Е. Г. Канель, З. Фрайман

Основы программирования на JAVA

для школьников... и не только

Е. Г. Канель, З. Фрайман

ОСНОВЫ ПРОГРАММИРОВАНИЯ НА JAVA

Для школьников... и не только

МОСКВА

ББК 32.973-018

Канель Евгений Гогаевич, Фрайман Зэев

Основы программирования на Java: Для школьников... и не только. — М.: ЛЕНАНД, 2019. — 200 с.

Курс «Основы компьютерных наук для средней школы» является базовым курсом по основам программирования. Данное издание рассчитано на учеников старших классов (9–11), не имеющих опыта в написании компьютерных программ или имеющих минимальный опыт. Такой подход позволяет использовать учебник и в рамках коллективных занятий (на уроках, факультативах или кружках), и при самостоятельном индивидуальном обучении.

Пособие содержит как общие теоретические положения подхода к написанию программ, так и описание практических приемов, а также разбор соответствующих излагаемому материалу примеров.

Книга написана на основании базовой части курса «Компьютерные науки», преподаваемого в 9–12 классах израильской школы. Авторы — ведущие преподаватели этого курса, с опытом работы более 25 лет.

Формат 60×90/16. Печ. л. 12,5. Зак. № АО-5804.

Отпечатано в ООО «ЛЕНАНД».
117312, Москва, проспект Шестидесятилетия Октября, 11А, стр. 11.

ISBN 978-5-9710-5765-9

© ЛЕНАНД, 2018

23573 ID 237142

Все права защищены. Никакая часть настоящей книги не может быть воспроизведена или передана в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на магнитный носитель, а также размещение в Интернете, если на то нет письменного разрешения владельца.

Оглавление

Раздел 1. Краткая история и перспективы языка Java 5

Раздел 2. Базовая структура программы на Java. Значение комментариев при написании программы. Этапы работы: написание, отладка, прогон 9

Раздел 3. Переменные и основные типы переменных. Объявление и инициализация переменных 19

Раздел 4. Немного про символьные и строковые типы 27

Раздел 5. Команды вывода и ввода информации 28

Раздел 6. Присвоение данных между переменными разных типов (приведение типов, casting). Специальные операторы 37

Раздел 7. Вычисление частного и остатка 46

Раздел 8. Команда выбора (ветвления) 51

Раздел 9. Короткий оператор выбора (if без else) 53

Раздел 10. Стандартная команда выбора (if… else) 56

Раздел 11. Простая команда выбора с блоком (блоками) 58

Раздел 12. Команда if со сложным условием 60

Раздел 13. Булевы значения и переменные 64

Раздел 14. Наиболее используемые функции библиотеки Math. Использование функции random — работа со случайными числами 66

Раздел 15 (часть 1). Команды повтора — Цикл for 70

Раздел 15 (часть 2). Команды повтора — Цикл for 78

Раздел 16. Вычисления с использованием команды повтора — Цикл for 82

Раздел 17. Вычисления с использованием команды повтора — Цикл for (продолжение) 85

Раздел 18. Определение минимума и максимума в циклах — Цикл for 89

Раздел 19 (часть 1). Структура и использование цикла while 93

Раздел 19 (часть 2). Структура и использование цикла do... while 99

Раздел 20. Вложенные циклы 100

Раздел 21. Методы 103

Раздел 22. Массивы 110

Раздел 23 (часть 1). Массивы и методы 120

Раздел 23 (часть 2). Массивы и методы 127

Раздел 24 (часть 1). Массивы счетчиков 132

Раздел 24 (часть 2). Массивы сумматоров 137

Раздел 25. Массив массивов 138

Объекты в языке Java 150

Заключение, которое можно считать вступлением 197

Раздел 1

Краткая история и перспективы языка Java

Для краткого знакомства с историей языка Java можно воспользоваться нашей книгой, а человек, владеющий даже минимальными, основными, навыками поиска информации в Интернете, может достаточно быстро найти много дополнительных сведений на эту тему в Сети. Достаточно набрать практически в любой поисковой машине фразу «история языка Java» или что-то подобное этой фразе, — и к вашим услугам будут миллионы ссылок. Разумеется, практически все они будут содержать мало чем различающиеся между собой тексты — просто потому, что история этого языка все еще достаточно коротка.

Язык Java в самом своем «зародышевом», первоначальном варианте начал создаваться группой разработчиков в 1991 году, — и цель перед собой эта группа ставила достаточно скромную. Речь шла всего-навсего о создании компьютерного языка, с помощью которого можно было бы и просто, и одновременно универсально решить проблемы программирования бытовых приборов, имеющих встроенные микропроцессоры.

Прошло менее десятка лет и оказалось, что из сравнительно скромной задачи «вырос» один из самых мощных инструментов сегодняшнего мира программирования. Интернет, мир мобильных телефонов (и, в первую очередь, смартфонов), мир планшетных компьютеров, самых разных и совершенно неожиданных устройств — сегодняшняя сфера приложения языка Java. И практически ни у кого нет сомнения, что сфера использования и применения языка Java будет постоянно расширяться.

Вот как изложил первый этап работы по созданию языка Java (с 1991 по 1995 гг., когда и было официально объявлено о создании языка Java) Майкл О'Коннелл в статье «Java: The inside story»[1], написанной им для журнала SunWorld Online в июле 1995 г.

---

[1] Полный вариант статьи на английском языке можно найти по адресу: http://www.sun.com/sunworldonline/swol-07-1995/swol-07-java.html

История разработки языка Java, изложенная Патриком Нотоном

5 декабря 1990 г. — Нотон отказывается от предложения перейти в компанию NeXT и начинает работу в компании Sun над проектом, получившим впоследствии название Green.

15 января 1991 г. — Совещание типа мозгового штурма по проекту Stealth (названном так Скоттом Макнили) в Аспене, в котором участвовали Билл Джой, Энди Бехтолсхейм, Уэйн Розинг, Майк Шеридан, Джейм Гослинг и Патрик Нотон.

1 февраля 1991 г. — Гослинг, Шеридан и Нотон всерьез берутся за работу. Нотон занимается графической системой Aspen, Гослинг — идеями языка программирования, Шеридан — бизнес-разработкой.

8 апреля 1991 г. — Переезд по новому адресу и разрыв прямого соединения с локальной сетью (и большинством других средств связи) компании Sun; проект продолжается под названием Green.

15 апреля 1991 г. — К проекту Green присоединяются Эд Фрэнк (архитектор системы SPARCstation 10), Крейг Форрест (дизайнер чипа SS10) и Крис Уорт (разработчик системы NeWS).

Май 1991 г. — Эд Фрэнк присваивает прототипу аппаратуры название *7 (или Star7; *7 — код, который было необходимо набрать в офисе Sand Hill, чтобы ответить на любой звонок с любого телефона).

Июнь 1991 г. — Гослинг начинает работу над интерпретатором Oak, который через несколько лет (при поисках торговой марки) переименован в Java.

1 августа 1991 г. — Осуществлено объединение Oak и Aspen; заработала их первая реальная программа.

19 августа 1991 г. — Коллектив разработчиков Green демонстрирует идеи базового пользовательского интерфейса и графическую систему сооснователям компании Sun Скотту Макнили и Биллу Джою.

17 октября 1991 г. — Шеридан и Нотон присваивают конструкторской философии своего коллектива девиз «1st Person», который со временем становится названием компании.

17 ноября 1991 г. — Офис проекта Green снова подключается к главной сети компании Sun линией на 56 Кбит/с.

1 марта 1992 г. — К проекту Green присоединяется Джонатан Пейн, который позднее участвует в написании HotJava.

Лето 1992 г. — Интенсивная деятельность по доработке Oak, Green OS, пользовательского интерфейса, аппаратуры Star7 и соответствующих компонентов.

4 сентября 1992 г. — Завершена разработка устройства Star7; оно продемонстрировано Джою и Макнили.

1 октября 1992 г. — Из компании SunLabs в штат переходит Уэйн Розинг, принимающий на себя руководство коллективом.

1 ноября 1992 г. — Организована корпорация FirstPerson.

15 января 1993 г. — Коллектив переезжает в Пало-Альто, в здание, где раньше находилась лаборатория Western Research Lab компании DEC, и была основана исходная группа Hamilton Group (она же OSF).

15 марта 1993 г. — После ознакомления с результатами испытаний кабельного интерактивного телевидения, проведенных компанией Time Warner, корпорация FirstPerson сосредотачивается на этой тематике.

Апрель 1993 г. — Выпуск первого графического браузера для Internet — Mosaic 1.0, разработанного в центре NCSA.

14 июня 1993 г. — Компания Time Warner продолжает проводить свои испытания интерактивного кабельного ТВ с компанией SGI, несмотря на признанное превосходство технологии компании Sun и уверения, что Sun выиграла эту сделку.

Лето 1993 г. — Нотон пролетает 300 тыс. миль, продавая Oak всем занимающимся бытовой электроникой и интерактивным телевидением; тем временем темп, с которой люди получают доступ к Internet, головокружительно нарастает.

Август 1993 г. — Через несколько месяцев многообещающих переговоров с компанией 3DO относительно разработки ОС для приставок, президент 3DO Трип Хокинс предлагает купить технологию. Макнили отказывается, и сделка срывается.

Сентябрь 1993 г. — К коллективу присоединяется Артур Ван Хофф, — поначалу чтобы создать среду разработки приложений, предназначенных для интерактивного телевидения, а потом разрабатывающий главным образом сам язык.

7 декабря 1993 г. — Экспертиза операций на высоком уровне в FirstPerson обнаруживает, что эта группа не имеет реальных партнеров или маркетинговой стратегии и неясно представляет себе дату выпуска.

8 февраля 1994 г. — Отменено публичное заявление компании FirstPerson о выпуске, которое должно было состояться на конференции Technology, Entertainment and Design (TED).

17 февраля 1994 г. — Исполнительным лицам компании Sun для разносторонней экспертизы представлен альтернативный бизнес-план корпорации FirstPerson по разработке мультимедийной платформы для CD-ROM и онлайновой работы.

25 апреля 1994 г. — Создана компания Sun Interactive; в нее переходит половина сотрудников FirstPerson.

Июнь 1994 г. — Начат проект Liveoak, нацеленный Биллом Джоем на использование Oak в крупном проекте небольшой операционной системы.

Июль 1994 г. — Нотон ограничивает область применения проекта Liveoak, просто переориентировав Oak на Internet.

16 сентября 1994 г. — Пейн и Нотон начинают писать WebRunner — браузер типа Mosaic, позднее переименованный в HotJava.

29 сентября 1994 г. — Прототип HotJava впервые продемонстрирован исполнительным лицам компании Sun.

11 октября 1994 г. — Нотон уходит в компанию Starwave.

Осень 1994 г. — Ван Хофф реализует компилятор Java на языке Java. (Ранее Гослинг реализовывал его на языке C).

23 мая 1995 г. — Компания Sun официально представляет Java и HotJava на выставке SunWorld '95.

Раздел 2

Базовая структура программы на Java. Значение комментариев при написании программы. Этапы работы: написание, отладка, прогон

Базовая структура программы на Java

Программа на языке Java — это набор указаний, которые язык исполняет, чтобы была достигнута определенная цель (как правило, поставленная в задании).

Программа на языке Java называется *классом*, а собственно текст класса часто также называют *кодом*.

Следует иметь в виду, что язык Java — «чувствительный к размеру букв». Это означает, что составляющие кода должны быть записаны с использованием букв того размера (большие или маленькие), как это оговорено и установлено в правилах языка. Следует обращать на это каждый раз внимание, так как запись кода с буквами «не того размера» является одной из самых распространенных ошибок, особенно на начальном этапе.

Классы могут быть практически любого размера и уровня сложности, но существует минимальный код, состоящий из таких составляющих, которые являются обязательными для любого класса. Мы будем называть такой код *базовым кодом* или *минимальным классом*, то есть классом с таким кодом, меньше которого быть в классе просто не может.

Приведем этот класс, а также сопроводим его пояснениями к каждой его составляющей.

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

  }
}
```

Теперь приведем комментарии к каждой части этого минимального класса.

class osnova

Заголовок класса содержит служебное слово (то есть слово, являющееся обязательным и не допускающим изменений элементом языка) **class** и имя класса, которое можно выбирать по желанию.

Для имени класса существуют следующие ограничения и требования:

- Им может быть только одно слово.
- В нем можно использовать буквы (только английского алфавита), цифры и некоторые знаки (подчеркивания, например).
- Оно должно начинаться с буквы.
- Не допускается использование некоторых знаков (например: плюс, минус, апостроф, кавычки).
- Имя класса не должно совпадать (и, по возможности, не должно начинаться) служебным словом (например: class, System, public, int).

Правильные имена класса:	**Неправильные имена класса:**
Rabota	1zadanie
Home_work1	Petr'ivanov
zada4a_N45	Sergey-rabota7
artem_i_olga	Andrej y Taras

Хотелось бы обратить внимание на следующую рекомендацию: имя файла, в котором будет (при практической работе) записан класс, должно совпадать с именем класса. Несоблюдение этого правила (при практической работе) может привести к возникновению ошибок. Одна из проблем связана именно с тем, что «может привести»: может, но не всегда приводит. Это иногда создает ощущение, что соблюдать правило «имя класса = имя файла» не обязательно и в дальнейшем вызвать совершенно ненужные проблемы в практической работе.

После имени класса записывается фигурная скобка, открывающая «тело класса», то есть ту часть кода, которая относится к данному классу. Этой открывающей скобке соответствует закрывающая фигурная скобка (точно так же, как и в математике).

Мы рекомендуем записывать скобку, открывающую тело класса, в отдельной строке для наглядности. Однако это требование не является обязательным и существует достаточно много литературы, в которой авторы предпочитают записывать фигурную скобку, открывающую тело класса, в

той же строке кода, где записан заголовок класса. Мы считаем, что каждый может выбрать ту манеру записи, которая ему будет наиболее удобна.

public static void main(String[] args)

Каждый класс состоит из *методов*, а каждый метод состоит (подобно, кстати, классу) из *заголовка метода* и *тела метода*. Число методов в классе не ограничено, но обязательным является наличие в каждом классе как минимум одного метода, и этот метод называется главным.

Вот заголовок этого, главного метода, и находится перед нами.

Разумеется, каждое служебное слово в заголовке главного метода имеет свое описание и объяснение, так же, как имеет объяснение то, почему заголовок главного метода класса на языке Java выглядит именно так, а не иначе. Однако объяснение и обоснование этого требует использование таких терминов и понятий, которые на данный момент вряд ли окажутся понятными. В результате вместо убедительного объяснения, делающего ясным ответ на вопрос «почему заголовок главного метода выглядит так и, почему именно так?» — мы окажемся в ситуации, когда все станет еще более запутанным и еще менее понятным.

Именно поэтому мы на этом этапе ограничимся простым подходом: «заголовок главного метода должен выглядеть именно так».

Следует обратить внимание на то, что слово **String** в заголовке обязательно пишется с большой буквой **S**, а все остальные слова пишутся только с использованием маленьких букв.

Две фигурные скобки после заголовка главного метода указывают на то, где расположено *тело главного метода* — то есть та часть класса, которая автоматически начинает исполняться при запуске класса.

Таким образом, минимальный класс содержит заголовок класса и тело класса, а тело класса содержит заголовок главного метода и тело главного метода. В дальнейшем полезно называть этот пример *базовым классом*, подчеркивая, что меньше, чем содержит он, не может содержать ни один класс на языке Java.

Поскольку совершенно необходимой частью курса является практическая работа и в классе, и дома, то есть написание, отладка и исполнение классов непосредственно на компьютере, то будет очень удобно использовать разобранный выше пример в качестве «заготовки». То есть иметь на диске файл, в котором записан минимальный класс. Этот файл всегда можно загрузить, скопировать из него минимальный код (напомним, что меньше него класс не может содержать) и использовать его для написания более сложного класса.

Правда, полезным оказывается и написание каждого нового класса «с нуля»: в этом случае вырабатывается прочный навык правильной записи основных строчек кода для любого класса.

Каждый, видимо, выберет для себя тот способ практической работы, который будет ему наиболее удобен и понятен.

Значение комментариев при написании программы

Комментарии не являются обязательной частью класса, но крайне желательно приучить себя к их написанию.

Комментарий помогает не только вспомнить, для чего написан тот или иной элемент класса (команда или оператор, переменная, выражение для вычисления значения и так далее), но и способствует выработке общего логического мышления.

Комментарий — это неисполняемая часть класса, что позволяет записывать в нем свободный текст и на любом языке.

Комментарий, распространяющийся на одну (только на одну!) строку, начинается с двойного знака «наклонная черта» (слеш) и может быть вставлен в любом месте класса и в любом месте строки. Действие такого комментария распространяется до конца строки. Иными словами, после такого комментария нет смысла писать какие-то операторы, например, потому что они воспринимаются языком как часть комментария, то есть не исполняются!

Комментарий, распространяющийся на несколько строк, начинается со знаков «наклонная черта и звездочка» и заканчивается знаками «звездочка и наклонная черта».

Все строки и вообще все между этими парами знаков — комментарии, то есть содержат неисполняемые (игнорируемые) классом тексты.

```
import java.util.*;// Добавка для работы с вводом с клавиатуры
class eggs
/* Этот класс решает задачу о количестве упаковок,
которые надо заказать в упаковочный цех,
если известно, сколько всего  яиц следует упаковать
*/
{
static Scanner reader=new Scanner(System.in);
public static void main(String[] args)
{
```

```
int eggs,ostatok,upak; // имена переменных
eggs=reader.nextInt();
ostatok=eggs%12; // сколько яиц останется неупакованными в коробки
upak=eggs/12;
if(ostatok>5)
  {
        upak++;
        ostatok=0;
  }
//Начинаем вывод результатов на экран
System.out.println("Всего упаковок="+upak);
if(ostatok<=5) System.out.println("назад на склад="+ostatok);
}
}
```

Обратите внимание, что в примере класса мы умышленно использовали «чисто английские» имена (**eggs**), и имена, у которых при английском написании, как того требует язык Java, «звучание» чисто русское (**ostatok, upak**). Это не совсем правильно, хотя, разумеется, и не имеет принципиального значения; все же желательно определиться с одинаковой манерой выбора имен переменных, хотя бы в рамках каждого отдельно взятого класса.

Приведем несколько примеров для возможных имен класса и укажем, какие из них правильные, а какие — ошибочные; обоснуем каждый ответ.

Имя класса	**Возможный вариант ответа**
Serega	Верное имя — содержит одно слово, на первом месте буква, не похоже на служебное слово
Work=22	Неверное — содержит математический знак (=)
class	Неверное — совпадает со служебным словом (class)
dom zadanie	Неверное — содержит знак «пробел», то есть состоит из двух слов
Peter_and_Alex	Верное — содержит только одно слово, потому что знак «подчеркивание» (_) не разделяет слова
treugolniki_12	Верное — содержит только одно слово, потому что знак «подчеркивание» (_) не разделяет слова
treugolniki-12	Неверное — содержит математический знак (-)

Еще несколько примеров: текст базового класса (коды) — и указание на содержащиеся в нем ошибки.

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

}
}
```

Отсутствует скобка, открывающая тело класса.

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

{
}
```

Перепутаны местами скобки, которые должны указывать на расположение тела главного метода.

