.	Написать программу, рассчитывающую элементы ряда Фибоначчи. Каждый элемент этого ряда равен сумме двух предыдущих: $X_n=X_{(n-1)}+X_{(n-2)}$. Полагать $X_0=1$ и $X_1=1$. Вычислять до заданного номера элемента ряда. Вывести номер элемента ряда, его значение, шестнадцатеричный, восьмеричный, двоичный код.
5.	Написать программу вычисления частного и остатка от деления двух целых чисел
6.	Написать программу, печатающую символы от “A” до введенного с клавиатуры символа (последний возможный “Z”). Для каждого символа вывести номер, сам символ, шестнадцатеричный, восьмеричный и двоичный код этого символа.
7.	Написать программу вычисления наибольшего общего делителя двух целых чисел. Наибольший общий делитель NOD(m, n) может быть вычислен рекурсивно следующим образом: if m mod n equals 0 then n; else NOD(n, m mod n).
8.	Написать программу вычисления длины окружности и площади круга по заданному радиусу.
9.	Написать программу вычисления длины периметра и площади прямоугольника по заданным длине и ширине.
10.	Написать программу, которая обрабатывает все символы введенной строки. Каждый символ печатается в новой строке, также печатаются его двоичный, десятичный и восьмеричный коды.
11.	Найти все положительные степени двойки, значение которых не превышает величины введенного с клавиатуры числа.
12.	Написать программу преобразования температуры в градусах Цельсия (например: *15C*) в температуру в градусах по Фаренгейту (например: *59F*), и наоборот. 0 градусов по Цельсию соответствует 32 градусам по Фаренгейту. Изменение температуры на 1 градус по Цельсию соответствует изменению на 1.8 градуса по Фаренгейту.
13.	Написать программу решения квадратного уравнения по заданным коэффициентам a,b и c.
14.	Написать программу нахождения всех возможных делителей введенного числа.
15.	Написать программу решения линейного уравнения по заданным коэффициентам k и b.

5. СОДЕРЖАНИЕ ОТЧЕТА

Отчет должен содержать:

Титульный лист, цель работы, постановку задачи, вариант задания, текст программы с комментариями, скриншоты выполнения и описание тестовых примеров, выводы по работе.

6. КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Каким образом выполняются программы, написанные на языке Java?
2. Что такое JDK?
3. Для каких целей используется среда Eclipse?
4. Какие типы данных поддерживает язык Java?
5. Как в языке Java определить шестнадцатеричное целое?
6. Приведите пример работы с массивом на языке Java?
7. Поясните, каким образом возможно получить доступ к аргументам командной строки Java программы?
8. Расскажите об обработке исключений в Java?
9. Каким образом в Java возможно осуществить чтение/запись с консоли/из файла?

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Ноутон, П. Java™ 2 [Текст] : пер. с англ. / П. Ноутон, Г. Шилдт. - СПб. : БХВ – Петербург, 2007. - 1050 с.

2. Шилдт, Г. Искусство программирования на Java [Текст] : пер. с англ. / Г. Шилдт, Д. Холмс. - М. ; СПб. ; К. : Вильямс, 2005. - 334 с

3. Хабибуллин, И. Ш. Java 2 [Текст] : самоучитель / И. Ш. Хабибуллин. - СПб. : БХВ - Петербург, 2005. - 720 с.

Заказ № _____ от «_____» _____2015г. Тираж _____экз.
Изд-во СевГУ