```
class zada4a
{
public static void main(string[] args)
{

}
}
```

В заголовке главного метода служебное слово String написано с маленькой буквы.

Этапы работы: написание, отладка, прогон

Создание класса на практике, то есть в виде файла на компьютере, состоит из трех этапов.

Первый этап — написание. Он является фактически свободной работой в специализированном текстовом редакторе. На этом этапе никогда не появится никаких сообщений ни о каких ошибках. Иными словами, программа-редактор принимает какой угодно текст, совершенно не реагируя на него, потому что для редактора это просто набор символов, а вовсе никакой не класс на языке Java.

С методической точки зрения этот этап — весьма «опасен», так как слишком прост: ученик может писать все, что он считает нужным и правильным, не получая никаких «сигналов» о допущенных ошибках.

Разумеется, существуют программы-редакторы для профессионалов, в которых все происходит совершенно иначе: в процессе написания кода появляются варианты-подсказки для кода, и даже проверка кода осуществляется практически на этапе его написания. Однако использовать такого рода программы-редакторы еще «опаснее», потому что одна из целей курса заключается в том, чтобы научится полностью осмысленно самому писать коды.

Второй этап — отладка написанного кода. В профессиональных терминах этот этап называется компилированием. На этом этапе программа-редактор, в которой пишется класс, обращается к механизмам языка Java, и проверяет правильность кода с точки зрения правил языка.

При обнаружении ошибок появляется сообщение о характере обнаруженной ошибки, и указывается предполагаемое место в программе, где эта ошибка содержится.

Следует обращать внимание на следующий факт: встроенный механизм проверки правописания (компилятор) не всегда обнаруживает действительную ошибку — иногда он указывает на ошибку мнимую, на самом деле не сделанную, и в том месте класса, где ее вовсе нет!

Это происходит из-за того, что ошибка в одном месте класса может вызывать указание на якобы сделанную ошибку в другом месте — из-за сложности правил и невозможности предусмотреть все возможные варианты ошибок.

После исправления сделанной ошибки режим отладки (компиляции) необходимо запустить снова, причем повторять этот этап до появления сообщения об отсутствии ошибок.

Важно понимать, что отсутствие ошибок с точки зрения правил языка Java вовсе не обязательно означает, что класс будет работать правильно! Например, программа для вычисления суммы двух значений с точки зрения компилятора является верной, даже если в ней на самом деле вычисляется разность этих значений.

Еще один важный момент: именно при первой компиляции происходит запись текста класса (кодов) в файл — и именно здесь следует обращать внимание на необходимость соблюдать правило «имя класса = имя файла».

Третий этап — прогон класса, его исполнение. На этом этапе главной сложностью является выявление ошибок в логике программы (программа вычисляет разность чисел вместо вычисления суммы), которые не являются ошибками с точки зрения правил языка Java. Строго говоря, именно этот этап и есть самый главный и самый сложный в работе с классом, по-

тому что выявить логические ошибки («все работает — но результаты не соответствуют тому, что должно быть») всегда чрезвычайно сложно.

Именно на этом этапе важно «выловить» наиболее сложные логические ошибки, то есть те, которые возникают только при определенном наборе значений, а иногда и вообще при одном единственном значении. Ведь правильный класс — это такой, который правильно работает всегда.

Например, в ситуации, когда класс на вводимые положительные числа реагирует сообщением: **Это положительное значение**; а на отрицательные — сообщением: **Это отрицательное значение**; важно проверить, как класс реагирует на значение 0!

В большинстве случаев процесс прогона класса снова и снова, на разных вариантах значений, является наиболее длительным и сложным этапом работы над созданием правильного класса.

Во многих случаях (строго говоря — всегда!), особенно в случаях, когда пишется сложный класс, правильно еще до его написания в виде кодов составить список возможных вариантов его будущей работы:

- какие данные следует использовать, чтобы проверить все(!) варианты работы класса;
- какие результаты должен давать класс (в частности, в виде вывода информации разного рода и вида на экран) в качестве реакции на эти данные;
- какими способами и инструментами следует воспользоваться для проверки результатов работы класса;
- какие инструменты следует задействовать на промежуточных этапах — а затем убрать из класса...

Очень важно представлять, причем с полной точностью, как должен работать класс — особенно еще до того, как вы начнете писать коды.

Мы возьмем на себя смелость утверждать (и это подтверждается не только теорией, но и результатами практической работы), что компьютерные науки являются одним из немногих (чтобы не сказать — единственным) предметов, в которых ученик не только решает задачу, но и формулирует ее (что заведомо сложнее), ставит эксперимент и оценивает его результаты.

О программах-редакторах для практической работы

Существует около десятка бесплатных программ-редакторов, то есть программ, при помощи которых можно писать, отлаживать и исполнять классы на языке Java в рамках практической работы. Кроме них, имеются и

платные программы-редакторы, используемые, как правило, профессионалами.

Наш курс не связан с точки зрения своего содержания ни с одним конкретным редактором и приводимые в курсе примеры, а также предлагаемые для курса задания, можно в рамках практической работы реализовывать на любой программе-редакторе.

В то же время мы, на основании нашего собственного опыта практического преподавания курса в рамках средней школы, считаем правильным поделиться несколькими соображениями.

Наш опыт свидетельствует: следует выбирать максимально простую в работе и установке программу-редактор, поскольку в рамках школы абсолютное большинство мощных возможностей программ для профессиональных разработчиков классов на языке Java — не востребованы.

Программа должна быть максимально понятной и максимально простой с точки зрения своего интерфейса — и при этом обеспечивать использование и работу всех необходимых инструментов языка.

На наш взгляд, такими программами-редакторами являются, в первую очередь, DrJava и JCreator: они невелики по размеру, легко устанавливаются, бесплатны и работают надежно на практически всех операционных системах, и даже на не очень мощных компьютерах.

Следует иметь в виду, что оба они не имеют русского меню, как, впрочем, не имеют они меню и на других языках; все — только на английском. Трудно сказать, что это — серьезный недостаток, в конце-то концов, для работы с программой-редактором требуется знание не более десятка английских терминов...

Гораздо более серьезной проблемой является понимание сообщений об ошибках, которые программа-редактор (точнее — язык Java...) выдает на этапе отладки класса или на этапе его исполнения. И тут тоже все только на английском, поэтому очень важно создать список наиболее типичных сообщений и использовать его в качестве справочного материала.

Хотим высказать еще несколько соображений относительно выбора редактора (среды программирования). Профессиональные оболочки типа Eclipse или Visual Studio для каждого задания (в том числе — самого простого, начального, минимального) требуют создания проекта, включающего в себя одну или несколько папок, набор файлов и прочего. Разумеется, что все это продиктовано правилами и необходимостями в работе профессионального программиста, но серьезно затрудняет (на первом этапе) запуск и отладку программ.

В то же время такая программа-редактор, например, как Dr.Java, позволяет работать по принципу «одна программа — один файл».

Кроме того, отсутствие подсветки синтаксиса, необходимость полного ввода команд, хотя и приводит на первом этапе к большому количеству чисто синтаксических ошибок, зато позволяет лучше понимать структуру программ, а также быстро овладеть минимальными навыками работы на английской клавиатуре, быстрее и надежнее запоминать требования и правила языка.

Раздел 3

Переменные и основные типы переменных. Объявление и инициализация переменных

Переменные — основной инструмент хранения *значений* в классах (программах). Переменная имеет *имя* и содержит некоторое *значение*. Поскольку круг задач, для решения которых пишутся классы, необычайно широк, в языке Java имеется несколько *типов* переменных.

Типы переменных (значений)

Каждый тип переменных определяется *служебным словом*, закрепленным за этим типом. Область значений в каждом типе ограничена и следует обращать внимание учеников на эти ограничения.

В языке есть много типов, но на первом, начальном этапе, совершенно не нужно использовать все существующие, поскольку с большинством из них работать все равно не придется. Безусловно, необходимо знать об их существовании, к счастью, нужную информацию сегодня, в эпоху всемирной информационной сети, не так уж и сложно найти в книгах или Интернет.

Мы приводим следующую небольшую таблицу, в которой указаны некоторые основные характеристики (особенности) каждого типа значений, используемые в языке Java. Следует иметь в виду, что речь идет о так называемых базовых типах, и, что язык Java предоставляет возможности для составления сложных типов значений.

Тип	Пояснение	Минимальное значение	Максимальное значение
int	«Простое» целое	–2147483648	2147483647
long	«Длинное» целое	–9223372036854775808	9223372036854775807
float	«Простое» десятичное	**3,4e-38**	**3,4e38**

Окончание таблицы

Тип	Пояснение	Минимальное значение	Максимальное значение
double	«Длинное» десятичное	**1,7e-308**	**1,7e308**
char	Отдельный символ (знак)	Пустое	Один знак из набора Unicode
String	Строка (набор символов)	Пустое	Стандартно принято говорить о максимальном размере в 256 знаков. Однако в современных языках программирования это ограничение скорее теоретическое, так как обычно размер строки хранится в 32-битовом поле, что дает максимальный размер строки в 4 294 967 295 байт (4 гигабайта).

Важно обратить внимание учеников на то, что, кроме привычных, естественных, хорошо знакомых из школьной математики, «числовых» типов (int, long, float, double) в языке Java существуют «нечисловые» типы (char, String). Специфика нечисловых типов в первую очередь в том, что с ними нельзя выполнять математические действия, во всяком случае, не так, как принято выполнять эти действия в математике. Но, разумеется, с этими «нечисловыми» типами можно выполнять иные действия, разрешенные в рамках языка Java, и специфические именно для этих типов значений.

Имя переменной

Имя переменной можно давать либо в «математической манере» (a, b, x, y — и так далее), либо «с намеком» (storona, diagonal, ves, vozrast) — то есть так, чтобы само имя с большей или меньшей степенью понятности указывало на смысл и цель его использования в классе (программе).

Сложно порекомендовать, какую манеру для выбора имен переменных следует использовать, и какую следует считать лучшей или правильной, —

в конце концов, это очень индивидуально и зависит скорее от того, какая манера близка именно тому, кто пишет класс.

Правильное имя переменной определяется практически теми же требованиями, какие предъявляются к правильному имени класса (о чем более подробно говорилось ранее). Мы советуем еще раз перечитать эти правила и рекомендации.

Важно помнить: язык Java различает имена с разными размерами букв — имена **x** (маленькая буква) и **X** (большая, заглавная буква) являются **разными** именами!

Кроме того, существуют в мире компьютерных наук неписанные и необязательное, но общепринятые соглашения: если имя переменной «состоит» из двух слов, то первое слово пишется со строчной буквы, а все остальные — с заглавной. Например, sumOfPositiveNumbers или nomerPassporta.

Объявление переменной

Объявление переменной — это указание типа, к которому переменная относится и имени этой переменной. Существует несколько правил, требований и возможностей для объявления переменной.

- В одном типе можно указывать несколько имен, разделяя их запятой, — то есть записывать список имен. Разумеется, что все эти переменные будут одного и того же типа.
- Нельзя в классе дважды указывать одно и то же имя, — ни в одном типе, ни в разных типах.
- Нельзя дважды указывать тип одной и той же переменной.
- Объявлять переменную можно практически в любом месте класса (программы), но при условии, что это не противоречит другим правилам (нельзя использовать переменную **до** ее объявления).

Относительно последнего правила следует сказать, что большинство автором и авторитетов рекомендуют собирать объявления переменных в начальных строках главного метода, несмотря на то, что можно описывать переменную практически в любой строке. Такая рекомендация связана с мнением, что подобная манера способствует большей организованности мышления, что крайне важно в компьютерных науках.

Следует, однако, иметь в виду, что существует и другая точка зрения: объявлять переменные в теле класса тогда, когда в них появляется нужда. Сторонники такого подхода считают, что он позволяет лучше структурировать программу и иногда очень значительно экономить выделяемую память.

Инициализация переменной

Инициализация переменной — это присваивание переменной значения, которое в дальнейшем она будет хранить, и которое можно будет использовать путем обращения к имени переменной.

Инициализация может производиться **одновременно** с объявлением, а может отдельно от него. Во втором случае (отдельно) инициализация переменной **всегда** должна происходить **после** объявления переменной.

Инициализация может быть выполнена *присвоением* переменной значения, — для этого используется знак равенства (=) между именем переменной и присваиваемым ей значением.

Инициализация может быть выполнена за счет использования оператора *ввода данных с клавиатуры*, а также за счет использования математических выражений или других действий, допустимых в языке Java.

Примеры объявления переменных (без инициализации!):

```
int x;
String string1;
int x1, x2, x3;
char znak;
double inputValue;
```

Примеры объявления переменной с последующей отдельной(!) инициализацией:

```
int x;
x=-6;
String string1;
int x1, x2, x3;
string1="Yes";
char znak;
x1=6, x2=0;
double inputValue;
inputValue=5.76, x3=8;
znak='7';
```

Примеры объявления переменной с одновременной инициализацией: то есть, когда и объявление, и инициализация выполняются в одной строке кода:

```
int x=5;
String string1="no, no, no!!!";
int x1=7, x2, x3=-678;
```

Примеры объявления переменной с инициализацией через дополнительные возможности языка Java, в частности, через использование правила «сначала вычисли значение, а потом присвой его переменной»:

```
int x1=6, x2, x3;
x2=10+x2;
x3=-x2;
```

Следует отметить, что возможны и более «экзотические» инициализации типа:

x1=x2=x3=0;

и даже

x1=x2=x3=in.nextInt();

На наш взгляд, не следует увлекаться подобными «сокращениями», особенно на первом этапе знакомства и освоения языка Java.

Очень важно обратить внимание на специфику инициализации переменных типа **char** и **String** (напомним, что эти типы являются нечисловыми):

- Переменные типа **char** инициализируются значениями, заключенными между двумя знаками «апостроф».
- Переменные типа **String** инициализируются значениями, заключенными между двумя знаками «кавычки».

Следует обратить внимание, что «законченное предложение» в языке Java завершается знаком «точка с запятой» (;), но более подробно правило использования этого знака будет рассмотрено уже в будущих разделах учебника.

Мы не сомневаемся, что каждому будет понятно и без особых дополнительных объяснений, по каким правилам можно использовать математические операторы «сложение» (+), «вычитание» (-), «умножение» (*).

Но **ни в коем случае** не спешите использовать оператор «деление» (/) — дело в том, что это действие в языке Java **принципиально** отличается от принятого в математике!

Преждевременное использование этого оператора может поставить неопытного пользователя в тупик. Правильному пониманию того, как этот оператор работает, отведено важное место в будущих темах, и ему посвящено много заданий для практической работы.

И уж, тем более, нет никакого смысла «изобретать» математические операторы, потому что со временем большую часть из них мы опишем и разберем в будущих разделах курса.

Приведем несколько примеров объявления переменных — с пояснениями, какие из них правильные, а какие — ошибочные.

Объявление переменных	Варианты ответа ученика
int x;	Нет ошибок
double a,b,c;	Нет ошибок
int a,a;	Ошибка: дважды используется одно и то же имя переменной
Int a,b;	Ошибка: тип переменной указан с большой буквой (правильно — int).
int a, A;	Нет ошибки в имени — большая и маленькая буквы считаются **разными** переменными (хотя это крайне нежелательно!)
String kukla1; kukla2;	Ошибка: при перечислении имен в списке используется разделительный знак «запятая» (,), а не знак «точка с запятой» (;)
char znak1,znak2,znak_1;	Нет ошибок
int x, a, b; double y ,z, x;	Ошибка: одно и то же имя (x) используется дважды в объявлении переменных
int x; double X;	Нет ошибок

И еще примеры объявления переменных вместе с их инициализацией — с указанием, какие из них правильные, а какие — ошибочные; и то же — с обоснованием ответов.

Объявление и инициализация переменных	Варианты ответа ученика
int x; x=6;	Нет ошибок
int a=6, b=2.5;	Ошибка: переменная типа «простое целое» получает значение десятичной дроби

int b=3;
double c;
c=5;
	Нет ошибок

Окончание таблицы

Объявление и инициализация переменных	Варианты ответа ученика

int a,b,c=5;
b=5;
a=-b;
	Нет ошибок

double a=3.6,b; int b=4;	Ошибка: используется дважды имя переменной

А вот пример «обратного подхода»: даны словесные описания объявления и инициализации переменных, которые затем записаны в правилах языка Java — причем в разных допустимых вариантах.

Словесное описание	Варианты ответа ученика

Переменная x (типа «простое целое») получает значение 16	int x;
	x=16;
	
	int x=16;

Переменные x и t относятся к типу «простое целое», при этом значение переменной x равно -2	int x,t;
	x=-2;
	
	int x=-2,t;
	
	int t, x=-2;

И еще два примера: даны словесные требования к классу, то есть его описание, — а надо сделать его в кодах языка Java.

Пример первый: напишите класс, который будет присваивать значения 5 и 9 двум переменных, а затем третья переменная будет получать значение, равное сумме значений первых двух переменных.

```
class primer1
{
 public static void main(String [] args)
 {
  int x,y,summa;
  x=5;
  y=9;
  summa=x+y;
 }
}
```

Пример второй: напишите класс, который будет присваивать значение 7.5 переменной **x**, а затем переменным **a** и **b** будет присваивать значения вдвое и второе (соответственно) большие значения переменной **x**.

```
class primer2
{
 public static void main(String [] args)
 {
  double x=7.5;
  double a=x*2;
  double b=x*3;
  }
}
```

Разумеется, что каждый из этих двух классов можно написать немного иначе, с точки зрения того как объявлять и инициализировать переменные. Предлагаем сделать это самостоятельно.

Раздел 4
Немного про символьные и строковые типы

Символьный и строковый типы не являются по-настоящему важными для первого года обучения типами данных. Их использование в задачах этого года имеет очень ограниченный характер, и акцентировать внимание на работе с переменными таких типов не имеет большого смысла. Кроме того, более широкое использование строковых типов планируется на втором году обучения, — да и сам этот тип гораздо сложнее и «хитроумнее», чем стандартные числовые типы.

В силу этих причин ученику в первый год обучения достаточно знать в отношении этих типов следующие вещи:

- Как присваивается значение переменным этих типов.
- Как вводятся значения для переменных этого типа с клавиатуры.
- Какие основные и принципиальные отличия этих типов от стандартных числовых типов.
- Некоторые основные действия, которые можно выполнять в отношении этих типов.

Строковые и символьные типы не предназначены для выполнения математических действий, так как не содержат (в явном виде) числовых значений.

Даже если переменные описаны и инициализированы следующим образом:

```
char a='4';
String b="12";
```

и выглядят как числа, — это совершенно не гарантирует возможность выполнять с ними математические действия. При этом с переменной строкового типа (String) это проявляется в гораздо более четко выраженной форме, а вот переменные символьного типа (char) могут в определенных ситуациях «вести себя» так, как будто они содержат числовое значение. При этом «как будто» проявляется в том, что числовое значение совершенно не совпадает с той «как бы цифрой», которую мы присваиваем переменной типа char.

К правилам работы со строковыми и символьными переменным мы вернемся через тему — после того, как разберемся с тем, как инициализировать переменные не через написание значений в кодах, а с помощью ввода информации с клавиатуры.

Раздел 5
Команды вывода и ввода информации

Операторы вывода и ввода информации необходимы, для того чтобы превратить класс в реально интерактивный продукт, осуществляющий диалог с пользователем и подстраивающий себя (в определенных рамках) к его потребностям.

В языке Java существует довольно много возможностей ввода и вывода информации, и в рамках нашего курса мы ограничиваемся лишь несколькими из них, — впрочем, и их оказывается более чем достаточно.

Команды вывода информации

Стандартно вывод информации осуществляется на экран. Для этого используются две команды:

System.out.print() и **System.out.println()**

Во второй из этих команд имеется добавка: ln — сокращение английского слова line.

Первая из этих двух команд, выводя информацию на экран, запоминает ту позицию на экране, в которой было выведено (напечатано) самое последнее значение, — и в следующий раз информация будет выводиться на экран, начиная с позиции, «прилепленной вплотную» к запомненной.

Вторая команда (с ln в конце) — после вывода (печати) информации на экран; в использованной для этого экранной строке больше никакой информации выводиться не будет. Место вывода информации в будущий раз автоматически переводится на начало следующей строки экрана, то есть следующий вывод информации на экран начинается «с красной строки».

Это значит, что две команды:

```
System.out.print("Ok");
System.out.print(23);
```

выведут на экран **Ok23**. Кроме того, и следующая команда вывода на экран начнет размещать информацию вплотную к цифре **3**.

А вот две команды:

```
System.out.println("Ok");
System.out.println(23);
```

выведут на экран ту же самую информацию, что и в предыдущем примере, но располагаться эта информация на экране будет иначе.

А именно:

Ok
23

Кроме того, при этом и следующая команда вывода на экран начнет размещать информацию «с красной строки».

В скобках команды «вывести на экран» указывается, что́ выводить на экран.

Информация без кавычек рассматривается как «вывести значение». Например, команда **System.out.println(x)** выведет на экран значение переменной **x**. Разумеется, эта переменная должна получить значение **до того**, как она будет использована в данной команде. Например, если переменная **x** имеет значение, равное 4, то приведенная только что в качестве примера команда будет выводить на экран 4 и переходить на следующую экранную строку.

Важно обратить внимание на то, что в языке Java практически **нет** присваивания переменным **значений по умолчанию**, то есть автоматически. В частности, переменные типа «простое целое число» не получают автоматически значения 0 (ноль) — как это происходит в некоторых других языках.

А это означает, что использование команды вывода с переменной, которая не инициализирована, приводит к сообщению об ошибке. Так, например, при отладке (компиляции) класса:

```
class primer3
{
 public static void main(String[]args){
  int a;
  System.out.println(a);

 }
}
```

возникает сообщение об ошибке:

Error: The local variable a may not have been initialized

Смысл этого сообщения в том, что переменная не инициализирована — ну и, соответственно, класс не имеет возможности вывести на экран ее значение, так как никакого значения переменной не присвоено.

Информация, записанная в команде вывода на экран, внутри скобок, в кавычках, выводится на экран точно в таком же виде, в каком она написана в кавычках («как записано, так и печатается»). Например, команда **System.out.println("x")** выведет на экран **x** — запись **x** в кавычках означает, что язык не воспринимает **x** как имя переменной, а как «просто букву».

Знак «плюс» в командах вывода на экран (+) имеет два смысла и, соответственно, означает совершенно разные действия:

- указание на выполнение арифметического действия;
- Указание на «приклеивание рядом» двух разных видов информации (вне кавычек и в кавычках).

Как видите, разница между тем, к каким результатам приводит использование в команде вывода знака «плюс» (+), — просто огромная. Строго говоря, это даже не разница в обычном смысле: между знаком «плюс», означающим математическое действие сложения, и между знаком «плюс», означающим операция «приклеивания», просто нет ничего общего. Невнимательность к использованию этого, казалось бы, совершенно простого знака действия, непонимание разницы между действиями, которые этот знак обозначает, нередко приводят к результатам, которые автор класса совершенно не рассчитывал получить.

Таблица примеров

(везде значение переменной **x** равно 4, а значение переменной **y** равно 7)

Команда	**Как выглядит вывод на экран**
System.out.println(x)	4
System.out.println("x=")	x=
System.out.println("x="+x)	x=4
System.out.println(x+"x=")	4x=
System.out.println(x+y)	11
System.out.println("x+y")	x+y
System.out.println("x+y"+(x+y))	x+y11
System.out.println("x="+x+" y="+y)	x=4 y=7
System.out.println(x+"+"+y+"="+(x+y))	4+7=11

Наш опыт работы однозначно свидетельствует: очень важно на самом начальном этапе практической работы по написанию классов на языке Java делать как можно большее число несложных задач на вывод одной и той же информации в самых разных комбинациях. Компьютеру, разумеется, совершенно безразлично, в каком виде информация выводится на экран, и выводится ли вообще. Но для формирования навыков написания «дружественных к пользователю» классов это очень важно.

Команды ввода данных с клавиатуры

В языке Java существует мощная и разветвленная система ввода и вывода данных, частным случаем которой является ввод данных с клавиатуры, который является частным случаем обработки общего потока данных.

Для того чтобы в рамках данного курса не вносить ненужное и, что гораздо важнее, не принципиальное многообразие возможных в языке Java вариантов ввода данных, мы остановились только на одном из них. Выбранный нами метод, с одной стороны, минимален в смысле требований к знанию дополнительных возможностей, а с другой — надежно гарантирует беспроблемный и простой ввод данных с клавиатуры.

В языке Java не выполняется ввод данных **напрямую** с клавиатуры в переменные. Команда ввода данных с клавиатуры работает по принципу системы заказа такси по телефону, то есть по системе «клиенты — диспетчер — таксисты». Применимо к языку Java эта система («клиенты — диспетчер — таксисты») означает, что при вводе данных с клавиатуры класс в обязательном порядке должен использовать специальную переменную-«посредника» (между клавиатурой и переменными).

Система, которую мы привели в качестве примера («клиенты — диспетчер — таксисты»), в применении к языку Java, распределяет «обязанности» следующим образом:

клиенты = значения, вводимые с клавиатуры;
диспетчер = отдельная переменная специального типа — типа Scanner;
таксисты = переменные, используемые в классе.

А требования эти в языке Java приводят, в свою очередь, к тому, что во всех классах, где будет использован ввод данных с клавиатуры, следует добавить две строки.

Первая такая дополнительная строка:

import java.util.*;

Она должна быть записана **до**(!) заголовка класса.

Вторая такая дополнительная строка:

static Scanner reader = new Scanner(System.in);

Она должна быть записана в теле класса **перед** заголовком главного метода. Следует иметь в виду, что в несколько ином виде эта строка может быть записана в других частях класса.

Строка
static Scanner reader=new Scanner(System.in);
объявляет, что все данные всех типов с клавиатуры будут вводиться в программу и передаваться соответствующим переменным только через переменную- «диспетчер», для которой выбрано имя reader. Разумеется, этой переменной можно дать и любое другое имя — в рамках принятых в языке Java правил.

В этом случае «базовый класс» будет выглядеть так:

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

        }
}
```

Кроме того, в языке Java операторы ввода данных, то есть значений, для разных типов значений (и, соответственно, для переменных разных типов) выглядят по-разному.

Таблица ввода данных для переменных разных типов
(в приведенных примерах предполагается, что ввод данных с клавиатуры осуществляется через переменную с именем reader)

Тип данных (тип переменной)	**Команда ввода (через переменную reader)**	**Пример команды ввода**
int	reader.nextInt()	a=reader.nextInt()
long	reader.nextLong()	b=reader.nextLong()
float	reader.nextFloat()	c=reader.nextFloat()
double	reader.nextDouble()	d=reader.nextDouble()
String	reader.next()	name=reader.next()
char	reader.next()charAt(0)	Znak=reader.next()charAt(0)

Например, класс, который принимает с клавиатуры целое число и выводит на экран его квадрат, выглядит, скажем, следующим образом:

```
import java.util.*;
class vvod_vyvod
{
 static Scanner reader =  new Scanner(System.in);
 public static void main(String[] args)
 {
        int a,b;
        System.out.print("Введите целое число=");
        a=reader.nextInt();
        b=a*a;
        System.out.println("Квадрат введенного числа="+b);
 }
}
```

Возвращение и дополнение к разделу 4 — о строковых и символьных переменных

Ввод строкового значения с клавиатуры осуществляется с помощью оператора **reader.next()** — если исходить из предположения, что переменная для ввода данных имеет имя **reader**.

Ввод символьного значения — с помощью оператора

reader.next().charAt().

При этом в операторе **reader.next().charAt()** во вторых скобках надо указывать, какой по счету символ из набираемого на клавиатуре будет вводиться в переменную. Например, оператор **reader.next().charAt(1)** принимает второй по счету символ из того строкового значения, которое будет набрано на клавиатуре перед нажатием на клавишу «Ввод» (Enter).

Чрезвычайно важно обратить внимание на то, что нумерация символов в строковом значении начинается с 0! Это следует постоянно иметь в виду, поскольку более «естественным», более привычным, более распространенным в повседневной жизни является счет не с 0, а с 1 (единицы).

Например, класс

```
import java.util.*;
class primer_str
{
static Scanner reader =  new Scanner(System.in);
```

```
public static void main(String[] args)
{
        String a=reader.next();
        char b=reader.next().charAt(1);
        System.out.print(a+" "+b);
}
}
```

исполняется следующим образом:

- сначала класс принимает с клавиатуры одно строковое значение, которое заносится в переменную **a**;
- затем класс позволяет набрать на клавиатуре еще одно строковое значение, из которого в качестве значения для символьной переменной **b** выбирается только один — второй по счету — символ (его порядковый номер — 1);
- После этого оба значения выводятся на экран в одну строку, через пробел.

Это означает, что если с клавиатуры было дважды введено значение **qwerty**, то на экран будет выведено:

qwerty w

Хотим обратить внимание вот на какую особенность ввода с клавиатуры строковых значений: использование команды ввода **s=reader.next()** приводит к тому, что переменную **s** вводится только часть строки (до первого пробела или табуляции). Если же по условию задачи необходимо ввести все символы, в том числе и пробел (например, если целью задачи является подсчет слов в введенном предложении), то можно применить другую, аналогичную использованной нами, команду **s=reader.nextLine()**.

Некоторые основные действия и операции с символьными и строковыми переменными

Несмотря на то, что речь сейчас идет о том, как выполняется в языке java операция выбора, мы сделаем небольшое отступление «вперед» для символьных переменных.

Сравнение переменных символьного (char) типа можно выполнять практически точно так же, как и сравнение числовых переменных, например, типа **int**:

```
char simb1, simb2;
simb1=reader.next().charAt(0);
simb2=reader.next().charAt(0);
if (simb1==simb2) System.out.println("Yes");
```

А вот сравнение переменных строкового (**String**) типа необходимо выполнять с использованием специальных методов.

Один из них возвращает значение **true**, если сравниваемые переменные содержат одинаковые значения; в ином случае этот метод возвращает значение **false**.

Приведем фрагмент класса с использованием этого метода:

```
String s, t;
s=reader.next();
t=reader.next();
if (s.equals(t)==true) System.out.println("yes");
else System.out.println("no");
```

Как видите, метод содержит имя одной переменной и после него точку и вызов самого метода (**equals**), а в скобках, в качестве параметра, указывается имя второй переменной: **s.equals(t)**

В случае если сравниваемые переменные содержат идентичные значения, метод возвращает значение **true**. Приведенный выше фрагмент можно записать и с использованием «короткой формы» условия:

```
String s, t;
s=reader.next();
t=reader.next();
if (s.equals(t)) System.out.println("yes");
else System.out.println("no");
```

Существуют и другие методы сравнения переменных строкового типа, но для данного курса описанный выше метод является вполне достаточным.

Практически необходимым является, при работе со строковыми значениями, и метод, который позволяет определить количество символов в строковой переменной.

Приведем фрагмент класса с использованием этого метода:

```
String s, t;
s=reader.next();
int long=s.length();
System.out.println("s="+s+"="+long);
```

Оператор + (плюс) в применении к переменным строкового типа означает операцию «подклеивания»: к одному строковому значению подклеивается другое строковое значение.

Приведем пример фрагмента класса:

```
String s, t;
s=reader.next();
t="One";
s=s+t;
System.out.println(s);
t="Kuku"+t;
System.out.println(t);
```

Если при исполнении этого фрагмента с клавиатуры будет введено значение qqq, то на экран будут выведены значения:

```
qqqOne
KukuOne
```

В будущих разделах курса мы разберем так называемые циклы, а здесь хотим привести пример использования цикла для работы с отдельными символами значения строковой переменной.

К приведенному ниже примеру стоит вернуться после того, как будут изучены циклы.

Для обращения к отдельным символам надо использовать метод charAt(), у которого в качестве параметра (в скобках) указывается индекс символа в строковой переменной.

Приведем пример фрагмента класса:

```
String s;
int long;
s=reader.next();
long=s.length();
for (int i=0;i<long;i++)
        System.out.print(s.charAt(i)+"#");
```

В случае если с клавиатуры будет введено значение qwerty, на экран будет выведено:

```
q#w#e#r#t#y#
```

Еще раз: обратите внимание, что нумерация символов в строковом значении начинается с 0!

Раздел 6

Присвоение данных между переменными разных типов (приведение типов, casting). Специальные операторы

В предыдущих разделах курса шла речь о том, что инициализацию переменных можно производить как прямым присваиванием значений (прямая инициализация), так и через использование уже инициализированных переменных (косвенная инициализация).

Приведем несколько примеров такой прямой и косвенной инициализации переменных:

Прямая инициализация	Косвенная инициализация
int a=4;	int a=6,b; b=a;

int b;
b=-8;
	int x,y;
	x=-8;
	b=10*a;

int x=reader.nextInt()	int x=reader.nextInt(); y=-x

Как правило, никаких проблем не возникает, когда прямая и косвенная инициализации производятся в рамках одного и того же типа переменных и типа данных. Иными словами, когда переменная типа «простое целое» (int) получает значение, лежащее в рамках диапазона этого типа (например, 8). Или — переменная типа «длинное десятичное» (double) получает, например, значение 5.7.

Думаем, что очень полезно в этом случае вновь продемонстрировать таблицу типов стандартных значений, — мы приведем ее здесь еще раз.

Тип	Пояснение	Минимальное значение	Максимальное значение
int	«Простое» целое	-2147483648	2147483647
long	«Длинное» целое	-9223372036854775808	9223372036854775807
float	«Простое» десятичное	**3,4e-38**	**3,4e38**
double	«Длинное» десятичное	**1,7e-308**	**1,7e308**
char	Отдельный символ (знак)	Пустое	Один знак
String	Строка (набор символов)	Пустое	Стандартно принято говорить о максимальном размере в 256 знаков. Однако в современных языках программирования это ограничение скорее теоретическое, так как обычно размер строки хранится в 32-битовом поле, что дает максимальный размер строки в 4 294 967 295 байт (4 гигабайта).

Полезно на практике написать (для тренировки, для выработки навыка работы) несколько классов с использованием разных вариантов прямой и косвенной инициализации переменных. Разумеется, в этих примерах должны быть такие инициализации, которые не будут вызывать конфликта между объявленными типами переменных и значениями, которые эти переменные будут получать.

В примерах достаточно ограничиться (для простоты) математическими типами из приведенной таблицы.

Перед вами пример объявления и инициализации переменной — возникает ли при этом конфликт?

```
int a = 100; long b = a;
long a=100; int b=a;
```

```
int x=21474836947;

int x=21474847;
```

Однако совершенно необязательно, чтобы к классу передача значений осуществлялась только и только между переменными одного и того же типа!

В языке Java предусмотрена возможность присвоения данных одного типа переменным другого типа. Допускается делать это и через явное использование переменной одного типа и значения другого типа, а также и через использование выражений с переменными (инициализированными, разумеется) разных типов.

При этом, разумеется, происходит (или должно быть произведено!) преобразование данных — такой процесс называется «приведением типов», (casting). Он может быть как неявным (фактически автоматически производимым самим языком), так и явным, то есть требующим написания соответствующих операторов (подобно явному использованию оператора «плюс» (+) для вычисления суммы).

Неявное приведение типов (implicit) происходит, когда значение более узкого диапазона помещается в переменную более широкого диапазона.

Можно вместо термина «диапазон» использовать популярный в математике термин «множество», и тогда только что сформулированное правило будет выглядеть так: значение из подмножества можно помещать в переменную, относящуюся к множеству, включающему подмножество.

Например, во множестве рациональных чисел есть множество целых чисел: то есть любое целое число формально можно записать как дробь, но не любую дробь можно записать как целое число.

Например, посмотрите на следующий фрагмент класса:

```
int a=4;
long b;
b=a;
```

Как видите, переменная типа «длинное целое» (long) может принимать значение типа «простое целое» (int) — это допустимо потому, что тип «простое целое» является внутренним подмножеством типа «длинное целое». Иными словами, подобное действие возможно потому, что любое значение типа «простое целое» находится внутри множества значений «длинное целое».

А вот наоборот — это не так; и именно потому, что это не так, для приведения типа «из подмножества в множество» (например, из типа long в тип int) требуются специальные, записываемые в явном виде, действия.

Явное (explicit) приведение типов требует использования специальных операторов.

Они представляют из себя описание типа, к которому (в который) надо выполнить приведение, заключенное в скобки и поставленное перед значением (или переменной), над которым (которой) надо выполнить действие приведения. Если попробовать сформулировать это правило максимально просто, то, видимо, один из возможных вариантов может выглядеть так: перед тем, **что** надо привести, следует в скобках указать тип, **к которому** надо привести.

Например: int x = (int)3957.229;

В этом примере значение множества «длинное десятичное» (десятичная дробь) заносится в переменную из «внутреннего подмножества» («простое целое», int), для чего и используется указание на явное приведение в виде заключенного в скобки типа (int).

Разумеется, что в таком случае теряется часть значения (как правило, происходит понижение степени точности) — в приведенном примере переменная x будет иметь значение 3957.

Аналогично выполняется операция приведения с переменными:

double a=4.95;
int b=(int)a;

Кстати, можно выполнять явное приведение типов и для случаев, когда в этом нет необходимости — то есть когда язык Java может произвести неявное приведение.

Например:

double a=(double)4;

или

int a=4;
double b=(double)a;

Следует обязательно иметь в виду следующее ограничение, существующее в языке Java: все дробные значение язык автоматически относит к типу **double**, поэтому запись **float a=4.95** считается ошибочной и требует выполнения явного приведения типа, то есть записи **float a=(float)4.95**

Операторы приведения типов окажутся особенно важными, когда потребуется подсчитать, например, среднее арифметическое двух целочисленных значений! Дело в том, что в этом случае мы снова сталкиваемся с

необходимостью выполнить операцию деления — а она в языке Java требует быть чрезвычайно внимательным именно из-за использования разных типов.

Математические и специальные операторы

Для выполнения основных математических действий в языке Java используются следующие операторы:

Оператор	Действие
+	Сложение двух значений
-	Вычитание из одного значения другого
*	Умножение двух значений

Разумеется, операторы эти могут использоваться как с числовыми значениями, так и с переменными — и порядок выполнения действий в сложных выражениях точно соответствует порядку выполнения действий, принятому в математике.

Аналогичным образом используются и скобки — только в языке Java, в отличие от математики, для записи сложных выражений используются **только** круглые скобки.

Гораздо сложнее обстоит дело с, казалось бы, простым оператором деления — это действие в языке Java обозначается символом «наклонная черта» (/), а также символом «процент» (%). Нередко можно встретить такие определения: оператор деления (/) и оператор вычисления остатка (%). Как видите, действие деления оказывается очень «многоликим» и далеко не всегда и в самом деле является оператором деления; все зависит от того, с переменными (и значениями) какого типа он используется!

Эта «многоликость» является источником многочисленных ошибок — в тех, разумеется, случаях, когда при написании класса забывают о необходимости точно представлять, с каким из действий, обозначаемых символом «наклонная черта», придется иметь дело.

Рассмотрим в качестве примера следующий фрагмент класса (программный блок, набор кодов, — можно использовать разные термины):

```
int a=reader.nextInt();
int b=2;
int c=a/b;
System.out.println(c);
```

Этот фрагмент будет давать для одних данных «ожидаемые» результаты, а для некоторых — вроде бы «неожиданные».

- При введении с клавиатуры (для переменной **а**) значения 4 — на экран выводится значение 2.
- При введении с клавиатуры (для переменной **а**) значения 5 — на экран выводится **тоже** значение 2.

Все дело в том, что при использовании с переменными только типа «целое число», знак «наклонная черта» (/) является оператором «вычисление частного», а вовсе не оператором «деление».

А теперь изменим этот фрагмент так, чтобы он выглядел следующим образом:

```
double a=reader.nextInt();
double b=2;
double c=a/b;
System.out.println(c);
```

Оказывается, работа, то есть исполнение фрагмента, становится заметно иным!

- При введении с клавиатуры (для переменной **а**) значения 4 — на экран выводится значение 2.0
- При введении с клавиатуры (для переменной **а**) значения 5 — на экран выводится значение 2.5

Все дело в том, что в этом случае знак «наклонная черта» (/) действительно является оператором «деление».

В случае же использование переменных и значений **разных** типов то, какое действие будет выполнять знак «наклонная черта» (/), будет зависеть от комбинации переменных и использования операторов приведения типов...

Как видите, один и тот же символ («наклонная черта») при всех своих использованиях так или иначе связан с операцией деления (ведь вычисление частного производится тоже в результате деления), однако значения, которые получаются в результате его использования, могут численно отличаться друг от друга.

Все это просто необходимо иметь в виду каждый раз, когда вы собираетесь использовать этот знак, и заранее, еще до написания кодов, четко себе представлять, какой из результатов вы хотите получить: «настоящий результат деления» или «частное от деления».

Кроме того, в языке Java имеются специальные (дополнительные) операторы для выполнения математических действий.

Оператор	Действие	Пример и объяснение
++	Увеличение значения переменной на 1	**int x=reader.nextInt();** **x++;** Если x получил с клавиатуры значение 6, то после выполнения оператора ++ переменная будет иметь значение 7
--	Уменьшение значения переменной на 1	**int x=reader.nextInt();** **x--;** Если x получил с клавиатуры значение 6, то после выполнения оператора -- переменная будет иметь значение 5
+=	Увеличение значения переменной на значение, записанное после знака =	**int x=reader.nextInt();** **x+=10;** Если x получил с клавиатуры значение 6, то после выполнения оператора +=10 переменная будет иметь значение 16
-=	Уменьшение значения переменной на значение, записанное после знака =	**int x=reader.nextInt();** **x-=10;** Если x получил с клавиатуры значение 16, то после выполнения оператора -=10 переменная будет иметь значение 6
*=	Увеличение значения переменной в число раз, равное значению, записанному после знака =	**int x=reader.nextInt();** **x*=10;** Если x получил с клавиатуры значение 6, то после выполнения оператора *=10 переменная будет иметь значение 60
/=	Деление (или вычисление частного) значения переменной на значение, записанное после знака =	**int x=reader.nextInt();** **x/=10;** Если x получил с клавиатуры значение 67, то после выполнения оператора /=10 переменная будет иметь значение 6
%=	Вычисление остатка при делении значения переменной на значение, записанное после знака =	**int x=reader.nextInt();** **x%=4;** Если x получил с клавиатуры значение 6, то после выполнения оператора *=10 переменная будет иметь значение 2

Разумеется, все эти операции можно записать в ином, без специфики языка Java, виде:

- x++ можно записать как x=x+1;
- x-- можно записать как x=x-1;
- x+=10 можно записать как x=x+10;
- x-=10 можно записать как x=x-10;
- x*=10 можно записать как x=x*10.

Следует знать оба варианта, поскольку кроме умения писать собственные коды, надо уметь и понимать коды, написанные кем-то другим. Кроме того, каждый, разумеется, может сам для себя решить, какую из форм записи он предпочтет использовать.

И хотя любой оператор с присваиванием, с точки зрения получаемого результата, эквивалентен выполнению соответствующей операции с последующим присваиванием, то работает он обычно быстрее.

В самом общем виде типичная для языка Java «сокращенная» форма записи строится следующим образом:

Существует одна особенность, связанная с использованием этих специальных операторов для изменения значения не **одной и той** же переменной, а **разных** переменных. Особенность эта связана с тем, где записывается указание на изменение — до или после переменной (пользуясь профессиональной терминологией: префиксная или постфиксная форма записи).

Вид блока команд	**Что выводится на экран?**	**Объяснение**

int a=6,b;
b=a++;
System.out.println(a+" & "+b);
	7 & 6	Во второй строке переменная **b** сначала получает от переменной **a** значение 6 — и только потом значение переменной **a** увеличивается на 1

int a=6,b;
b=++a;
System.out.println(a+" "+b);
	7 & 7	Во второй строке сначала значение переменной **a** увеличивается на 1, а только потом это увеличенное значение получает переменная **b**

Иными словами:

- команда **b=a++;** эквивалента командам **b=a; a=a+1;**
- команда **b=++a;** эквивалента командам **a=a+1; b=a;**

Это означает, что в такого рода командах чрезвычайно важным является то, записаны ли специальные операторы языка Java **до** или **после** соответствующей переменной.

Раздел 7
Вычисление частного и остатка

В одной из предыдущих тем мы уже рассматривали особенности работы в языке Java оператора, обозначаемого знаком «наклонная черта» (/). Там мы говорили о том, что этот знак может означать и операцию **вычисления частного** (при делении одного целого числа на другое), и операцию **«истинного деления»** (если используются переменные типа «десятичные дроби» — double или float).

В языке Java существует и оператор, который вычисляет математический остаток, который образуется при делении одного числа на другое. Обозначается этот оператор знаком «процент» (%).

Например, при выполнении следующего фрагмента класса (блока):

```
int a=7;
int b=5;
int c=a % b;
```

переменная **c** будет получать значение 2, — то есть остаток от деления 7 на 5.

Операторы / (вычисление частного) и **%** (вычисление остатка) связаны с несколькими чрезвычайно широкими группами задач, часть из которых происходят из «жизненной практики», а часть — из «хитрой математики».

К первой группе можно отнести, например такую задачу: *«При выезде группы школьников на экскурсию администрация должна заказать нужное число автобусов. Напишите класс, который принимает с клавиатуры количество школьников, которые должны выехать на экскурсию, и определяет число автобусов, которые следует заказать. Известно, что в каждом автобусе имеется 40 мест для учеников».*

Ко второй группе задач относятся такие, которые можно назвать «разложение целого числа на составляющие его числа или цифры». Это, например, такая задача: *«Напишите класс, который принимает с клавиатуры целое трехзначное число и определяет, является ли произведение первой и последней его цифр (числа единиц и числа сотен) больше квадрата его средней цифры (числа десятков)».*

Еще одна группа задач, в которых ключевым является определение остатка при делении, это те задачи, в которых требуется проверить, делится ли одно число на другое. Во всех этих случаях мы используем математическое правило «Число **a** делится на число **b**, если остаток при делении **a** на **b** равен нулю». А это означает необходимость вычисления остатка при делении, что и позволяет сделать оператор «процент» (%).

Для второй группы задач («разложение целого числа на составляющее») особенно значение имеет использование степеней 10, то есть значений 10, 100, 1000, 10000 и так далее. Для пояснения приведем несколько конкретных числовых примеров.

Число	**Частное от деления числа на...**					**Остаток от деления числа на...**				
	10	**100**	**1000**	**10000**	**100000**	**10**	**100**	**1000**	**10000**	**100000**
6	0	0	0	0	0	6	6	6	6	6
45	4	0	0	0	0	5	45	45	45	45
769	76	7	0	0	0	9	69	769	769	769
2618	261	26	2	0	0	8	18	618	2618	2618
69362	6936	693	69	6	0	2	62	362	9362	69362
658791	65879	6587	658	65	6	1	91	791	8791	58791
2456765	245676	2456	2457	245	24	5	65	765	6765	56765

При внимательном изучении таблицы можно сделать следующие общие выводы (они, собственно, понятны и из математики, просто мы сформулируем их в более «подходящем» для компьютерных наук виде):

- операция «вычисление частного при делении на 10» всегда «укорачивает» число на последнюю цифру;
- операция «вычисление частного при делении на 100» всегда «укорачивает» число на две последних цифры;
- операция «вычисление частного при делении на 1000» всегда «укорачивает» число на три последних цифры;
- и так далее…
- операция «вычисление остатка при делении на 10» всегда «возвращает» одну последнюю цифру;
- операция «вычисление остатка при делении на 100» всегда «возвращает» две последних цифры;
- операция «вычисление остатка при делении на 1000» всегда «возвращает» три последних цифры;
- и так далее…

Разумеется, для выделения отдельной цифры из числа в большинстве случаев надо будет использовать комбинированные операции «вычисления частного» и «вычисления остатка».

В качество дополнительного материала приведем примеры использования этих операций применительно к двузначным, трехзначным и четырехзначным числам.

Извлечение из числа его отдельных цифр и «внутренних чисел»

Серьезным ограничением в этой группе задач является необходимость заранее знать, из какого количества цифр составлено число. В дальнейшем, когда будут изучаться циклы, мы увидим, как можно не учитывать это ограничение.

Но пока все задачи мы будем строить на предположении, что число цифр во вводимых числах заранее известно и является в точности таким, как требуется по условию задачи.

Двузначное число

Первая цифра (десятки) получается вычислением частного от деления числа на 10.

Вторая цифра (единицы) получается вычислением остатка от деления числа на 10.

Приведем пример класса, который принимает с клавиатуры двузначное число и выводит его на экран (как часто принято говорить — печатает) его в «подробном виде», то есть в виде соответствующего числа десятков и соответствующего числа единиц. Это значит, например, что при вводе с клавиатуры 23 на экран выводится 23=20+3.

```
import java.util.*;
class primer11
{
        static Scanner reader=new Scanner(System.in);
        public static void main(String[] args)
        {
                int x,dig1,dig2;
                x=reader.nextInt();
                dig1=x/10;
```

```
            dig2=x%10;
            System.out.println(x+"="+dig1*10+"+"+dig2);
        }
}
```

Трехзначное число

Первая цифра (сотни) получается вычислением частного от деления числа на 100.

Вторая цифра (десятки) получается после двух вычислений. Сначала вычисляем остаток от деления на 100, в результате чего получается двузначное число, составленное из двух последних цифр исходного числа. Затем от этого полученного двузначного числа вычисляем частное от деления на 10 (как в случае двузначного числа).

Кстати, есть и другой способ вычисления средней цифры трехзначного числа (цифры десятков) — какой?

Кроме того, два вычисления, описанных выше, можно свести в одно математическое выражение с двумя последовательно выполняемыми действиями (это, кстати, и сделано в приведенном ниже классе).

Последняя цифра (единицы) получается вычислением остатка от деления на 10

Приведем пример класса, который принимает с клавиатуры трехзначное число и печатает его в «подробном виде» — аналогичном тому, как мы выше сделали с двузначным числом. Мы имеем в виду, что при вводе с клавиатуры значения 123 на экран выводится 100+20+3:

```
import java.util.*;
class primer12
{
    static Scanner reader=new Scanner(System.in);
    public static void main(String[] args)
    {
        int x,dig1,dig2,dig3;
        x=reader.nextInt();
        dig1=x/100;
        dig2=(x%100)/10;
        dig3=x%10;
        System.out.println(x+"="+dig1*100+"+"+dig2*10+"+"+dig3);
    }
}
```

Четырехзначное число

Первая цифра (тысячи) получается вычислением частного от деления числа на 1000.

Вторая цифра (сотни) вычисляется в два приема: сначала вычисляем частное от деления числа на 100, а потом для полученного двузначного числа вычисляем остаток от деления его на 10.

Разумеется, тут тоже имеется другая возможность вычисления той же цифры — предлагаем вам самим найти этот способ.

Третья цифра (десятки) вычисляется, как и вторая, в два приема: сначала вычисляем остаток от деления на 100, а потом из него вычисляем частное от деления на 10.

Разумеется, тут снова имеется другая возможность вычисления той же цифры, — и мы снова предлагаем вам самим найти этот способ.

Последняя цифра (единицы) получается вычислением остатка от деления на 10.

Приведем пример класса, который принимает с клавиатуры четырехзначное число и печатает его в «подробном виде»: например, при вводе с клавиатуры числа 5123 на экран выводится 5000+100+20+3.

```
import java.util.*;
class primer13
{
    static Scanner reader=new Scanner(System.in);
    public static void main(String[] args)
    {
        int x,dig1,dig2,dig3,dig4;
        x=reader.nextInt();
        dig1=x/1000;
        dig2=(x/100)%10;
        dig3=(x/10)%10;
        dig4=x%10;
        System.out.println(x+"="+dig1*1000+"+"+dig2*100+"+"+
        dig3*10+"+"+dig4);
    }
}
```

Разумеется, аналогичным образом можно «расчленять» числа с большим количеством цифр, можно «вычленять» из одного числа другие, «внутренние» ему числа (из трехзначного, например, двузначные), а также придумывать разные иные «игры с числами». Надо сказать, что это целый класс довольно интересных и далеко не всегда простых задач.

Раздел 8
Команда выбора (ветвления)

Возможно, перед началом знакомства с этим разделом, стоит сначала прочитать материал из раздела 18, поскольку именно там рассматриваются так называемые логические, или булевы, выражения, являющиеся неотъемлемой частью команд выбора.

Все примеры, которые мы решали и разбирали до сих пор, были построены по так называемому «линейному принципу». Этот принцип означает, что команды в классе были «выстроены в линию» и с точки зрения их расположения в теле главного метода класса («первая команда, за ней вторая, третья, четвертая...»), и в смысле их исполнения («сначала исполняется первая команда, за ней вторая, третья, четвертая...»).

«Линейный класс» построен по принципу «как пишется, так и работает»: выполняются все до одной команды и строго в том порядке, в каком они записаны в главном методе.

Однако это — совершенно не обязательное правило! Более того — в большинстве классов содержатся так называемые «команды ветвления», которые, в зависимости от данных и результатов проверки данных, направляют исполнение класса либо по одной группе команд («ветке»), либо по другой.

Поскольку то, по какой из «веток» направляется исполнение класса, зависит от проверки каких-то условий, команда ветвления часто называется еще и «условным оператором».

Есть у нее и еще одно, также популярное название — «команда выбора«: это связано с тем, что она, проверяя данные в условии, выбирает вариант исполнения («ветку»).

Вот об этой команде ветвления, она же — команда выбора, она же — условный оператор, и будет идти речь и в этой теме, и в нескольких следующих.

Следует иметь в виду, что команда выбора относится к числу наиболее часто встречающихся, наиболее широко употребляемых инструментов языка Java (да и вообще практически любого языка — невозможно себе представить, что нет в языке необходимости проверки данных).

Это — одна из самых важных команд в любом языке, потому что с ее помощью становится возможным «различать» разные значения, разные соотношения между значениями, направлять исполнение программы по разным ветвям.

Важно начать изучение команды выбора с самых простых вариантов, постепенно усложняя возможности записи и исполнения этой команды. Дело тут в том, что «мощность» команды выбора скрывает в себе опасность для недостаточно опытного, особенно в формальной логике, программиста (а школьник — это как раз этот случай и есть). Опасность эта — риск запутаться не в технике написания команды, а именно в понимании логики, которой должна следовать эта команда.

На начальной стадии изучения команды выбора (ветвления) во всех примерах мы будем использовать только **одно** условие. Разумеется, что возможна проверка сразу нескольких условий (это когда речь идет о так называемом «составном условии»), но мы займемся правилами записи составных условий позже.

Раздел 9

Короткий оператор выбора (if без else)

Суть работы такого оператора выглядит следующим образом: если некое условие выполняется (является правильным, истинным), то надо выполнить некоторое действие.

А если это условие не выполняется? Что ж, в этом случае класс просто переходит к исполнению следующей команды.

Таким образом, распределение в таком операторе «по веткам» выглядит так:

- одной «веткой» в таком операторе является команда или группа команд (блок), записанные как часть самой команды выбора (ветвления);
- второй «веткой» является следующая (по порядку) команда класса.

Короткий оператор выбора (на языке Java)	**Смысл оператора (перевод на «нормальный» язык)**
if (a>0) System.out.println("positive");	Если значение переменной **a** больше нуля (переменная **a** хранит положительное значение), то на экран выводится слово positive
if (a>b) System.out.println(a-b);	Если значение переменной **a** больше значения переменной **b**, то на экран выводится разница между значение переменной **a** и значением переменной **b**
if (a==0) a=4;	Если значение переменной **a** равно нулю, то переменной **a** присваивается новое значение, равное 4
if (a!=b) c=a*b;	Если значение переменной **a** не равно значению переменной **b**, то переменная **c** получает значение, равное произведению значений переменных **a** и **b**

Важно соблюдать одно техническое требование языка Java: проверяемое в операторе выбора условие обязательно(!) должно быть заключено в скобки.

И обратите внимание, что в приведенных выше примерах «ветка» внутри команды выбора (ветвления) содержит только одну-единственную команду.

Таблица сравнений, используемых в командах выбора языка Java

Оператор сравнения	Смысл
>	Больше
>=	Больше или равно (≥)
==	Действительно равно
!=	Не равно (≠)
<	Меньше
<=	Меньше или равно (≤)

В случае если требуется, чтобы эта исполняемая «ветка» содержала две команды или больше, то ее, эту «ветку», необходимо заключить в фигурные скобки. Приведем примеры таких команд.

Короткий оператор выбора (на языке Java)	Смысл оператора (перевод на «нормальный» язык)

if (a>0)
{
a++;
Sys-
tem.out.println("positive");
}
	Если значение переменной **a** больше нуля (переменная **a** хранит положительное значение), то надо сначала увеличить значение этой переменной на 1, а затем вывести на экран слово positive

if (a>b)
{
a++;
b-=2;
System.out.println(a-b);
}
	Если значение переменной **a** больше значения переменной **b**, то сначала значение переменной **a** увеличивается на 1, затем значение переменной **b** уменьшается на 2, а потом на экран выводится разница между новыми значениями переменных **a** и **b**

if (a==0)
{
a=4;
System.out.println("Ok");
}
	Если значение переменной **a** и в самом деле равно нулю, то сначала переменной **a** присваивается новое значение, равное 4, а затем на экран выводится сообщение Ok

Окончание таблицы

Короткий оператор выбора (на языке Java)	Смысл оператора (перевод на «нормальный» язык)
`if (a!=b) { c=a*b; a=b; b=0; }`	Если значение переменной **a** не равно значению переменной **b**, то сначала переменная **c** получает значение, равное произведению значений переменных **a** и **b**, потом переменная **a** получает значение, хранящееся в переменной **b**, а после этого значение переменной **b** устанавливается равным 0

Раздел 10

Стандартная команда выбора (if... else)

В предыдущем разделе мы рассмотрели команду выбора, которая содержала только одну «ветку», — ту, которая исполнялась, в случае если оказывалось правильным условие, записанное в команде. Второй «веткой» при этом была следующая за командой выбора часть класса (точнее — главного метода).

Однако у команды выбора есть вариант, который позволяет записать в самой команде выбора сразу обе «ветки»: первую, которая исполняется, когда выполняется содержащееся в команде условие, и вторую — ту, что исполняется, когда записанное в команде условие **не** выполняется (неверно, не является истинным).

В этом варианте вторая «ветка» начинается со служебного слова **else**. Такую команду выбора (ветвления) часто называют *стандартной командой выбора*.

Иными словами, стандартная команда выбора в языке Java состоит из двух частей:

- первая часть: начинается с **if**, содержит условие и ту группу команд, которая выполняется, если условие оказывается верным;
- вторая часть: начинается с **else**, содержит только ту группу команд, которая выполняется, если условие оказывается неверным.

В качестве «группы команд» может выступать и одна команда, а в редких, почти не встречающихся случаях — вообще «пустая группа»; хотя очень трудно представить, для чего такое может понадобиться, теоретически существует и такая возможность.

Приведем несколько примеров таких стандартных команд выбора, с пояснениями.

if (a>5) t=a+7; else t=a-3;	Если значение переменной **a** больше 5, то переменная **t** получает значение на 7 большее значения переменной a; в противном случае переменная **t** получает значение на 3 меньшее значения переменной **a**
if (a>b) c=a+b; else System.out.println("No");	Если значение переменной **a** больше значения переменной **b**, то переменная **c** получает значение, равное сумме значений переменных **a** и **b**; в противном случае на экран выводится сообщение **No**

Разумеется, каждая ветка может содержать не одну-единственную команду, как это показано в приведенных в таблице примерах, а несколько команд. Их в этом случае надо будет объединить в блоки, то есть заключить в фигурные скобки. Более подробно об этом — в следующем разделе.

Раздел 11

Простая команда выбора с блоком (блоками)

Часто бывает необходимость, чтобы в команде выбора одна или обе «ветки» (и та, что выполняется, когда условие верно, и та, которая выполняется, когда условие не соблюдено) содержали не одну-единственную команду, а несколько команд.

Например, если проверяемое в условии команды выбора значение (переменная) отрицательно, то следует выполнить несколько условий:

- Сначала превратить значение переменной в положительное.
- Затем вывести на экран текстовое сообщение **Значение изменено на положительное**.
- Затем увеличить это значение в 4 раза.
- И в итоге вывести новое значение на экран в дополнительной строке.

Для того чтобы в команде выбора можно было выполнить все эти действия, в той последовательности, как они описаны, их объединяют в так называемый «блок», то есть группу команд, которые класс воспринимает как единое целое.

Для этого используются уже знакомые нам фигурные скобки (пара фигурных скобок), в которых и записывают нужные для включения в блок команды.

Для нашего примера это будет выглядеть следующим образом:

```
if (x<0)
   {
    x=-1*x;
    System.out.println("Значение изменено на положительное ");
    x=4*x;
    System.out.println(x);
   }
```

Точно так же можно использовать блок и после **else** — следует только обязательно обратить внимание, что в конце первого блока (после закрываю-

щей фигурной скобки, после которой идет **else**), **ни в коем случае не ставится** знак «точка с запятой»!

Вот как будет выглядеть команда выбора с двумя блоками:

```
if (x<0)
  {
   x=-1*x;
   System.out.println("Значение изменено на положительное ");
   x=4*x;
   System.out.println(x);
  }
  else
  {
   x=2+x;
   System.out.println("Новое значение "+x);
  }
```

Мы приводим в качестве примеров словесного описания класса два алгоритма, которые можно «перевести» на язык Java, то есть написать соответствующий алгоритму класс.

Обратите внимание на нумерацию действий в алгоритмах: с ее помощью легко выделить то, что мы только что описывали как «блоки». В частности, в алгоритме 2 таких блоков нет, а в алгоритме 1 — есть: это действия, пронумерованные как (3.1) и (3.2).

Алгоритм 1

1. Введи с клавиатуры целое число для переменной num.
2. Если в переменной num находится четное число, то:
   2.1. выведи на экран значение переменной num.
3. Иначе:
   3.1. увеличь значение переменной num на 1;
   3.2. выведи на экран значение переменной num.
4. Выведи на экран сообщение **Работа завершена.**

Алгоритм 2

1. Введи с клавиатуры две цифры — в переменные dig1 и dig2.
2. Если dig1>dig2, то:
   2.1. num← dig1*10+dig2.
3. Иначе:
   3.1. num← dig2*10+dig1.
4. Выведи на экран значение переменной num.

Раздел 12

Команда if со сложным условием

До сих пор мы рассматривали задачи, в которых, по сути дела, достаточно было проверить одно условие, — или несколько раз разные условия, но, все же, каждое из этих условий проверялось отдельно от остальных.

«...если введенное значение положительно...»

«...если значение первой переменной больше или равно значению второй переменной...»

«...для четного числа... а для нечетного...»

Однако очень часто встречаются ситуации, в которых необходимо проверять не одно-единственное условие, а два; а иногда и больше; что самое главное — проверять их не каждое в независимости от остальных, а все вместе, «в одной связке». В этом случае, команда выбора (ветвления) позволяет использовать *сложное (составное) условие.*

Сложное (составное) условие состоит из нескольких условий, связанных между собой *логическими отношениями*, например:

- «и...и...», то есть условия должны выполняться оба, одновременно;
- «или... или...», то есть достаточно выполнение хотя бы одного из условий.

Таблица логических сравнений (для простых условий)

Пример	**Запись сравнения в языке JAVA**	**Форма записи, принятая в математике**	**Описание смысла логического сравнения**
(a==3)	==	$=$	Равно
(a!=0)	!=	$\neq$	не равно
(a<b)	<	$<$	Меньше
(a<=b)	<=	$\leq$	меньше или равно
(a+b>c)	>	$>$	Больше
(a+b>c*2)	>=	$\geq$	больше или равно

Таблица логических связей между условиями (для составного условия)

<table>
<tr><th>Запись логической связи в языке JAVA</th><th>Пример</th><th>Описание смысла логической связи в составном условии</th></tr>
<tr><td>&</td><td>(a<100)&(a>9)</td><td>и… и… (AND, И)</td></tr>
<tr><td>|</td><td>(a==0) | (b==0)</td><td>или… или… (OR, ИЛИ)</td></tr>
<tr><td>!</td><td>!(a<10)</td><td>не</td></tr>
<tr><td>&&</td><td>(a<100)&&(a>9)</td><td>«краткое», «быстрое» и
(short circuit AND)</td></tr>
<tr><td>||</td><td>(a==0) || (b==0)</td><td>«краткое», «быстрое» или
(short circuit OR)</td></tr>
</table>

Таблица истинности

<table>
<tr><th>a</th><th>b</th><th>a & b
a && b</th><th>a | b
a || b</th></tr>
<tr><td>true</td><td>true</td><td>true</td><td>true</td></tr>
<tr><td>false</td><td>false</td><td>false</td><td>false</td></tr>
<tr><td>true</td><td>false</td><td>false</td><td>true</td></tr>
<tr><td>false</td><td>true</td><td>false</td><td>true</td></tr>
</table>

Одной из особенностей языка Java является использование так называемых «быстрых» (иногда их еще называют «краткими») форм логических связей между условиями. На них следует остановиться подробнее, поскольку они и широко распространены, и имеют определенные преимущества и специфику по сравнению со «стандартными» формами записи логических связей между условиями.

Операторы быстрой (краткой) оценки логических выражений (short circuit logical operators && и ||)

В языке Java существуют два интересных дополнения к набору логических операторов.

Это — альтернативные версии операторов AND и OR, служащие для быстрой оценки логических выражений. Строго говоря, сегодня эти операторы уже давно перестали быть альтернативными, а является как раз наиболее широко применяемыми вариантами записи логических связей в составном условии.

Как видно из таблицы истинности, составное условие с «и... и...» возвращает результат **true**, если каждое из условий возвращает **true**; и возвращает результат **false**, если хотя бы одно из условий возвращает **false**. При этом, если уже первое условие (в составе сложного условия) возвращает **false**, то нет никакой, строго говоря, необходимости проверять второе условие — ведь все равно в результате будет возвращено **false**.

Вот по такой «краткой», «быстрой» схеме и работает оператор «быстрое и... и...», обозначаемый двойным символом «и» — **&&**. Это означает, что если первое условие возвращает **false**, то значение второго условия вообще не вычисляется, потому что на общий результат оно уже все равно не повлияет — результат все равно в этом случае будет равен **false**!

Аналогично обстоит дело и с оператором ||: если первое условие возвращает значение **true**, то независимо от того, что возвращает второе условие, общим результатом все равно же будет величина **true**. А посему — второе условие просто вообще не проверяется...

Если вы используете операторы && и || вместо обычных форм & и |, то Java не производит оценку правого (стоящего после знака логической связи) выражения, если ответ ясен из значения левого (стоящего перед знаком логической связи) выражения. Общепринятой практикой является использование операторов && и || практически во всех случаях оценки булевых логических выражений.

Приведем пример, демонстрирующий разницу в применении операторов && и &:

Даны две команды выбора:

```
if (b!=0 && a/b >1) System.out.println ("a>b");
if (b!=0 & a/b >1) System.out.println ("a>b");
```

В случае, когда значение переменной **b** равно 0, использование второй записи приведет к сообщению об ошибке (во время выполнения программы!), так как деление на 0 является запрещенной операцией.

А вот в первом примере этого сбоя не произойдет, так как класс определит, что условие **b!=0** не выполняется, и после этого вообще не будет проверять второе условие **(a/b >1)**.

Важно помнить, что составное выражение **обязательно** должно быть заключено в скобки (круглые), в то время как внутренние логические выражения (условия), входящие в его состав, можно и не заключать в скобки (каждое по отдельности), а можно и заключать.

Наиболее распространенные проверки

<table>
<tr><td>if (x>0)… if (x<0)…
if (x>=0)… или if (x>0 || x==0)...</td><td>x — положительное, отрицательное, неотрицательное</td></tr>
<tr><td>if (x>9 && x<100)…</td><td>Целое число x — двузначное положительное</td></tr>
<tr><td>if (x>-10 && x<10)…</td><td>Целое число x — однозначное</td></tr>
<tr><td>if (x %2==0)…</td><td>Целое число x — четное</td></tr>
<tr><td>if (x %10==y)…</td><td>y равно последней цифре в x</td></tr>
<tr><td>if (x/10 %10==y)…</td><td>y равно числу десятков в x</td></tr>
<tr><td>if (x<=y && y<=z)…</td><td>x, y, z — упорядочены по возрастанию</td></tr>
<tr><td>if (x % y ==0)…</td><td>x делится на y без остатка</td></tr>
<tr><td>if (x==(int) x)…</td><td>У положительного вещественного числа x — нет десятичной части</td></tr>
</table>

Вы наверняка заметили, что мы постоянно использовали фразу «значение логического выражения» — рассчитывая на то, что смысл ее будет в общем смысле достаточно понятен.

Но, на самом деле, в математике существует целый раздел, занимающийся не числовой алгеброй, а алгеброй логических значений и выражений — и он не менее сложен, чем любой другой раздел математики.

Именно поэтому мы считаем важным включить в число рассматриваемых тем в нашем курсе и небольшой раздел, посвященный использованию логических значений и выражений в компьютерной сфере. Важность этой темы демонстрируется еще и тем, что в языке Java, как и в практически всех современных языках, существует специальный логический тип значений и переменных.

Раздел 13

Булевы значения и переменные

Мы уже познакомились с несколькими типами значений и переменных: целыми числами, дробными, символьными и строковыми. Кроме того, при изучении команды выбора (ветвления) **if**, разбирая условия, которые в этой команде используются, мы упоминали два логических «результата», которые «образуются» в результате проверки условий — **false** и **true**.

Эти результаты «сигнализируют» о том, каким оказывается результат проверки условия:

- в случае если условие не выполняется, — результатом будет значение **false**; по-русски часто говорят, что результатом будет значение **ложь**;
- в случае если условие выполняется, — результатом будет значение **true**; по-русски часто говорят, что результатом будет значение **истина**.

Поскольку логические значения и выражения имеют очень широкое применение в программировании, в языке Java для них введен отдельный тип: **boolean**. У этого типа есть всего два значения — уже упомянутые нами **true** и **false**. Соответственно переменные этого типа могут принимать одно из этих двух значений.

Кстати, название этого типа связано с английским математиком (и учителем математики) Джорджем Булем (1815–1864), которого по праву считают создателем такого направления науки, как «математическая логика».

Приведем несколько примеров объявления и инициализации переменных этого типа.

```
boolean x, y;
x=false;
y=x;
```

```
boolean x=true, y=true;
```

Кроме таких способов «прямой инициализации», переменные этого типа могут получать значения в результате проверки на истинность каких-либо математических или логических выражений.

Приведем пример:

```
boolean x, y;
x=(a>b);
y=!x;
```

исполняется приведенный фрагмент следующим образом: в случае если значение переменной **a** будет больше значения переменной **b**, — переменная **x** получит значение **true**, а затем переменная **y** получит значение **false**. Дело в том, что выражение в третьей строке нашего примера имеет смысл «значение переменной **y** равно иному (противоположному) значению, по отношению к тому, какое значение имеется в переменной **x**».

При этом следует иметь в виду, что в логике:

- «иное значение» по отношению к значению **истина** — это значение **ложь;**
- «иное значение» по отношению к значению **ложь** — это значение **истина**.

Переменные булевого типа (логические переменные, как их еще называют) позволяют записывать некоторые выражения в сокращенном виде, особенно в командах выбора (ветвления).

Приведем пример:

```
boolean flag=(a!=b);
if (flag) System.out.print("Переменные не равны");
```

В приведенном примере переменная **flag** получает значение **true,** в случае если значения переменных **a** и **b** — различны. В случае если значения этих переменных равны, логическая переменная **flag** получает значение **false**.

Затем, если значение переменной **flag** оказалось равно **true**, то на экран выводится сообщение **Переменные не равны**. В случае же если значение этой переменной окажется равным **false**, приведенное в нашем фрагменте сообщение не будет выводиться на экран.

Раздел 14

Наиболее используемые функции библиотеки Math. Использование функции random — работа со случайными числами

В языке Java имеется набор «готовых к употреблению» математических функций (команд), которые позволяют упростить выполнение некоторых важных вычислений, операций и действий.

Ниже приведена таблица наиболее часто используемых таких функций.

Функция	**Краткое опцсание**	**Примеры применения и результат**
Math.abs(x)	Абсолютное значение x	Math.abs(10) →10 Math.abs(-10) →10
Math.pow(a,b)	a в степени b (a^b)	Math.pow(2,3) → 8.0 Math.pow(-2,3) → -8.0 Math.pow(5.5,2) → 30.25 Math.pow(2,2.5) → 5.66
Math.sqrt(x)	Квадратный корень (положительный)	Math.sqrt(16) → 4.0
Math.round(x)	Округленное число	Math.round(6.7) → 7 Math.round(6.3) → 6 Math.round(-6.7) → -7 Math.round(-6.3) → -6
Math.floor(x)	Ближайшее слева число с нулевой дробной частью	Math.floor(15.2) → 15.0 Math.floor(0.98) → 0.0 Math.floor(-9.8) → -10.0 Math.floor(-9.2) → -10.0
Math.ceil(x)	Ближайшее справа число с нулевой дробной частью	Math.ceil(15.2) → 16.0 Math.ceil(0.98) → 1.0 Math.ceil(-9.8) → -9.0 Math.ceil(-9.2) → -9.0

Окончание таблицы

Math.min(a,b)	Наименьшее из двух значений	Math.min(5,9) → 5 Math.min(5.2,1.1) → 1.1
Math.max(a,b)	Наибольшее из двух значений	Math.max(5,9) → 9 Math.max(5.2,1.1) → 5.2
Math.random()	Случайная десятичная дробь из диапазона от 0 (включая) до 1 (не включая)	

Дополним таблицу несколькими важными пояснениями:

- Все перечисленные в таблице функции принимают в качестве параметров переменные всех типов (int, long, double, float). Исключение составляет функция Math.round(x), которая выполняет округление параметра, и потому параметр должен быть или типа double, или типа float.
- Обратите внимание, что функция Math.random(), создающая случайные дроби, вообще не имеет параметра.
- Функции Math.pow(a,b), Math.sqrt(x), Math.floor(x), Math.ceil(x), Math.random() — обязательно возвращают значение типа double
- Функция Math.round(x) преобразует параметр типа float в значение типа int, а параметр типа double в значение типа long.
- Функции Math.abs(x), Math.min(a,b), Math.max(a,b) возвращают тип значения в соответствии с используемыми в каждом конкретном случае типами параметров.

Кроме этих функций, в классе (библиотеке) Math имеются еще несколько, которые могут оказаться полезными при программировании задач, связанных, например, с математикой и физикой.

Среди них, в частности:

- Math.cos(n), Math.sin(n), Math.tan(n) — тригонометрические функции синус, косинус и тангенс, вычисляемые от аргумента n, указанного в радианах.
- Math.acos(n), Math.asin(n), Math.atan(n) — обратные тригонометрические функции (арккосинус, арксинус и арктангенс), возвращают значение угла в радианах.
- Math.toDegrees(n) — возвращает градусную меру угла в n радианов (пересчитывает величину угла, выраженную в радианах, — в величину того же угла, но выраженную в угловых градусах).
- Math.toRadians(n) — возвращает радианную меру угла в n градусов (пересчитывает величину угла, выраженную в угловых градусах, — в величину того же угла, но выраженную в радианах).

- Math.log(n) — возвращает значение натурального логарифма числа n.
- Math.log10(n) — возвращает значение десятичного логарифма числа n.

Все перечисленные функции принимают вещественные аргументы, а тип возвращаемого значения зависит от типа аргумента и от самой функции.

Кроме функций в рассматриваемом классе имеются две часто используемых константы:

- Math.PI — число «пи», с точностью в 15 десятичных знаков.
- Math.E — число Непера (основание экспоненциальной функции), с точностью в 15 десятичных знаков.

Примеры использования функций этой библиотеки:

- System.out.println(Math.abs(-2.33)); // выведет на экран 2.33.
- System.out.println(Math.round(Math.PI)); // выведет на экран значение числа «пи».
- System.out.println(Math.round(9.5)); // выведет на экран 10.
- System.out.println(Math.round(9.5-0.001)); // выведет на экран 9.
- System.out.println(Math.ceil(9.4)); // выведет на экран 10.0.
- double c = Math.sqrt(3*3 + 4*4);
  System.out.println(c); // выведет на экран значение длины гипотенузы прямоугольного треугольника с катетами 3 и 4.
- double s1 = Math.cos(Math.toRadians(60));
  System.out.println(s1); // выведет на экран косинус угла в 60 градусов.

Работа со случайными числами

Функция библиотеки Math, генерирующая случайные дроби, Math.random(), — одна из наиболее широко используемых в самых разных задачах. Дело в том, что использование этой функции позволяет избежать многократного введения данных с клавиатуры, — особенно актуально это становится в заданиях, которые требуют вводить с клавиатуры десятки, а то и сотни значений.

В ближайших темах мы начнем изучение и использование так называемых «команд повторения» (циклов) — вот тогда мы и сможем в полной мере оценить использование функции-генератора случайных чисел.

Как уже было сказано, для работы со случайными числами в языке Java используется функция Math.random(), которая выбирает случайным образом одну из десятичных дробей из диапазона от 0 (включая) до 1 (не включая).

Для того чтобы увеличить диапазон выбора чисел, умножают функцию на соответствующее значение: например, 60*Math.random() выбирает слу-

чайным образом одну из десятичных дробей из диапазона от 0 (включая) до 60 (не включая).

Для того чтобы «сдвинуть» диапазон выбора чисел, добавляют к функции число (положительное или отрицательное), устанавливающее диапазон сдвига: например, 60*Math.random()+10 выбирает случайным образом одну из десятичных дробей из диапазона от 10 (включая) до 70 (не включая).

Для того чтобы использовать эту функцию для выбора целых чисел, производят действие «преобразования типа»: перед заключенным в скобки выражением ставят в скобках указание на тип, в который надо произвести преобразование — в нашем случае, для целых чисел, это (int).

Таким образом, если общий вид команды для выбора случайных чисел выглядит, например, следующим образом:

x=-20 + (int)(75*Math.random())

то ее смысл — *«в переменную **x** заносится случайным образом выбранное целое число, одно из 75-и чисел, начиная с –20 и вверх»*, — или *«в переменную **x** заносится случайным образом выбранное целое число из диапазона от –20 до +54»*.

Общая формула для генерации случайного целого числа x, такого, что $a \leq x \leq b$:

x= a+ (int)(Math.random()*(b-a+1))

Например, для генерации случайного целого положительного двузначного числа (a=10; b=99) мы получаем:

x= 10+ (int)(Math.random()*(99-10+1))
или x= 10+ (int)(Math.random()*90)

Обратите внимание, что 90 — это количество двузначных положительных чисел (количество вариантов для генерации случайного числа), а 10 — значение минимального варианта.

Раздел 15 (часть 1)

Команды повтора — Цикл for

В большинстве классов (программ) есть необходимость в многократном выполнении каких-то команд, действий, блоков.

Приведем несколько примеров таких действий:

- необходимо вывести на экран серию из 100 чисел;
- подсчитать сумму 250 элементов арифметической последовательности;
- принять с клавиатуры 25 значений и найти наибольшее из них;
- и так далее...

Разумеется, писать для каждой из таких задач код из последовательных команд — не только утомительно, но и совершенно неразумно: мало того, что количество строк кода становится слишком большим при слишком простой цели, так еще и коды содержат фактически одни и те же, повторяющиеся снова и снова команды.

Вот как раз для решения такого рода проблем («много строк фактически повторяющегося кода») в каждом языке, в том числе и в Java, существует инструмент под названием «цикл». С помощью цикла можно написать класс таким образом, чтобы те действия, которые необходимо повторять снова и снова, были записаны один раз, но внутри повторяющегося фрагмента (в этом случае говорят — внутри *тела цикла*).

Кстати, в компьютерных науках циклы называются «инструментом повторного исполнения» — с их помощью можно писать классы по принципу «мало написано — много делается».

В Java существует несколько видов циклов — и мы начнем знакомиться с ними с цикла, который называется «цикл for». Он также известен как «цикл с параметром», «цикл с заранее известным числом повторов».

Следует иметь в виду, что в языке Java цикл **for** является чрезвычайно мощным инструментом, но в рамках данного курса мы его используем достаточно «классически», ограничиваясь только частью его возможностей.

Сегодня в языке Java цикл for получил массу дополнительных возможностей и это, с одной стороны, сделало его гораздо более сильным инструментом, но, при этом, и более сложным; с другой — его по-прежнему можно использовать в стандартном, классическом виде.

Базовая форма цикла for

Цикл for состоит из двух частей:

- заголовка цикла;
- тела цикла.

В заголовке цикла указывается, сколько раз будет исполняться цикл, а в теле цикла указываются (записываются) те команды, которые должны выполняться снова и снова при каждом повторе (проходе) цикла.

Заголовок цикла выглядит следующим образом (мы приведем для начала общую форму):

for (начальное значение *счетчика***;** логическое условие — «до каких пор продолжать повторы»**;** изменение *счетчика* при каждом повторе**)**

Важно обратить внимания на обязательные требования записи заголовка:

- служебное слово for пишется маленькими буквами;
- поле него — пара круглых скобок;
- после начального значения счетчика и после логического условия ставится знак «точка с запятой» (;);
- после закрывающей скобки в заголовке **не ставится** знак «точка с запятой» (то есть — конец команды).

Примеры записи заголовка цикла for:

- for (int i=1; i<=10; i++)
- for (int i=1; i<=10; i+=2)
- for (int i=10; i>0; i-=3)

Объяснения этих примеров будут приведены чуть позже.

Общий принцип работы цикла for можно описать примерно так:

1. сначала *переменная цикла* (*счетчик*) устанавливается в начальное значение;
2. затем выполняется первый проход (повтор) цикла, при котором выполняются команды, записанные в теле цикла;
3. затем *переменная цикла* (*счетчик*) изменяется в соответствии с правилом, записанным в заголовке; правило это выполняется сразу после исполнения тела цикла — и до того, как проверено условием;
4. проверяется условие продолжения выполнения цикла:
   4.1. если значение измененного *переменной цикла* (*счетчика*) соответствует условию, записанному в заголовке, — снова выполняются пункты 2, 3 и 4;
   4.2. если это условие не выполняется, — цикл завершен и исполнение передается команде, записанной после цикла (после тела цикла).

Как видно из описания общего принципа работы цикла for, одной из важных компонент цикла является переменная цикла-счетчик. Кстати, именно поэтому у цикла for есть еще одно название: «цикл со счетчиком».

Изменение этой переменной цикла-счетчика, собственно, является одним из фактором, влияющих на число проходов (повторов) цикла — и это можно установить по заголовку. Приведем несколько примеров, используя, среди прочих, и те, которые только что были записаны выше в качестве примеров.

Заголовок цикла	Смысл заголовка
for (inti=1; i<=10; i++)	• Начальное значение счетчика (переменная i) устанавливается равным 1 • За каждый проход счетчик увеличивается на 1 • Повторы цикла продолжаются, пока значение счетчика остается не больше 10 • Всего выполняется 10 повторов цикла
for (inti=1; i<=10; i+=2)	• Начальное значение счетчика (переменная i) устанавливается равным 1 • За каждый проход счетчик увеличивается на 2 • Повторы цикла продолжаются, пока значение счетчика остается не больше 10 • Всего выполняется 5 повторов цикла
for (int i=10; i>0; i-=3)	• Начальное значение счетчика (переменная i) устанавливается равным 10 • За каждый проход счетчик уменьшается на 1 • Повторы цикла продолжаются, пока значение счетчика остается больше 0 • Всего выполняется 4 повтора цикла
for (int i=1; i<=a; i++)	• Начальное значение счетчика (переменная i) устанавливается равным 1 • За каждый проход счетчик увеличивается на 1 • Повторы цикла продолжаются, пока значение счетчика остается не больше значения переменной **a** • Всего выполняется **a-1+1** повторов цикла
for (int i=a; i<=10; i+=2)	• Начальное значение счетчика (переменная i) устанавливается равным **a** • За каждый проход счетчик увеличивается на 2 • Повторы цикла продолжаются, пока значение счетчика остается не больше 10

Окончание таблицы

Заголовок цикла	Смысл заголовка
	• Всего выполняется **(10-a+1)/2** повторов цикла (обратите внимание, — речь идет не про деление, а про вычисление частного!)
for (int i=a; i>b; i-=3)	• Начальное значение счетчика (переменная i) устанавливается равным **a** • За каждый проход счетчик уменьшается на 3 • Повторы цикла продолжаются, пока значение счетчика остается больше **b** • Всего выполняется **(a-b)/3** повторов цикла (обратите внимание, — речь идет не про деление, а про вычисление частного!)

Как видите, даже если в заголовке цикла используются не только числа, но и дополнительные переменные, всегда заранее известно, сколько проходов (повторов) выполнит цикл, — ведь эти переменные должны иметь значения!

Главное достоинство такого цикла в том, что его заголовок (цикла for) всегда можно записать в максимально простом и понятном виде. Именно это и делает базовую, «классическую» форму заголовка цикла for наиболее распространенной.

Примеры циклов for

Приведем несколько примеров того, как можно использовать цикл **for** в кодах.

Пример 1

Написать цикл, который выводит на экран (в одну сроку) все целые положительные двузначные числа, начиная с наименьшего:

```
for (int i=10; i<100; i++ )
{
        System.out.print( i );
}
```

Пример 2

Написать цикл, который генерирует и выводит на экран 26 целых случайных чисел из диапазона от 10 до 45:

```
for (int i=1; i<=26; i++ )
{
    x=(int)(36*Math.random()+10);
    System.out.println( x );
}
```

Пример 3

Написать цикл, который принимает с клавиатуры 45 десятичных дробей и для введенных положительных значений выводит на экран квадратные корни (с соответствующим текстовым сообщением):

```
for (int i=1; i<46; i++ )
{
    x=reader.nextDouble();
    if (x>=0)
    {
        y=Math.sqrt(x);
        System.out.println("Квадратный корень "+x+" равен "+y );
    }
    else
        System.out.println("нельзя извлечь квадратный корень!");

}
```

Пример 4

Написать цикл, который принимает с клавиатуры целое число, затем генерирует 40 случайных чисел из диапазона от 1 до 6.
Цикл выводит на экран (в одну строку) только те из сгенерированных случайных чисел, которые имеют четное значение.

```
x=reader.nextInt();
for (int i=1; i<=40; i++ )
{
    y=1+(int)(6*Math.random();
    if (y%2==0) System.out.print( y );
}
```

Следует всегда обращать внимание на один очень важный момент при написании этого рода циклов: если переменная-счетчик объявляется внутри заголовка цикла (так, как это сделано во всех приведенных выше при-

мерах), то она «существует» только в пределах цикла (заголовка и тела). Иными словами — всякая попытка использовать эту переменную уже **после** тела цикла будет вызывать ошибку.

Сравните два главных метода:

<table>
<tr><td><pre>public static void main(String[]args)
{
  for (int i=1; i<7;i++)
  {
System.out.print(i+" ");
  }
System.out.println(" and "+i);
}</pre></td><td>Уже на стадии компиляции появляется сообщение об ошибке, смысл которой: необъявленная переменная.

При этом в качестве строки, содержащей эту необъявленную переменную, указывается:

System.out.println(" and "+i);</td></tr>
<tr><td><pre>public static void main(String[]args)
{
int i;
  for (i=1; i<7;i++)
  {
System.out.print(i+" ");
  }
System.out.println(" and "+i);
}</pre></td><td>Нет проблем ни на стадии компиляции, ни на стадии исполнения</td></tr>
</table>

Приведем еще два примера — возможные решения следующего задания: *«напиши класс, который выводит на экран последовательность двузначных положительных чисел, делящихся на 6».*

<table>
<tr><td><pre>class primer_for2
{
 public static void main(String[]args)
 {
   for (int i=12; i<=96; i+=6)
   {
System.out.print(i+" ");
   }
 }
}</pre></td><td>В этом варианте переменная-счетчик принимает только нужные значения: начиная с 12 (все равно 10 и 11 не делятся на 12!), заканчивая 96 (все равно 97, 98 и 99 тоже не делятся на 12!), и, увеличиваясь при каждом проходе (повторе цикла) на 6 (все равно нет смысла проверять все числа подряд!).

При этом сама переменная-счетчик и используется для вывода значений на экран</td></tr>
</table>

Окончание таблицы

<table>
<tr><td><pre>class primer_for3
{
public static void main(String[]args)
  {
  int x=10;
  for (int i=1; i<=90; i++)
     {
      if ((x+i)%6==0) Sys-
tem.out.print(x+" ")
     }
  }
}</pre></td><td>В этом варианте переменная счетчик отвечает только за установление количества проходов, которое равно общему числу двузначных положительных чисел — и совершенно не используется в теле цикла.

За установление значений, которые будут выводиться на экран, отвечает отдельная переменная (x), которая принимает значения всех возможных положительных двузначных чисел, — и, поэтому, каждый раз (при каждом проходе, повторе цикла) производится проверка — выводить ли значение на экран</td></tr>
</table>

Мы хотим привести еще три варианта решения только что разобранного примера и предлагаем каждому оценить различия в этих вариантах.

<table>
<tr><td><pre>class primer_for3
{
public static void main(String[]args)
  {
for (int i=10; i<=99; i++)
     {
if (i %6==0) System.out.print(i+" ");
     }
  }
}</pre></td><td>Вариант 1</td></tr>
<tr><td><pre>class primer_for3
{
public static void main(String[]args)
  {
for (int i=12; i<=96; i+=6)
     {
     System.out.print(i+" ");
     }
  }
}</pre></td><td>Вариант 2</td></tr>
</table>

Окончание таблицы

class primer_for3
{
public static void main(String[]args)
  {
for (int i=2; i<=12; i++)
    {
    System.out.print(i*6+" ");
    }
  }
}
	Вариант 3

В будущих разделах мы познакомимся с дополнительными возможностями, которые предоставляет программисту цикл for, но на данный момент гораздо важнее освоиться с пониманием и написанием классов, использующих базовую, «классическую» форму цикла for.

Раздел 15 (часть 2)

Команды повтора — Цикл for

Продолжим наше знакомство с возможностями цикла for — и на этот раз поговорим об использовании в этом цикле не только конкретных числовых значениях, но и различных переменных. Их использование делает цикл for более «гибким», то есть количество проходов все равно всегда заранее известно, но теперь оно зависит от того, какие значение принимают используемые в заголовке цикла переменные.

Поясним это примером.

Дан класс.

```
class primer_for2
{
public static void main(String[] args)
{
Int x;
for (int i=1; i<=10; i++)
  {
   x=(int)(6*Math.random()+1);
   System.out.print(x+" ");
  }
  System.out.println("Конец серии");
}
}
```

Этот класс генерирует и выводит на экран (в одной строке) 10 случайных чисел из диапазона от 1 до 6, затем в конце строки выводит сообщение **Конец серии** и переходит на новую строку.

При исполнении программы на экране будут выводиться, например, такие серии числовых значений:

4 2 6 4 2 1 6 1 6 4 Конец серии

или

5 3 5 4 1 1 3 4 6 4 Конец серии

или

6 3 4 2 3 4 3 4 2 3 Конец серии

и так далее...

Обратите внимание, что состав серий каждый раз иной, но общими для всех случаев остаются два признака:

- числа в серии всегда принадлежат к диапазону [1;6];
- количество чисел в серии **всегда** равно 10.

При этом количество чисел в серии определяется тем, как написан заголовок цикла: переменная-счетчик принимает последовательно целые значения от 1 до 10 — что и определяет количество проходов (повторов) цикла.

Если же мы хотим, чтобы количество проходов (повторов) цикла не оставалось всегда равным 10, а определялось, например, числом, которое будет вводиться с клавиатуры, класс надо будет изменить.

```
import java.util.*;
class primer_for3
{
static Scanner reader =  new Scanner(System.in);
public static void main(String[] args)
{
  int x,a;
  a=reader.nextInt();
  for (int i=1; i<=a; i++)
  {
    x=(int)(6*Math.random()+1);
    System.out.print(x+" ");
  }
  System.out.println("Конец серии ");
}
}
```

Как видите, количество проходов (повторов) цикла теперь будет выражаться не числом, которое будет оставаться неизменным при каждом запуске класса, а тем значением (числом), которое перед началом исполнения цикла будет вводиться с клавиатуры (в качестве значения для переменной **a**).

Например, если с клавиатуры будет введено значение 3, то на экран будет выводиться, например:

5 1 2 Конец серии

А вот если с клавиатуры будет введено значение 16, то на экран будет выводится, например:

2 4 3 5 1 1 5 1 2 6 3 2 4 5 2 4 Конец серии.

В общем виде для приведенного примера число проходов (повторов) цикла будет выражаться не числом, а выражением **a-1+1** (то есть просто значением переменной **a**).

Разумеется, в заголовке можно использовать несколько переменных, например:

Заголовок цикла	**Выражение, которое определяет количество проходов (повторов) цикла, с объяснением**
for (int i=5; i<b; i++)	**b-5** раз (обратите внимание на строгое неравенство)
for (int i=5; i<b; i+=2)	Количество повторов (проходов) цикла будет равно частному от деления **b-5** на 2 (значение переменной-счетчика при каждом повторе увеличивается на 2)
for (int i=a; i<b; i+=5)	Количество повторов (проходов) цикла будет равно частному от деления **b-a** на 5 (значение переменной-счетчика при каждом повторе увеличивается на 5)
for (int i=a; i<b; i+=c)	Количество повторов (проходов) цикла будет равно частному от деления **b-a** на значение переменной **c** (значение переменной-счетчика при каждом повторе увеличивается на значение переменной **c**)

Мы уже видели, что в теле цикла можно использовать любые команды, в том числе — и команду выбора (if). Однако возможен и вариант «наоборот», когда цикл используется внутри команды выбора.

Рассмотрим в качестве иллюстрации только что сказанного следующее задание: *«Напиши класс, который принимает с клавиатуры целое число и выводит на экран все целые числа между 0 и введенным числом — начиная с наименьшего возможного числа»*.

Это означает, что, например, если с клавиатуры будет введено значение 6, то на экран надо вывести серию **1 2 3 4 5;** а если с клавиатуры будет введено значение -6, то на экран надо вывести **-5 -4 -3 -2 -1**.

Из этого следует, что, в общем случае (будем считать, что значение, вводимое с клавиатуры, принимает переменная **x**), надо выводить на экран либо серию **1..x-1** (если с клавиатуры введено положительное значение), либо серию **x+1…-1** (если с клавиатуры введено отрицательное значение).

Вот один из возможных вариантов решения:

```
import java.util.*;
class primer_for4
{
static Scanner reader =  new Scanner(System.in);
public static void main(String[] args)
{
 int a=reader.nextInt();
 if (a>0)
  for (int i=1; i<a; i++)
      {
         System.out.print(i+" ");
      };
 if (a<0)
  for (int i=a+1; i<0; i++)
      {
         System.out.print(i+" ");
      };
}
}
```

Раздел 16

Вычисления с использованием команды повтора — Цикл for

Команды повтора оказываются очень эффективными для производства самых разных проверок и вычислений, и в первую очередь таких, которые требуют повторяющихся, многократных действий.

Наиболее распространенные среди таких действий:

1. Подсчет количества значений, соответствующих определенному условию, — в более профессиональных терминах это называют использованием *«переменной-счетчика»*.
2. Подсчет суммы значений, соответствующих определенному условию — в более профессиональных терминах это называют использованием *«переменной-сумматора»*.
3. Вычисление величин, которые можно вычислить методами определения суммы или произведения (сумма элементов последовательности, факториал, целая степень, произведение серии значений и так далее).

Использование «счетчика случаев» в цикле

В этой группе задач — такие, в которых требуется определять, сколько раз из общего числа случаев повторяется выполнение какого-то определенного случая, какого-то определенного условия.

Например: с клавиатуры вводится серия чисел и надо подсчитать, сколько в этой серии было чисел положительных. Или: посчитать общее число делителей введенного с клавиатуры значения. Или: какое количество учеников класса согласны поехать на экскурсию. И так далее…

Вот как раз для такого подсчета «количества случаев» и используется переменная-счетчик. При этом очень и очень важно не путать ее с переменной, которая будет выполнять функцию другого счетчика, а именно — счетчика повторов (проходов) цикла.

Счетчик повторов — используется в заголовке цикла (как правило).
Счетчик случаев — **не** используется в заголовке цикла (как правило).

В отношении переменной-счетчика случаев действуют следующие правила:

- Ее следует инициализировать (то есть установить начальное значение) **до** начала цикла.
- Внутри цикла ее значение, как правило, увеличивается на 1.

Разберем пример класса с использование «счетчика случаев».

Задание для класса будет выглядеть следующим образом: «*Напиши класс, который генерирует и выводит на экран серию из 10 случайных двузначных положительных чисел — и определяет, сколько значений в этой серии были четными. Например, в серии 13 45 32 67 88 42 41 11 94 63 имеется 4 случайных числа (32, 88, 42 и 94*).

Один из возможных вариантов решения:

```
import java.util.*;
class primer_for5
{
static Scanner reader =  new Scanner(System.in);
public static void main(String[] args)
{
  int x, count=0; //счетчик случаев count устанавливается в начальное
                  //значение, равное 0
  for (int i=1; i<=10; i++)  //цикл из 10 повторов, счетчик повторов —
                  //переменная i
    {
      x=(int)(90*Math.random()+10); //генерируется случайное целое
                                    //двузначное положительное число
      System.out.print(x+" "); // случайное число выводится на экран
      if (x%2==0)
      count++; //если случайное число — четное, счетчик случаев
      //увеличивается на 1
    }
  System.out.println(" Число четных чисел в серии="+count); // Итоговое
  // значениепеременной-счетчика случаев выводит на экран только(!)
  //после завершения цикла
}
}
```

Обратите внимание, что команда «увеличить на 1 счетчик случаев» (**count++;**) не записана в фигурных скобках: это допускается в тех случаях, когда блок команд, которые надо выполнить, в случае если условие верное, содержит только одну-единственную команду.

Разумеется, при необходимости можно использовать не одну, а две, три и даже более переменных-счетчиков. Однако в тех случаях, когда необходимое для реализации поставленной задачи количество переменных-счетчиков становится больше 3–4-х, используются не отдельные переменные, а так называемый массив счетчиков — о таких инструментах речь пойдет в последних разделах курса.

Раздел 17

Вычисления с использованием команды повтора — Цикл for (продолжение)

Очень большую группу задач составляют те, в которых есть необходимость в подсчете не **количества** значений, соответствующих определенному условию, а **суммы** таких значений. В этих случаях принято говорить, что следует использовать *сумматор* (переменную-сумматор). Это могут быть задачи, в которых надо определить общий вес товаров, доставляемых на склад, или общую сумму, которую должен заплатить покупатель в магазине за покупки, или среднюю оценку класса (а для подсчета среднего арифметического ведь надо сначала подсчитать сумму оценок), и так далее.

Использование «сумматора» в цикле

Работа с переменной-сумматором в цикле похоже на работу со «счетчиком случаев» (о чем мы достаточно подробно говорили в предыдущей теме). Сходство это заключается в первую очередь в том, что сумматор, как и «счетчик случаев», необходимо обязательно обнулить перед началом использования (как правило — до начала цикла).

Однако есть и чрезвычайно существенная разница между использованием сумматора и «счетчика циклов»:

- «счетчик случаев» в каждом «подходящем случае» увеличивается, как правило, на 1;
- «сумматор» же при каждом «подходящем случае» увеличивается на некоторое значение (как правило — на значение, связанное с сутью задачи).

Рассмотрим ситуацию: покупатель в магазине расплачивается за 30 покупок, и надо определить количество покупок, стоимость каждой из которых превышает 500 рублей, а также определить общую сумму всех сделанных им покупок.

Примем решение, что в классе используется переменная-счетчик случаев **count**, а также переменная-сумматор **sum**.

Перед началом цикла обе переменные необходимо обнулить, то есть записать:

```
count=0;
sum=0;
```

В теле же цикла (он будет повторяться 30 раз — по числу покупок), подсчет количества товаров, стоимость которых превышает 500 рублей, будет выполняться командой (решим, что переменная **x** содержит значение, указывающее на стоимость товара в каждой из покупок):

```
if (x>500) count++;
```

Это означает, что мы используем переменную-счетчик для подсчета количества соответствующих покупок.

А вот подсчет общей стоимости всех купленных товаров будет выполняться командой:

```
sum=sum+x
```

Это означает, что значение переменной-сумматора **sum** каждый раз (как видите, нет в этой команде никакого условия) увеличивается на значение переменной **x**.

Кстати, эту команду можно записать в форме, принятой в языке Java:

```
sum+=x
```

В итоге класс, о котором мы говорим, может выглядеть так:

```
import java.util.*;
class primer_for6
{
static Scanner reader =  new Scanner(System.in);
public static void main(String[] args)
{
 int x, count=0, sum=0;
 for (int i=1; i<=30; i++)
   {
     x=reader.nextInt();
     if (x>500) count++;
     sum=sum+x
   }
 System.out.println(" Количество дорогих товаров="+count);
 System.out.println("Общая сумма закупки="+sum);
}
}
```

Разумеется, далеко не всегда переменная-сумматор увеличивается при каждом повторе (проходе) цикла; это, как и увеличение «счетчика случаев», может происходить только при выполнении определенного условия (или нескольких условий).

Например, надо реализовать в языке Java следующее задание: принимать с клавиатуры суммы, которые клиент вкладывает на свой счет в банке или снимает с этого счета. В таком классе (программе) изменение сумматора (он будет показывать, например, состояние счета после каждой операции или общие суммы снятых или вложенных денег, или общую сумму операций) или сумматоров может происходить и без проверки условий, и при проверке разных условий. В частности, раздельные переменные-сумматоры могут использоваться, если необходимо отдельно учитывать случай, когда сумма снята со счета, а отдельно — случай, когда она, наоборот, вложена на счет.

Следует иметь в виду, что использование в классе более 3-4 сумматоров (как и соответствующего числа счетчиков случаев) — совершенно нерационально. Для большего числа сумматоров или счетчиков следует использовать массивы сумматоров и массивы счетчиков; но о них речь пойдет в будущих разделах курса.

А пока разберем поподробнее упомянутый только что пример с банком.

Клиент за месяц произвел 28 операций со своим счетом в банке.

Напишите класс, который принимает с клавиатуры значение, равное сумме на счету клиента в начале месяца, а затем — 28 значений, соответствующих проведенным клиентом операциям со счетом. Положительные значение соответствуют внесению суммы на счет, отрицательные — снятию суммы со счета.

Следует определить следующие итоговые данные:

- *сумму на счету в конце месяца;*
- *сумму внесенных (не снятых!) на счет денег;*
- *количество операций по снятию денег со счета.*

Общие соображения:

- итоговая сумма будет определяться переменной-сумматором, начальным значением которой будет не 0, как обычно, а принимаемое с клавиатуры значение (начальное состояние счета, поскольку не обязательно и не очевидно, что на счету клиента в начале месяца был ноль...);
- сумму внесенных сумм будет фиксировать вторая переменная-сумматор, которую вначале надо будет обнулить;

- количество операций по снятию денег со счета будет фиксировать переменная-счетчик случаев, которую вначале надо будет обнулить;
- в классе будет цикл на 28 повторов — по числу операций, которые произвел клиент.

Код класса:

```
import java.util.*;
class primer_for6
{
static Scanner reader = new Scanner(System.in);
public static void main(String[] args)
{
 int x, count=0, sum1=0, sum;
 x=reader.naxtInt(); // начальное состояние счета
 sum=x;
for (int i=1; i<=28; i++)
  {
    x=reader.naxtInt();
    sum=sum+x;
    if (x>0) sum1+=x;
    if (x<0) count++;
  }
 System.out.println("На счету в банке="+sum);
 System.out.println("Общая сумма внесенных на счет денег="+sum1);
 System.out.println("Число операций по снятию денег="+count);
}
}
```

Раздел 18

Определение минимума и максимума в циклах — Цикл for

Отдельной группой стоят задачи, в которых требуется найти наибольшее или наименьшее значение в некоторой серии значений. Многочисленность класса этих задач связана, в первую очередь, с многочисленностью жизненных ситуаций, в которых требуется определить наименьшее или наибольшее значение.

Однако мы начнем не с нахождения наибольшего или наименьшего (максимума или минимума) значения в **серии** значений, а с нахождения таковых для **двух чисел.**

Существует несколько способов для достижения этой цели.

Во-первых, можно использовать оператор выбора (if). Предположим, что имеются две переменных, **num1** и **num2**, содержащие некоторые значения, а нам предстоит найти наибольшее из пары этих значений.

Для этого можно использовать конструкцию **if... else**:

```
if (num1>num2) max=mun1;
else max=num2;
```

При исполнении этой команды выбора переменная **max** либо получит

- либо значение от переменной **num1**, если в ней содержится большее из двух значений;
- либо значение от переменной **num2**, если максимум — в этой переменной.

Существует и другой вариант использования конструкции **if... else**:

```
max=num2;
if (num1>num2) max=mun1;
```

В этом случае переменная **max** получает еще до проверки «кто тут больше кого» значение от переменной **num2**. В случае если в переменной **num1** все-таки окажется значение, большее уже занесенного в **max**, с помощью команды выбора будет произведена замена значения в переменной **max**; в противном случае замена не производится. Так или иначе, но в перемен-

ной **max** после выполнения этих двух команд будет зафиксировано наибольшее из двух значений, записанных в переменных **num1** и **num2**.

Во-вторых, можно использовать методы из библиотеки Math, с которыми мы познакомились на одном из предыдущих занятий:

Mat.max(a,b)

Это будет выглядеть так:

max=Math.max(num1, num2)

Разумеется, совершенно аналогичным образом будут выглядеть операторы, которые используются для определения наименьшего из пары значений (минимума) — с той разницей, что вместо знака «больше» (>) надо будет использовать знак «меньше» (<), а вместо метода Math.max(a,b) надо будет использовать метод Math.min(a,b).

В заданиях, где требуется найти наибольшее или наименьшее значение (максимум или минимум) не для двух, а для серии чисел, имеются несколько важных особенностей:

- первоначальное значение в переменную, которая должна в итоге получить искомый максимум или минимум, следует занести **до** начала цикла;
- соответственно, число повторов (проходов) цикла должно быть **на одно** меньше того, что описано в условии;
- каждый раз при повторе цикла будет производиться проверка очередного значения (в большинстве случаев они содержится в одной и той же переменной, что существенно упрощает код).

Первое из этих действий особенно важно: оно необходимо, чтобы гарантировать получение «целевой» переменной (той, которая в итоге должна содержать максимум или минимум) начального значения.

Дело в том, что в определенных комбинациях кода язык Java может не допустить выполнения класса (уже на этапе компиляции класса), поскольку «нет твердой гарантии», что эта «целевая» переменная получит хоть какое-то значение.

Присваивать же такой переменной просто какое-то значение, безо всякой связи с теми значениями, которые в дальнейшем будут проверяться — чревато ошибкой. В частности, одна из наиболее распространенных ошибок — обнулять такую переменную, то есть присваивать ей значение 0. Но вот в случае если среди будущих данных все значения окажутся отрицательными, то вместо реального наибольшего из них (которое, например, может оказаться равным, -5), мы получим в качестве максимума

значение 0... Сами понимаете, никакого отношения к истинному максимальному значению среди отрицательных это не имеет...

В качестве примера приведем решение следующего задания: «*напишите класс, который принимает с клавиатуры 25 целых чисел и определяет наибольшее и наименьшее значения в этой серии чисел*».

```
import java.util.*;
class primer_for16
{
static Scanner reader =  new Scanner(System.in);
public static void main(String[] args)
{
        int max, min, x;
        x=reader.naxtInt();
        max=x;
        min=x;
         for (int i=1; i<=24; i++)
         {
                x=reader.naxtInt();
                max=Math.max(max, x);
                min=Math.min(min, x);
         }
        System.out.println("Наибольшее значение в серии="+max);
        System.out.println("Наименьшее значение в серии="+min);
}
}
```

Обратите внимание, что итоговый результат выводится на экран уже **после** завершения цикла.

Теперь усложним задание: требуется не только найти наибольшее и наименьшее значения в серии чисел, но и определить, на каком месте в серии были эти значения.

Например, для серии 56, 12, 5,87, 39, 62, 94, 4, 15 результаты будут следующими:

- максимум равен 94;
- минимум равен 5;
- максимум расположен на 7-м месте в серии;
- минимум расположен на 3-м месте в серии.

Важное замечание: пока мы будем исходить из предположения, что во всей серии максимальное и минимальное значения встречаются каждое только по одному-единственному разу! Это, разумеется, совершенно не

обязательно и вовсе не очевидно, но мы пока будем исходить из этого предположения для простоты решения. В дальнейшем, разумеется, можно и не использовать этого предположения, но при этом и условие задачи будет выглядеть иначе.

Для этого мы введем две дополнительных переменных, mesto_max и mesto_min:

- первоначально, еще до начала цикла, занесем в них значение 1. Таким образом, мы предполагаем, что искомые значения находятся на первом месте в серии. Разумеется, в дальнейшем мы проверим эти предположения и либо оставим их без изменений, либо изменим;
- добавим в цикл две команды выбора, которые будут изменять значение переменных mesto_max, mesto_min — в том случае, если в цикле будет обнаружено **очередное** наибольшее или наименьшее значение.

Теперь решение будет выглядеть следующим образом:

```
import java.util.*;
class primer_for16
{
static Scanner reader =  new Scanner(System.in);
public static void main(String[] args)
{
        int max, min, x;
        x=reader.naxtInt();
        max=x;
        min=x;
         for (int i=1; i<=24; i++)
         {
                x=reader.naxtInt();
                max=Math.max(max, x);
                if (max==x) mesto_max=i;
                min=Math.min(min, x);
                if (min==x) mesto_min=i;

         }
        System.out.println("Наибольшее значение в серии="+max+" на
        позиции "+mesto_max);
        System.out.println("Наименьшее значение в серии="+min+" на
        позиции "+mesto_min);
}
}
```

Раздел 19 (часть 1)

Структура и использование цикла while

Несмотря на то, что мы еще не изучили всех возможностей использования цикла for, мы перейдем к изучению нового вида команды повтора, к циклу while.

Как вы помните, цикл for — это возможность организации такого цикла, в котором число повторов (проходов) можно определить (установить) заранее.

Однако существует масса задач, ситуаций, в которых, с одной стороны, требуется организация цикла, а с другой стороны — в принципе отсутствует возможность заранее знать, сколько повторов потребуется выполнить.

Иными словами, мы должны иметь возможность организовать так называемый «гибкий цикл», то есть такой, в котором число повторов не определяется установленными заранее какими-то значениями (в том числе — и значениями, введенными с клавиатуры для каких-то переменных). Нам нужен цикл, в котором после каждого завершения очередного повтора (прохода), будет выполняться проверка: а выполнить ли еще один повтор (проход).

Обратите внимание: каждый раз проверка выполняется заново — и каждый раз на **только еще один** дополнительный повтор (проход).

Для сравнения этих двух видов циклов можно привести ситуацию из жизни: учитель музыки в качестве домашнего задания может велеть ученику «сыграть пьесу 8 раз» — или «сыграть пьесу до тех пор, пока не станет получаться, как надо».

В первом случае заранее известно число повторов, а во втором — в принципе нет никакой возможности знать, сколько же раз ученик будет повторять пьесу: один, пять, восемь, пятнадцать?.. Дело в том, что, закончив в очередной раз исполнение пьесы, ученик «проверяет условие» — «получилось ли, как надо?» И только после получения результата проверки, в зависимости от этого результата проверки, **каждый раз** решает, играть ли пьесу еще **один**, дополнительный раз.

Структура цикла while

Как и цикл for, цикл while тоже состоит из двух частей:

- заголовка цикла;
- тела цикла.

Заголовок цикла while состоит из двух частей:

- служебного слова while;
- условия продолжения цикла, заключенного в скобки.

Приведем несколько примеров таких заголовков:

- while (x>0)
- while (a!=b)
- while (sum<66)

Обратите внимание, что условие, размещенное в заголовке, проверяется каждый раз **перед** очередным повтором цикла.

Очень важно иметь в виду, что условие, размещенное в заголовке, является условием **продолжения** цикла. Это означает, что, если условие **выполняется**, цикл while делает еще один повтор (проход); а вот если условие это **не выполняется**, то цикл больше не повторяется и класс перейдет к выполнению команды, записанной после тела цикла.

Стоит обратить внимание, что цикл while, довольно заметно, похож, в смысле использования условия, на команду выбора (ветвления) if.

Поскольку в условии, которое записывается в заголовке цикла, используется, как правило, по крайней мере, одна переменная, **необходимо** инициализировать ее (присвоить ей значение) **еще до** заголовка цикла.

Опасность бесконечного цикла

К сожалению, в циклах while очень легко создать ситуацию, при которой количество повторов цикла становится бесконечным, то есть цикл повторяется снова и снова, и повторения эти в принципе никогда не прекратятся. А это означает, что класс просто «зависает», то есть фактически никогда не произойдет передача выполнения класса команде, следующей за циклом.

Например, если при проверке условия в заголовке цикла:

while (x>0)

окажется, что переменная **x** содержит положительное значение, цикл выполнится в первый раз. Но если внутри, в теле цикла, переменная **x**

не изменяется, следовательно, условие снова окажется верным, и цикл выполнится во второй раз... и условие снова окажется верным, и цикл выполнится в третий раз... в четвертый... в пятый... в шестой... и так без конца...

В результате возникает так называемый «бесконечный цикл» — или, точнее сказать, бесконечное повторение цикла.

Для того чтобы избежать подобной ситуации, следует в теле цикла обязательно изменять значение переменной, проверяющейся в условии.

Но и этого — недостаточно! Необходимо так изменять значение переменной, проверяющейся в условии, чтобы в какой-то момент проверка условия обнаружила бы, что оно неверное — и исполнение цикла завершилось бы.

Например, если фрагмент класса выглядит следующим образом:

```
int a=1;
    while (a>0)
    {
     a++;
     System.out.print(a+" ");
    }
```

то мы снова получим бесконечный цикл, так как значение переменной **a** все время увеличивается, увеличивается и увеличивается... И все время остается положительным... В результате чего условие, записанное в заголовке цикла, выполняется «всегда»... Снова получается бесконечный цикл, то есть бесконечное повторение цикла...

Рекомендации для использования цикла while

Таким образом, при использовании в классе цикла while следует соблюдать как минимум три основных правила:

1. Переменная, которая используется в условии заголовка цикла, должна получать значение до начала цикла.
2. Условие, входящее в состав заголовка цикла, является условием только еще одного повтора цикла.
3. Следует менять в теле цикла значение переменной, которая используется в условии заголовка цикла, но менять так, чтобы сохранялся шанс на прекращение повторов цикла.

Существует и еще одна очень существенная разница между циклами **for** и **while**.

При изучении цикла **for** мы говорили, что **не рекомендуется** изменять значение переменной-счетчика внутри (в теле) цикла. А вот в цикле while все как раз наоборот: **обязательно надо** изменять переменную, от которой зависит исполнение или неисполнение цикла в очередной раз.

Как определить, какой из циклов, for или while, следует использовать при решении?

Однозначного, простого, универсального правила не существует, но можно дать довольно простую рекомендацию: прочитав условие задания, постарайтесь понять, можно ли каким-то образом заранее определить количество повторов цикла. Если это возможно — стоит использовать цикл for, если такой возможности нет, — скорее всего, надо писать цикл while.

Пример с использованием «кода прерывания»

Рассмотрим следующий пример: «*Напишите класс, который принимает с клавиатуры целые числа. Ввод данных завершится, когда с клавиатуры будет введено значение –1. Класс должен выводить на экран количество значений, которые были введены с клавиатуры, но не засчитывать значение –1*».

Обратите внимание: фразы вроде «ввод завершится, когда...», «будет продолжаться, пока...», «до тех пор, пока» и им подобные — достаточно ясный намек на то, что число проходов цикла заранее неизвестно. Иными словами: надо будет использовать цикл while.

Кроме того, в классе нам понадобится переменная-счетчик — и очень важно не забыть сделать так, чтобы в последний проход по телу цикла переменная-счетчик ни в коем случае не увеличивалась. Иными словами, для серии 6, 9, 234, -9, 0, 6, -1 — значение переменной-счетчика должно быть равно 6, а не 7.

Решение может выглядеть так:

```
import java.util.*;
class primer_while1
{
static Scanner reader =  new Scanner(System.in);
public static void main(String[] args)
{
 int x, count=0;
 x=reader.nextInt();
 while (x!=-1)
 {
```

```
    count++;
    x=reader.nextInt();
  }
  System.out.println(" Количество значений в серии="+count);
}
}
```

Обратите внимание: если первое же значение, введенное с клавиатуры, будет равно –1 — цикл вообще **ни разу** не будет (точнее — не должен!) выполняться!

Это — одна из самых важных особенностей цикла while: этот цикл может ни разу не выполняться; вариант этот всегда следует иметь в виду.

В нашем примере имеется значение **–1**, при введении которого с клавиатуры цикл прекращается. В общем случае такое значение принято называть «кодом прерывания» (при его введение прерывается повторение цикла), «караульным» (караулит, когда в очередной раз надо «не пропустить» исполнение в тело цикла), zip-code (только не в смысле «почтовый индекс», а в смысле «код, который закрывает» — имеется в виду, что он закрывает исполнение цикла).

При этом совершенно не обязательно, чтобы условие в заголовке цикла содержало использование именно «кода прерывания»; в достаточно многих случаях это может быть и ограничение, налагаемое на значение одной из переменных.

Пример с использованием «ограничения значения»

Рассмотрим пример: «*Напишите класс, который генерирует серию однозначных положительных чисел до тех пор, пока сумма значений серии остается меньше 100. Класс должен выводить на экран количество значений в сгенерированной серии и их сумму*».

Решение может выглядеть так:

```
import java.util.*;
class primer_while2
{
static Scanner reader =  new Scanner(System.in);
public static void main(String[] args)
{
  int x=0, count=0, sum=0;
  while (sum<=100)
  {
```

```
    x=(int)(10*Math.random());
    count++;
    sum+=x;
  }
  sum-=x;
  count--;
  System.out.println("Сумма="+sum+" количество значений="+count);
}
}
```

В решении есть несколько особенностей, на которые очень важно обратить внимание:

1. В заголовке **while (sum<=100)** использовано условие, связанное с ограничением, налагаемым, в соответствии с условием задания, на сумму серии; и сумма эта, разумеется, должна быть в начале класса обнулена.
2. После окончания цикла, то есть после перехода исполнения к команде, которая расположена в классе **после** цикла, сумма уменьшается на значение последнего значения в серии, а переменная-счетчик уменьшается на 1. Это делается потому, что цикл прерывается **после** того, как сумма превысит ограничение, то есть значение 100.
3. Обратите внимание на то, что переменная **x** инициализирована значением 0. Это делается из-за того, что значения переменой **x** создаются **внутри** цикла, а после выхода из цикла переменную-сумму надо уменьшить на последнее значение переменной **x**. Однако, без инициализации этой переменной до начала цикла, еще на этапе компиляции класса возникает ошибка, смысл которой: «переменная **x** может не получить значения». Эта ошибка связана с тем, что язык Java очень придирчиво и требовательно относится к вероятности того, что переменная может остаться неинициализированной. А поскольку свои значения переменная **x** получает в теле цикла, которое, как может случиться, ни разу не будет исполняться, то язык Java, уже на этапе компиляции, требует не допустить даже вероятности такой ситуации. Вот и приходится инициализировать переменную хотя бы значением 0 еще до цикла; разумеется, что с таким же успехом ее можно было бы инициализировать любым другим значением.

Раздел 19 (часть 2)

Структура и использование цикла do... while

В языке Java существует вариант цикла while, который выглядит следующим образом:

- В заголовке используется служебное слово **do.**
- Тело цикла заключено в фигурные скобки.
- После фигурной скобки, закрывающей тело цикла, записывается служебное слово **while()** с условием в круглых скобках -- совершенного аналогично тому, как это делается в «обычном» цикле **while.**

Разница между «обычным» циклом **while** и циклом **do... while** — в том, что «обычный» цикл **while** может не выполняться **ни разу**, а цикл **do... while** обязательно выполняется как минимум в первый раз. И только после выполнения прохода проверяется условие — повторить исполнение цикла еще раз или нет.

Раздел 20

Вложенные циклы

Существуют ситуации, которые можно описать выражением «цикл в цикле» — то есть когда один повторяющийся процесс выполняется внутри другого повторяющегося процесса.

Например, дни недели повторяются снова и снова внутри годового цикла, который состоит из повторяющихся месяцев.

Еще один пример из «истории времен»: секунды повторяются снова и снова в часовом цикле, где минуты повторяются друг за другом.

Если мы выпишем числа от 100 до 999, то легко увидим, что цифры в позиции «число единиц» повторяются снова и снова в каждом десятке, да и цифры в позиции «число десятков» повторяются снова и снова в каждой сотне.

Думаю, что каждый может привести дополнительные примеры ситуаций, когда одно повторяющееся действие выполняется внутри другого повторяющегося действия: это называется «цикл в цикле» или «вложенные циклы». Вложенные друг в друга, то есть когда один цикл (или даже несколько циклов, идущих один за другим, по отдельности) вложен в другой цикл, внутрь другого цикла.

Вот, например, блок (группа команд, часть программного кода), который проверяет, является ли переменная **num** — простым числом. Делается это с помощью подсчета количества делителей этой переменной: начиная с 2 и до «середины» **num** (нет смысла искать делители, которые больше половины проверяемого числа, потому что таких делителей в принципе быть не может).

```
int count=0, num=reader.nextInt();
for (int i=2; i<=(num/2); i++)
        if (num%i==0) count++;
```

В случае если переменная **num** действительно является простым числом, значение переменной **count** после исполнения этого блока останется равным **0**, потому что ни одного делителя у простого числа нет.

Разумеется, у каждого числа вообще, и, в том числе, у простого числа, есть два делителя: это единица и само число. При желании только что

приведенный блок можно было бы записать, с учетом только что сказанного, несколько иначе:

```
int count=0, num=reader.nextInt();
for (int i=1; i<=num; i++)
        if (num%i==2) count++;
```

понятно, что в этом случае значение переменной **count** после окончания проверки должно быть равно не 0, а 2 (единица и само число — два делителя любого числа).

А теперь усложним задачу: надо найти и вывести на экран все простые двузначные числа. Для этого мы организуем цикл по всем двузначным числам (можно начать с 13, но «для порядка» мы все же начнем с 10), а внутри этого цикла, то есть на каждое двузначное число, задействуем приведенный выше блок, в котором тоже есть цикл! Вот так и получится у нас «конструкция», в которой один цикл будет вложен в другой: внешний цикл будет перебирать одно за другим двузначные числа, а внутренний цикл будет перебирать для каждого такого числа все его возможные делители. Как вы понимаете, реализацией слова «перебирать» является не что иное, как использование цикла.

```
int count=0;
for (num=10; num<=99; num++)
{
        for (int i=2; i<=(num/2); i++)
                if (num%i==0) count++;
        if (count==0)
                System.out.println(num);
        count=0;
}
```

Работать эта конструкция будет так:

- Сначала переменная **num** во внешнем цикле принимает очередное значение (первым будет значение 10, потом значение 11, далее — значение 12... и так до значения 99).
- Затем полностью выполняются **все** проходы (повторы) во внутреннем цикле, во время которых числа проверяются на то, являются ли они делителем значения переменной **num**.
- В случае если в переменной-счетчике осталось начальное значение 0, на экран выводится значение переменной **num**, потому что оно, это значение, действительно является простым числом.
- Очень важный момент, который (особенно на первых порах), ускользает от внимания: перед повтором внешнего цикла снова обнуляется переменная-счетчик. Делается это потому, что для каждого нового числа все проверки на «делитель — не делитель» надо будет выполнять заново.

Обратите внимание: во внешнем и внутреннем циклах мы использовали разные переменные-счетчики; делается это для того чтобы не происходила путаница при счете повтором каждого из циклов.

Кстати, использование и для внешнего, и для внутреннего цикла одной и той же переменной-счетчика повторов, является чрезвычайно распространенной ошибкой.

Разберем еще один пример: «*Напишите класс, который выводит на экран прямоугольник из символов #. При этом размеры прямоугольника, то есть количество строк и количество знаков в каждой строке указывается введением с клавиатуры двух чисел: первое — количество строк, второе — количество знаков в каждой строке*».

Это значит, что, при введении с клавиатуры значений 4 и 7, на экран должен выводиться вот такой прямоугольник (4 строки по 7 знаков в каждой строке):

```
#######
#######
#######
#######
```

Для того чтобы «организовать» нужное количество строк, мы используем внешний цикл, а чтобы гарантировать вывод нужного количества знаков в каждой строке, — мы используем внутренний цикл. Кроме того, чтобы после окончания очередной строки из символов # осуществлялся переход к новой строке, после окончания внутреннего цикла мы используем команду «пустого» вывода на экран.

Мы приводим фрагмент такого класса:

```
int strok, znakov;
strok=reader.nextInt();
znakov==reader.nextInt();
for (j=1; j<=strok; j++)
{
        for (i=1; i<=znakov; i++)
                System.out.print("#");
        System.out.println();
}
```

Раздел 21
Методы

На определенном этапе классы становятся все более и более сложными и, одновременно с увеличением сложности, разрастаются с точки зрения количества строк кода. Кроме того, в них начинают появляться части (блоки), которые повторяют друг друга, что просто нелогично и непродуктивно.

На этом этапе стоит начать использовать в классе новый инструмент, который называется «методы»; вот им и будет посвящен этот раздел.

Что такое метод

Вместо того чтобы писать класс как один большой блок команд, можно выделить в нем отдельные (как правило — повторяющиеся, но не обязательно повторяющиеся) группы кода, группы команд. Вот эти группы и можно оформить как отдельный от главного метода блок, который и будет называться методом.

Мы уже знаем, что любой, даже пустой, класс на языке Java должен содержать как минимум один метод, который называется «главный метод»; по аналогии с ним и остальные части, не входящие в главный метод, тоже называют методами.

Приведем пример: «*Напишите класс, который выводит на экран все целые трехзначные положительные числа, в которых квадрат средней цифры равен сумме квадратов крайних цифр*».

Одно из возможных решений выглядит так:

```
class primer_method1
{
 public static void main(String[]args)
 {
  int x100,x10,x1,s1,s2;
```

```
  for (int x=100; x<=999; x++)
   {
       x100=x/100;
       x10=(x/10)%10;
       x1=x%10;
       s1=x100*x100+x1*x1;
       s2=x10*x10;
       if (s2==s1) System.out.print(x+" ");
   }
 }
}
```

Теперь возьмем четыре первых команды в теле цикла и вынесем их в отдельный блок вне главного метода; теперь наше решение выглядит иначе:

```
class primer_method1
{
 public static void main(String[]args)
 {
  int x2,s2;
  for (int x=100; x<=999; x++)
   {
       x2=(x/10)%10;
       s2=x2*x2;
       if (s2==s1(x)) System.out.print(x+" ");
   }
 }

 public static int s1(int x)
 {
       int x100,x10,x1,s;
       x100=x/100;
       x10=(x/10)%10;
       x1=x%10;
       s=x100*x100+x1*x1;
       return s;
 }
}
```

Разберем с подробными объяснениями новый вариант решения.

class primer_method1 **{**	Заголовок класса и открывающая тело класса фигурная скобка остались на месте без изменений
public static void main(String[]args) **{**	Заголовок главного метода и открывающая тело главного метода фигурная скобка остались на месте без изменений
int x2,s2;	Изменился список переменных, которые используются в главном методе, поскольку часть переменных ушла в отдельный метод (об этом — ниже)
for (int x=100; x<=999; x++) **{**	Заголовок цикла и открывающая тело цикла фигурная скобка остались на месте, без изменений
x2=(x/10)%10; **s2=x2*x2;**	В теле цикла используется переменная **x2**, в которую заносится вторая цифра числа (число десятков), и переменная **s2**, в которую заносится квадрат этой второй цифры
if (s2==s1(x)) System.out.print(x+" ");	Очень важно: в команде выбора сравнивается переменная **s2** с «чем-то», что фактически не существует(!) в главном методе: s1(x)... Как в этом случае поступает Java? Начинает проверять класс, есть ли в какой-то его части упоминание о «чем-то», что называется **s1**. В нашем случае сразу за главным методом, в продолжении класса обнаруживается блок, который действительно так называется
}	Скобка, закрывающая цикл
}	Скобка, закрывающая тело главного метода
public static int s1(int x)	Вот он, блок, который называется **s1**! На языке Java такие блоки называются методами. Любой метод (в том числе и наш, **s1**), как и главный метод, имеет заголовок и тело.

Окончание таблицы

	Заголовок содержит служебные слова static public, после которых указывается *тип значения*, который этот метод вернет «туда, откуда его вызвали». Поскольку в нашем случае вызов состоялся из команды выбора, то и возвращаться значение будет в главный метод, в условие команды выбора. После типа метода записывается его название (**s1** — в нашем случае). За названием, в круглых скобках, указывается список параметров, которые получает метод из «точки вызова». Вернитесь к команде выбора и обратите внимание, что в условии метод **s1** вызывается так, что во время вызова ему посылается значение переменной **x**. Если метод получает несколько параметров, все они записываются в заголовке метода со своими типами и именами
{	Скобка, открывающая тело метода
int x100,x10,x1,s; **x100=x/100;** **x10=(x/10)%10;** **x1=x%10;** **s=x100*x100+x1*x1;**	В теле метода мы записали те переменные и команды, которые используются для определения первой и последней цифр, а также для вычисления суммы их квадратов
return s;	Поскольку наш метод должен возвращать «туда, откуда его вызвали» (в нашем случае — в условие команды выбора, расположенной в теле главного метода) значение, в методе обязательно должна быть команда **return**, после которой указывается переменная, возвращающая значение. Очень важно: тип этой переменной и тип метода **должны** совпадать
}	Скобка, закрывающая тело метода s1
}	Скобка, закрывающая тело класса

Основные правила для работы с методами

Давайте перечислим основные правила написания использования методов.

1. Метод пишется как отдельный блок внутри класса, со своим заголовком (заголовок метода) и телом (тело метода).
2. Для метода, который возвращает в «точку вызова» какое-то значение, в заголовке надо указывать тип, соответствующий типу возвращаемого значения. В методе, который ничего не возвращает (например, метод, который что-то подсчитывает и выводит на экран результат подсчета, и только), вместо типа пишется служебное слово void (мы приведем такие примеры).
3. У метода должно быть имя (указывается в заголовке), которое дается по «правилам имен», принятым в языке Java.
4. Если метод получает из «точки вызова» какие-то параметры, то в заголовке метода, в круглых скобках, указываются соответствующие переменные с соответствующими типами (примеры — чуть ниже).
5. Вызов метода в «точке вызова» осуществляется с помощью «упоминания» имени метода с перечислением после имени, в круглых скобках, отсылаемых в метод переменных.

Вызов метода

Разберем следующий пример:

Даны объявления нескольких переменных и заголовки двух методов:

int a, b, f;
double c, d;
Boolean e;
static double test1(double x, int y, boolean z)
static void test2(double x, int y, boolean z)

Для каждого из приведенных ниже вызовов методов (можно использовать и формулировку «обращений к методам») следует указать, верный этот вызов или нет, и объяснить ответ.

Однако прежде мы хотим обратить ваше внимание на два следующих момента:

- в примере используются переменные логического типа (**boolean**), которые могут принимать только одно из двух возможных значений: либо **true**, либо **false**;
- использование служебного слова public — не обязательно.

Вызов метода	Ответ и объяснение
a=test1(5.3, a, e);	Вызов неверный: метод возвращает значение типа **double**, которое нельзя присвоить переменной типа **int**
c=test1(5.3, d, e);	Вызов неверный: на втором месте в скобках указана переменная типа **double**, которая не может передать значение переменной типа **int**, указанной на втором месте в заголовке метода
d=test1(5.3, a, e);	Вызов верный, все необходимые соответствия типов соблюдены
d=test1(1, 2, **true**);	Вызов верный, все необходимые соответствия типов соблюдены
if(test1(d, b, e)==f) test2(d, c+b, e);	Вызов неверный: в условии сравнивается переменная типа **int** и возвращаемое методом значение, которое относится к типу **double**
f=test2(c, 5, e);	Вызов верный, все необходимые соответствия типов соблюдены
c=test1(c, d, **false**);	Вызов неверный: на втором месте в скобках указана переменная типа **double**, которая не может передать значение переменной типа **int**, указанной на втором месте в заголовке метода

Соответствие вызова метода и заголовка метода можно, с определенными оговорками, сравнить с правилами на соревнованиях по эстафете:

- Участники забега должны быть одного «типа» (возраст, пол и так далее): это как бы соответствие типа метода и типа переменной, которой он, например, возвращает значение.
- Палочку можно передавать только участнику своей команды: это как бы соответствие между типами переменных или значений в вызове и типами переменных или значений в заголовке метода.

Разберем еще один пример:

Даны объявления переменных и заголовок метода:

int a, x;
public static void queeze(int c, int a, double r)

Для каждого из приведенных ниже вызовов метода следует указать, верный ли этот вызов или нет, и объяснить ответ.

Вызов метода	Ответ и объяснение
queeze (a , 7);	Вызов неверный: метод должен получать три параметра, а в вызове указаны только два
queeze (x, 8/3 , 5/0.1);	Вызов верный: есть полное соответствие между типами параметров и значений в вызове и в заголовке (не забудьте, что 8/3 возвращает не результат деления, а частное, получаемое при делении 8 на 3)
queeze (4 , 5 , 1.11);	Вызов верный: есть полное соответствие между типами параметров и значений в вызове и в заголовке
System.out.println(queeze((x+a), 10, 3.14));	Вызов неверный: метод типа **void** не возвращает никакого значения, поэтому просто нечего выводить на экран

Язык Java не предъявляет совершенно никаких ограничений на то, где внутри класса написаны методы, по отношению к главному методу — до главного метода или после главного метода. Каждый может выбрать такой подход к выбору места для методов, который ему, пишущему коды, кажется более удобным.

Раздел 22
Массивы

Одно из самых важных применений компьютерных технологий — хранение и обработка большого количества данных.

Например, при работе с данными «Оценки учеников класса» существует не только необходимость вычислять среднюю арифметическую оценку всего класса, но и необходимость определять число учеников, получивших больше или меньше средней оценки, количество учеников, справившихся с контрольной, и число тех, кто получил «неуд»... и прочая, и прочая, и прочая...

Самое главное, что для вычисления всех этих статистически важных результатов необходимо в первую очередь уметь хранить большое количество данных (в приведенном примере — оценки). Однако при этом закономерно возникает проблема: а как, каким образом, где, в каком количестве переменных хранить все эти данные. Любая переменная из тех, с которыми мы до сих пор имели дело, предназначена для хранения только и только одного единственного значения в каждый данный момент. Означает ли это, что для хранения 30 оценок учеников класса (хотя бы только по одной оценке на ученика) потребуется 30 разных переменных, по одной на каждую оценку? Но главное даже не в этом; главное в том, что даже такое, крайне неудобное решение — это только вершина целого «айсберга» проблем: ведь для определения даже простого максимума потребуется какое-то совершенно «невменяемое» количество операторов выбора...

Разумеется, для решения такого рода задач, то есть задач, связанных с необходимостью хранить и обрабатывать большое количество значений, сохраняя при этом некие разумные размеры класса, те «простые» переменные, с которыми мы до сих пор имели дело, совершенно не годятся.

Для решения такого рода задач используются специальные варианты переменных, которые называются массивами (array).

Что такое массив

Избегая профессиональных и очень непростых, не всегда понятных для начинающих определений массива, попробуем описать его главную особенность максимально простыми словами.

Выходит это примерно так: массив позволяет под **одним** именем хранить и использовать одновременно **много** значений.

Вспомним: «простая» переменная под **одним** именем позволяет хранить одновременно только **одно** значение.

Визуально принято изображать массив в виде таблицы, в каждой ячейке которой записано какое-то значение.

Наиболее простой вид массива — одномерный массив, и визуально его принято изображать в виде таблицы, в которой есть только одна строка, разделенная на некоторое количество пронумерованных ячеек. Номер ячейки принято называть *индексом ячейки*.

В массиве хранятся данные одного и того же типа (массив целых чисел, массив десятичных дробей, массив символьных значений и так далее) и эти разные значения, одновременно хранящиеся в разных ячейках массива, в принципе совершенно не должны быть связаны друг с другом и совершенно не должны влиять друг на друга. То есть они могут и быть связаны, и влиять — могут, но не должны.

В литературе часто можно встретить сравнение массива с многоквартирным домом: все квартиры имеют **одинаковое имя** (одинаковый адрес типа «город—улица—дом»), но в разных ячейках («номера квартир») может проживать разное количество людей. При этом вполне возможно, что количество людей в разных квартирах никак не связано друг с другом — а, может быть, как-то и связано…

Массив дает прямой доступ к данным любой своей ячейки для использования этих данных: для вычислений, занесения данного в ячейку, изменения данного в ячейке, вывода данного ячейки на экран и других.

Этих представлений о массивах, простых и весьма неполных, вполне достаточно для того чтобы начать с ними работать; а более подробную, более профессиональную информацию о массивах мы представим позже.

Пока мы будем использовать **только** одномерные массивы — это замечание важно потому, что мы не всегда будем использовать фразу «одномерный массив», а иногда, для краткости, использовать просто термин «массив» (имея в виду все тот же одномерный массив).

Объявление массива

Сравните два объявления переменных:

```
int a;
int [] b;
```

В первой строке объявляется «обычная» переменная, которая имеет имя **a** и предназначена для работы с целочисленными значениями (значениями типа **int**).

Во второй строке объявляется переменная типа массив целых чисел (можно говорить — «объявляется массив»), которая имеет имя **b** и предназначена для работы с целочисленными значениями (значениями типа **int**). О том, что вторая переменная является именно массивом, говорят две квадратные скобки, поставленные после указания типа данных.

Также допустим вариант, в котором квадратные скобки ставятся не после указания типа массива, а после его имени:

```
int  b[];
```

Очень важно помнить, что объявление массива еще не делает его доступным для работы; происходит это по той же причине, по которой одно только объявление «простой» переменной не делает ее доступной для работы. Какая может быть работа, если нет никакого значения!..

Инициализация массива

В отличие от «обычной» переменной, в которую при инициализации надо занести какое-то значение, массив можно инициализировать и несколько иначе, причем разными способами.

Первый способ инициализации массива

Для инициализации массива используется операция **new** и указание *размера массива*, то есть количества ячеек, которые массив будет содержать. Например, для массива **b**, который мы только что использовали в качестве примера, эта инициализация может выглядеть так:

```
b=new int[10];
```

После выполнения этой строки мы получаем возможность использовать одномерный массив размером 10 ячеек.

И вот тут мы сталкиваемся с одной очень важной, даже исключительно важной разницей между «обычными» переменными и массивами.

После инициализации «обычной» переменной можно совершенно спокойно говорить «в переменной хранится значение»:

```
int a=5;
```

Только приведенная запись означает, что под именем переменной **a** «скрывается» значение 5.

Массив же, по сути дела, «хранит» не значение (точнее — значения, так как предназначен он, в принципе, для работы под одним именем сразу с несколькими значениями), — массив хранит ссылку на ту область памяти, в которой записаны значения.

Кстати, это отличие вы, возможно, заметили уже в самом способе инициализации: в нем было использована операция **new**. Это такой элемент языка Java, который создает не «простые» переменные, имя которых напрямую связано со значением, а «объект ссылочного типа». Иными словами, создает указание на место, в котором, собственно, значение и хранится.

Мы пока не станем углубляться в специфику различий, но вы и сами увидите, насколько разными инструментами в языке Java являются переменные, которые массивы, и переменные, которые «простые».

Во-первых, можно объявить массив размером… 0!

```
int [] a=new int[0];
```

Разумеется, такая возможность сразу вызывает вопрос: а зачем нужен массив, размер которого равен 0, что в него можно поместить, для чего его можно использовать? Оказывается, при написании классов, которые используют сложные типы данных (объекты) есть немало ситуаций, когда массив нулевого размера как раз очень даже удобен. Но рассмотрение задач этого уровня выходит за рамки данного курса — и пока мы ограничимся констатацией факта: можно создавать массив нулевого размера.

Исключительно важной особенностью языка Java является возможность так задавать размер массива, чтобы никакие конкретные числа при этом не указывались вообще. Можно сказать, что с помощью этого способа размер массива задается «гибко», потому что при этом удается избежать указания заранее известного размера с помощью конкретного числа:

```
int x=reader.nextInt();
int [] a=new int[x];
```

И можно даже так:

int [] a=new int[reader.nextInt()];

В этом случае, как видите, в принципе нет никакой возможности(!) ответить на вопрос: какого размера (в числах!) будет массив?

Единственное(!), что в этом случае можно сказать: размер массива будет равен значению переменной **x**. А это, в свою очередь, означает, что запуская на исполнение один и тот же класс, мы можем создавать каждый раз массив иного размера.

Мы совсем не случайно употребили в предыдущем абзаце несколько восклицательных знаков: использование гибкого создания массивов — действительно очень сильный инструмент.

Следует только иметь в виду: после того, как размер массива будет установлен в виде числового значения (не важно, каким способом) — изменять размер массива еще раз уже нельзя.

Второй способ инициализации массива

Существует еще один способ объявить и инициализировать одномерный массив: сразу с помощью прямого занесения в него набора конкретных значений уже на стадии инициализации. Это делается с помощью использования в инициализации пары фигурных скобок, в которых через запятую записаны заносимые в массив значение.

int [] c={7,7,3,9,6,-3,0,1};

Обратите внимание, что при таком способе инициализации массива вообще нет необходимости указывать в явном виде размер массива: количество значений, записанных в фигурных скобках, автоматически устанавливает и размер массива.

Разумеется, аналогичным образом можно объявлять и инициализировать и массивы для работы с другими типами значений:

double [] c=new double[7];
boolean [] k=new boolean[2];
String [] c=new String[12];

Нумерация ячеек массива

В языке Java нумерация ячеек массива происходит автоматически, при этом **первая** ячейка получает порядковый номер, равный **0**! На это следует обращать внимание, чтобы как можно скорее перестать путаться:

- **первая** ячейка имеет порядковый номер, равный **0**
- **вторая** ячейка имеет порядковый номер, равный **1**
- **третья** ячейка имеет порядковый номер, равный **2**
- **четвертая** ячейка имеет порядковый номер, равный **3**
- и так далее...

Важно следить за этой, немного «неестественной» (с точки зрения наших привычек в «обычной жизни») особенностью языка Java; кстати, эта особенность имеется и во многих других современных языках.

Например, блок команд (фрагмент класса), заполняющий массив случайными однозначными положительными числами, будет выглядеть следующим образом:

```
int [] b;
b=new int[10];
for (int i=0; i<=9; i++)
   b[i]=1+(int)(10*Math.random());
```

Как видите, квадратные скобки в имени массива используются для указания на порядковый номер (индекс) ячейки, к которой надо получить доступ. При этом индекс ячейки может быть и конкретным числом, и переменной (ее значение, разумеется, тоже должно указывать на индекс ячейки).

Обратите внимание: переменная-счетчик цикла принимает значения от **0** до **9**, — что и соответствует точно **10** ячейкам одномерного массива **b**.

Разумеется, можно совместить в одной строке и объявление массива, и его инициализацию, что, кстати, обычно и делается:

```
int [] b=new int[10];
for (int i=0; i<=9; i++)
   b[i]=(int)(10*Math.random());
```

Выход за пределы массива

Одна из наиболее распространенных ошибок при работе с массивами — это попытка обратиться к ячейке, которой не существует в массиве, то есть указать индекс, меньший 0 или больший индекса последней ячейки массива.

Приведем в качестве примера класс, который заполняет случайными числами массив размером 10 ячеек, то есть массив с номерами от 0 до 9. Затем класс «пробует» вывести на экран значение ячейки с индексом (порядковым номером) 10...

```
import java.util.*;
class primer_array3
{
static Scanner reader = new Scanner(System.in);
public static void main(String[] args)
{
  int [] b;
  b=new int[10];
  for (int i=0; i<=9; i++)
           b[i]=(int)(10*Math.random());
  System.out.println(b[10]);
 }
}
```

Команда **System.out.println(b[10];** **не вызывает ошибки** на этапе компиляции, но «обрывает» исполнение класса, сообщая об ошибке:

java.lang.ArrayIndexOutOfBoundsException

Смысл сообщения — «выход за пределы массива», иными словами, зафиксирована попытка обращения к несуществующей в данном массиве ячейке.

Свойство «размер массива»

Есть ситуации, в которых необходимо создавать массив, размер которого при повторных запусках класса, должен быть разным. Более того, как раз такие приемы «гибкого» установления размера массива очень популярны; в большинстве заданий условие формулируется так, чтобы избежать указаний на конкретно-числовой размер массива. Именно поэтому приемы гибкого задания размера массива очень эффективны в языке Java.

В языке Java осуществить гибкое установление размера массива довольно просто: достаточно объявить и инициализировать массив с использованием ввода размера массива с клавиатуры непосредственно в процессе инициализации:

```
int[] ocenka = new int[reader.nextInt()]
```

При таком варианте нет никакой возможности заранее знать размер массива — и вроде бы возникает проблема с написанием, например, цикла для заполнения массива значениями. Почему проблема? Да потому, что вроде бы непонятно (поскольку неизвестно…), до какого значения должна изменяться переменная-счетчик.

В таком случае нужно использовать встроенный в язык Java инструмент: свойство массива под названием «размер». Этот инструмент возвращает целое число равное количеству ячеек в инициализируемом массиве (размер массива).

Свойство это записывается как служебное слово **length**, которое указывают после имени массива, через точку.

Приведем пример класса, заполняющего массив случайными числами и выводящего на экран значения ячеек массива. При этом размер массива устанавливается гибко, то есть, вводится с клавиатуры.

```
import java.util.*;
class primer_array4
{
static Scanner reader =  new Scanner(System.in);
public static void main(String[] args)
{
  int [] b=new int[reader.nextInt()];
  for (int i=0; i<=b.length-1; i++)
      {
           b[i]=(int)(10*Math.random());
           System.out.print(b[i]);
      }
}
}
```

Как видите, в заголовке цикла используется свойство «размер» — но обратите внимание, как написана верхняя граница изменений переменной-счетчика (она же, кстати, и индекс ячейки массива):

i<=b.length-1

Поскольку в этом выражении используется логическое правило «меньше или равно», то, для того чтобы остановиться на последней ячейке, **необходимо** уменьшать на 1 значение равное размеру массива.

Безусловно, это создает определенную путаницу — во всяком случае, до тех пор, пока правильная запись не станет для вас автоматической.

Однако можно использовать и другой вариант записи этого условия: с логическим правилом «строго меньше»:

for (int i=0; i<b.length; i++)

Разумеется, никакой разницы между этими двумя вариантами заголовка цикла нет, и каждый может выбрать тот вариант, который ему ближе.

Покажи мне весь массив...

Один из встроенных в язык Java методов позволяет вывести на экран весь массив одной командой, причем без использования цикла «от начальной ячейки до последней, одну ячейку за другой».

Выглядит этот метод следующим образом:

System.out.println(Arrays.toString(a));

Удобство этого метода в том, что нет нужды использовать цикл.

Неудобство же этого метода в том, что он всегда выводит массив на экран в одном и том же виде (пример — чуть ниже по тексту), в то время как использование цикла позволяет выводить массив на экран так, как нам это нужно или хочется.

А если массив не заполнять?

В языке Java предусмотрена возможность заполнять массив (ячейки массива) значениями «по умолчанию», то есть без того, чтобы в классе «откровенно» происходило заполнение массива какими-то значениями.

Возможность эта предусматривает:

- заполнение массива типа **int** значениями 0;
- заполнение массива типа **double** значениями 0.0;
- заполнение массива типа **boolean** значениями false;
- заполнение массива типа **String** значениями null.

Вот пример класса, который может показать вам эти данные на экране:

```
import java.util.*;
class primer_00
{
static Scanner reader =  new Scanner(System.in);
public static void main(String[] args)
{
    int [] a=new int[5];
    double [] b=new double[5];
    boolean [] c=new boolean[5];
    String [] d=new String[5];
    System.out.println(Arrays.toString(a));
    System.out.println(Arrays.toString(b));
```

```
        System.out.println(Arrays.toString(c));
        System.out.println(Arrays.toString(d));
    }
}
```

А вот как будет выглядеть на экране результат работы этого класса:

```
[0, 0, 0, 0, 0]
[0.0, 0.0, 0.0, 0.0, 0.0]
[false, false, false, false, false]
[null, null, null, null, null]
```

В заданиях и примерах мы будет называть такие массивы «пустыми», умышленно беря в кавычки это слово. Разумеется, такие массивы не являются пустыми, поскольку пустых массивов в принципе не бывает!

Такие массивы точнее и правильнее было бы называть «заполненными значениями по умолчанию».

Как видите, массивы эти вовсе не пустые, но именно потому, что при объявлении и инициализации их «сознательно» не заполнили, мы будем называть их «пустыми».

Раздел 23 (часть 1)
Массивы и методы

Методы — один из самых распространенных «инструментов» при написании классов, в которых используются массивы. Именно при работе с массивами требуется производить несколько операций над одним и тем же массивом. Разумеется, нет практически никаких запретов на то, чтобы все требуемые операции писать в главном методе класса, и при желании это можно делать. Стоит только иметь в виду, что класс при этом становится очень длинным и очень неудобным для изменений и редактирования.

А вот если написать достаточное число методов, то главный метод становится гораздо более простым и удобным для изменений и дополнений, а также гораздо более понятным, что особенно важно в задачах с длинным и сложным решением.

Пример заполнения массива с использованием метода

Продолжим работать с примером, в котором массив заполняется случайными числами, но теперь напишем класс так, чтобы и заполнение массива, и вывод значений в массиве (значений, находящихся в его ячейках) происходили с использованием отдельного метода.

```
import java.util.*;
class primer_array5
{
static Scanner reader =  new Scanner(System.in);
public static void main(String[] args)
{
  int [] b=new int[reader.nextInt()];
  fill_arr(b);
  print_arr(b);
}
  //Метод для заполнения массива
  public static void fill_arr (int[] c)
  {
   for (int i=0; i<c.length; i++)
```

```
    {
        c[i]=(int)(10*Math.random());
        System.out.print(c[i]+" ");
    }
    System.out.println("Массив с заполнен");
  }

//Метод для вывода на экран содержания каждой ячейки массива
Public static void print_arr(int[] t)
{
  for (int i=0; i<t.length; i++)
     {
        System.out.print(t[i]+" ");
     }
    System.out.println();
}
}
```

Обратите внимание:

- В главном методе есть команда **fill_arr(b)**, которая является обращением к методу, который, собственно, и заполняет массив значениями. Он написан после главного метода, но можно писать и до главного метода, никакого значения это не имеет. Обращение использует массив **b** в качестве передаваемого в метод параметра, при этом массив инициализирован, но значения в его ячейки, разумеется, еще не занесены (хотя, разумеется, он содержит значения по умолчанию).
- При начале исполнения метода **fill_arr**, инициализируется массив **c,** причем инициализируется еще одним, новым для нас способом: он фактически «копирует» инициализацию массива **b**. Разумеется, что при этом размер массива **c** точно такой же, как и массива **b**.
- Внутри метода **fill_arr** производится заполнение случайными числами массива **c**, значения его ячеек выводятся на экран в одну строку и затем на экран выводится сообщение **Массив с заполнен**
- При завершении исполнения метода **fill_arr** происходит возврат в главный метод. Очень важно: при возврате, все «содержимое» массива теперь является и содержимым массива **b**!
- Для проверки на экран выводятся (тоже с использованием отдельного метода, **print_arr**) значения из массива **t,** и, разумеется, они совпадают со значениями массива **b**.

Как же произошло, что заполнялся один массив (массив **c**), а значения его оказались также и значениями другого массива (массива **b**)?

Существует еще один вариант заполнения массива — с помощью метода, возвращающего массив.

```
import java.util.*;
class primer_array6
{
static Scanner reader =  new Scanner(System.in);
public static void main(String[] args)
{
  int [] b=new int[reader.nextInt()];
  b=fillArr(b.length);
  for (int i=0; i<b.length; i++)
           System.out.print(b[i]+" ");
}
  //Метод, создающий массив и возвращающий его
  public static int[] fillArr (int x)
  {
   int [] c=new int[x];
   for (int i=0; i<c.length; i++)
   {
           c[i]=(int)(10*Math.random());
           System.out.print(c[i]+" ");
   }
   System.out.println("Массив с заполнен");
   return c;
  }
}
```

Методы и вычисления в массиве

Очень удобно использовать методы для определения или для вычисления каких-то значений, характеризующих массив.

Рассмотрим следующее задание: *«Назовем „сбалансированным" одномерный массив, в котором количество четных и количество нечетных значений — одинаковы.*

Напишите класс, который заполняет одномерный массив случайными однозначными неотрицательными числами и проверяет, является ли массив „сбалансированным"».

Вначале сформулируем несколько общих соображений:

- Диапазон случайных чисел — от 0 до +9.
- Используем метод **fillArr (int x)**, который будет возвращать массив, для заполнения массива.
- Используем метод **countArr(int k, int[] c)** для подсчета и четных, и, его же, для подсчета нечетных значений. Сделаем мы это за счет того, что в первом случае вызовем метод с параметром 0, а во втором — с параметром 1; параметр этот будет определять остаток от деления на 2.
- Используем метод **is_balance (int a, int b)** для проверки того, является ли массив «сбалансированным»
- Используем метод **printArr (int[] c)** для вывода массива на экран
- За счет использования методов сведем к минимуму количество строк кода (команд) в главном методе класса.

Приведем решение — с комментариями к каждой части класса

<table>
<tr>
<td>
<pre>
import java.util.*;
class primer_array7
{
static Scanner reader = new Scanner(System.in);
public static void main(String[] args)
{
  int count1=0, count2=0;
  int [] b=new int[reader.nextInt()];
  b=fillArr(b.length);
  printArr(b);
  count1=countArr(0,b);
  count2=countArr(1,b);
  if (is_balance(count1,count2)==true)
          System.out.println("Yes");
  else
          System.out.println("No");
}
</pre>
</td>
<td>Стандартное начало класса и главный метод

В главном методе используются переменные count1 и count2 — для подсчета количества четных и нечетных значений, а также обращения к соответствующим методам.

Обратите внимание, что к методу countArr есть два обращения с разными параметрами на первом месте</td>
</tr>
<tr>
<td>
<pre>
//==========
public static boolean is_balance(int a, int b)
{
  if (a==b) return true;
  else return false;
}
</pre>
</td>
<td>Метод возвращает значение true, если переданные в метод параметры равны; а если они не равны, то метод возвращает значение false</td>
</tr>
</table>

Окончание таблицы

<table>
<tr><td><pre>//==========
public static int countArr(int x, int[] c)
{
 int count=0;
 for (int i=0; i<c.length; i++)
    if (c[i]%2==x) count++;
 return count;
}</pre></td><td>Метод возвращает количество четных или нечетных значений в массиве, в зависимости от того, равен ли параметр x или 0, или 1</td></tr>
<tr><td><pre>//==========
public static void printArr (int[] c)
{
    for (int i=0; i<c.length; i++)
          System.out.print(c[i]+" ");
    System.out.println(" End of array");

}</pre></td><td>Метод выводит на экран в одной строке все значения массива (с пробелами) и сообщение Enf of array в конце этой строки</td></tr>
<tr><td><pre>//==========
  public static int[] fillArr (int x)
  {
   int [] c=new int[x];
   for (int i=0; i<c.length; i++)
         c[i]=(int)(10*Math.random());
   printArr(c);
   return c;
  }</pre></td><td>Метод заполняет массив случайными числами из диапазона от 0 до 9 и возвращает его в главный метод</td></tr>
<tr><td>}</td><td>Конец класса</td></tr>
</table>

Обратите внимание, что к методу можно обращаться (метод можно вызывать) не только из главного метода, но и из любого другого метода — именно так, как это сделано в методе **fillArr**, который вызывает метод **printArr**.

Используя этот подход, можно изменить приведенный выше класс так, чтобы главный метод стал еще короче:

```
import java.util.*;
class primer_array8
```

```
{
static Scanner reader =  new Scanner(System.in);
public static void main(String[] args)
{
  int [] b=new int[reader.nextInt()];
  b=fillArr(b.length);
  printArr(b);
  if (is_balance(b)==true) System.out.println("Yes");
  else  System.out.println("No");
}


//==========
public static boolean is_balance(int[] c)
{
  if (countArr(0,c)==countArr(1,c)) return true; // if
  (countArr(0,c)==c.length/2)
  else return false;
}


//==========
public static int countArr(int x, int[] c)
{
 int count=0;
 for (int i=0; i<c.length; i++)
    if (c[i]%2==x) count++;
 return count;
}


//==========
public static void printArr (int[] c)
{
    for (int i=0; i<c.length; i++)
            System.out.print(c[i]+" ");
    System.out.println(" End of array");
}
```

```
//=========
  public static int[] fillArr (int x)
  {
   int [] c=new int[x];
   for (int i=0; i<c.length; i++)
         c[i]=(int)(10*Math.random());
   printArr(c);
   return c;
  }
  //=====================
}
```

Раздел 23 (часть 2)

Массивы и методы

Когда мы начинаем писать классы с использованием даже одного, а, тем более, с использованием нескольких методов, становится очень важно каждый из них сопровождать пояснениями. Пояснения эти называются «состояние входа» и «состояние выхода».

Состояние входа и выхода

Использование методов позволяет сделать проще и понятнее главный метод, а также сократить размер класса, если какие-то действия выполняются в классе несколько раз. Однако увеличение числа методов требует помнить, что выполняет каждый метод, при каких параметрах он начинает исполняться, какие данные он возвращает при своем завершении, каково состояние различных переменных при завершении исполнения метода.

Разумеется, при желании всю эту информацию можно держать в голове, но в таком случае она остается скрытой от того, кому приходится разбираться с кодом класса: например, учиться с помощью написанных кем-то классов или просто проверять сделанную кем-то работу.

Кроме того, многие «серьезные» задания можно написать с использованием разных методов, и в смысле количества, и в смысле того, что исполняет каждый метод. Иллюстрацией этого утверждения могут служить примеры классов, решение которых мы разобрали в предыдущем разделе. В этом случае пояснения, связанные описанием того, в каком состоянии метод начинает исполняться, и того, в каком состоянии метод заканчивается, становятся особенно важными.

Подытожим: при использовании метода очень важно писать для него комментарии, поясняющие, как начинается и как завершается исполнение метода.

Состояние входа / входные параметры

Состояние входа — это комментарий, описывающий, какие параметры получает метод при обращении к нему, если, разумеется, такие параметры

имеются. В случае если параметров нет, можно вообще не указывать «состояние входа».

В любом случае, формулировка должна быть краткой и укладываться в одно, максимум в два, предложения.

Состояние выхода / результат

Состояние входа — это комментарий, описывающий, что возвращает метод при завершении исполнения или какова общая цель метода.

В качестве примера того, как пишутся «состояние входа» и «состояние выхода», используем оба варианта класса из предыдущей темы.

Напомним задание: *«Назовем „сбалансированным“ одномерный массив, в котором количество четных и количество нечетных значений — одинаковы.*

Напишите класс, который заполняет одномерный массив случайными однозначными неотрицательными числами и проверяет, является ли массив „сбалансированным“».

Первый вариант решения, но на этот раз с указаниями состояния входа и состояния выхода:

```
import java.util.*;
class primer_array7
{
static Scanner reader =  new Scanner(System.in);
public static void main(String[] args)
{
  int count1=0, count2=0;
  int [] b=new int[reader.nextInt()];
  b=fillArr(b.length);
  printArr(b);
  count1=countArr(0,b);
  count2=countArr(1,b);
  if (is_balance(count1,count2)==true)
            System.out.println("Yes");
  else
            System.out.println("No");
}
public static boolean is_balance(int a, int b)
/**
```

```
Состояние входа: метод получает два целочисленных значения
Состояние выхода: метод возвращает значение true, если полученные
параметры равны; в противном случае метод возвращает значение false:
*/
{
  if (a==b) return true;
  else return false;
}

public static int countArr(int x, int[] c)
/**
Состояние входа: метод получает  целочисленное значение и одномерный
массив.
Состояние выхода: метод возвращает либо количество четных значений
в массиве (если целочисленный параметр равен 0), либо количество не-
четных значений в массиве (если целочисленный параметр равен 1).
*/
{
 int count=0;
 for (int i=0; i<c.length; i++)
    if (c[i]%2==x) count++;
 return count;
}

public static void printArr (int[] c)
 /**
Состояние входа: метод получает одномерный массив.
Состояние выхода: метод выводит на экран значения массива -- в одной
строке с пробелами и текстом End of array в конце строки.
*/
{
    for (int i=0; i<c.length; i++)
          System.out.print(c[i]+" ");
    System.out.println(" End of array");


}

public static int[] fillArr (int x)
/**
Состояние входа: метод получает  целочисленное значение, равное раз-
меру массива.
Состояние выхода: метод возвращает массив, заполненный случайными
числами из диапазона [0;9].
*/
```

```
{
   int [] c=new int[x];
   for (int i=0; i<c.length; i++)
         c[i]=(int)(10*Math.random());
   printArr(c);
   return c;
}

}
```

Второй вариант решения — с аналогичными дополнениями:

```
import java.util.*;
class primer_array8
{
static Scanner reader =  new Scanner(System.in);
public static void main(String[] args)
{
  int [] b=new int[reader.nextInt()];
  b=fillArr(b.length);
  printArr(b);
  if (is_balance(b)==true) System.out.println("Yes");
  else  System.out.println("No");
}

public static boolean is_balance(int[] c)
/**
Состояние входа: метод получает массив целых чисел.
Состояние выхода: метод возвращает значение true, если записанное в нем
условие выполняется; в противном случае метод возвращает значение false
*/
{
  if (countArr(0,c)==countArr(1,c)) return true;
  else return false;
}

public static int countArr(int x, int[] c)
/**
Состояние входа: метод получает целочисленное значение и одномерный
массив.
Состояние выхода: метод возвращает либо количество четных значений
в массиве (если целочисленный параметр равен 0), либо количество не-
четных значений в массиве (если целочисленный параметр равен 1).
```

```
*/
{
 int count=0;
 for (int i=0; i<c.length; i++)
    if (c[i]%2==x) count++;
 return count;
}

public static void printArr (int[] c)
/**
Состояние входа: метод получает одномерный массив.
Состояние выхода: метод выводит на экран значения массива — в одной строке с пробелами и текстом End of array в конце строки.
*/
{
    for (int i=0; i<c.length; i++)
          System.out.print(c[i]+" ");
    System.out.println(" End of array");

}

public static int[] fillArr (int x)
/**
Состояние входа: метод получает целочисленное значение, равное размеру массива.
Состояние выхода: метод возвращает массив, заполненный случайными числами из диапазона [0;9].
*/
{
   int [] c=new int[x];
   for (int i=0; i<c.length; i++)
         c[i]=(int)(10*Math.random());
   printArr(c);
   return c;
}
}
```

Раздел 24 (часть 1)

Массивы счетчиков

В одной из предыдущих тем мы рассматривали использование счетчиков, то есть переменных, которые позволяют подсчитать количество каких-то значений, случаев, наступления или не-наступления каких-то определенных ситуаций.

Например:

- Количество четных значений в серии чисел.
- Число неудовлетворительных оценок, полученных за контрольную работу.
- Количество максимальных значений в массиве целых чисел.

Однако использовать переменную-счетчик удобно только в том случае, если надо подсчитать один или два варианта значений: четные или нечетные значения, удовлетворительная или не удовлетворительная оценка, максимальное или не максимальное значение.

А вот если надо определить и подсчитать гораздо больше вариантов, то ситуация заметно осложняется: например, если надо определить количество каждой цифры в серии целочисленных значений. В этом случае нам понадобятся 10 счетчиков: по одному на каждую цифру. И в этой ситуации естественный выход заключается в том, чтобы использовать массив, каждая ячейка которого станет выполнять роль переменной-счетчика.

Такие массивы называются массивами счетчиков.

Рассмотрим пример: *«Напишите класс, который генерирует серию из 100 случайных чисел из диапазона [0;9] и определяет, сколько раз каждое число встречалось в этой серии».*

Для решения воспользуемся массивом счетчиков размером 10, так что в каждой ячейке в результате должно быть записано значение, соответствующее ответу на вопрос: «Сколько раз в серии было сгенерировано число, равное индексу ячейки?».

Иными словами, если по окончании генерации серии в ячейке с индексом, например, 3, будет записано значение 17, это будет означать, что число 3 было сгенерировано 17 раз.

Для этого каждое сгенерированное случайное значение будет указывать на индекс той ячейки, в которой надо будет увеличить значение на 1, то есть каждая ячейка массива будет использоваться точно в соответствии с правилами использования переменных-счетчиков.

Разумеется, нам надо будет в самом начале обнулить весь массив, и это опять же в соответствии с правилом «перед началом подсчета поставь счетчик в значение 0».

Таким образом, в нашем классе мы может использовать методы для следующих действий:

- Обнуления массива счетчиков.
- Генерацию серии с одновременным изменением значений в соответствующих ячейках массива счетчиков.
- Вывод состояния массива счетчиков на экран.

Решение может выглядеть так:

```
import java.util.*;
class primer_array9
{
static Scanner reader =  new Scanner(System.in);
public static void main(String[] args)
{
  int [] count=new int[10];
  zeroArr(count);
  fillArr(count);
  printArr(count);
}

public static void printArr (int[] c)
/**
Состояние входа: метод получает массив.
Состояние выхода: метод выводит массив на экран, индекс и значение каждой ячейки в отдельной строке.
*/
  {
    for (int i=0; i<c.length; i++)
          System.out.println(i+"="+c[i]);
  }

  public static void fillArr (int[] c)
/**
```

```
Состояние входа: метод получает массив, содержащий во всех ячейках
значение 0.
Состояние выхода: метод изменяет массив, который теперь будет за-
полнен количеством появлений значения, равного индексу каждой ячейки,
в серии из 100 случайных чисел из диапазона [0;9].
*/
  {
   int x;
   for (int i=1; i<=100; i++)
     {
        x=(int)(10*Math.random());
        c[x]++;
      }
  }

  public static void zeroArr (int[] c)
/**
Состояние входа: метод получает массив.
Состояние выхода: метод заменяет все значения в массиве на значения 0.
*/
  {
    for (int i=0; i<c.length; i++)
          c[i]=0;
   }
}
```

Предлагаем вам самостоятельно разобраться с еще одним вариантом класса для этого же примера.

```
import java.util.*;
class primer_array9
{
static Scanner reader =  new Scanner(System.in);
public static void main(String[] args)
{
  int [] count;
  count=zeroArr(10); // создание и обнуление массива счетчиков
  fillArr(count, 100); // заполнение массива счетчиков
  printArr(count); // печать массива счетчиков
}

public static void printArr (int[] c)
/**
```

```
Состояние входа: метод получает массив.
Состояние выхода: метод выводит массив на экран, индекс и значение
каждой ячейки в отдельной строке.
*/
  {
   for (int i=0; i<c.length; i++)
        System.out.println(i+"="+c[i]);
  }

  public static void  fillArr (int[] c, int c)
/**
Состояние входа: метод получает массив, содержащий во всех ячейках
значение 0
Состояние выхода: метод заполняет массив количеством появлений зна-
чения, равного индексу каждой ячейки, в серии из n случайных чисел из
диапазона [0;9].
*/
  {
   int x;
   for (int i=1; i<=n; i++)
     {
        x=(int)(10*Math.random());
        c[x]++;
     }
  }

  public static void  zeroArr (int[] c)
/**
Состояние входа: метод получает массив.
Состояние выхода: метод заполняет массив нулями.
*/
  {
   for (int i=0; i<c.length; i++)
        c[i]=0;
  }
}
```

Разумеется, массив счетчиков можно использовать и в тех случаях, когда индексы ячеек не имеют прямого или вовсе никакого отношения к сущности значений, которые следует подсчитывать.

Для этого просто следует решить, какое значение будет считать та или иная ячейка массива счетчиков.

Например, при социологическом исследовании населения в анкетах просили указать «категорию граждан», — чтобы определить, например, потребности детских садов, школ, других элементов социальной инфраструктуры. Категории были такими: дошкольник, школьник, студент, работающий, безработный, пенсионер, инвалид.

В этом случае можно создать массив счетчиков размером 7 (по числу категорий) и решить, что ячейка с индексом 0 будет счетчиком дошкольников, ячейка с индексом 1 — школьников, ячейка с индексом 2 — студентов, и так далее.

Раздел 24 (часть 2)

Массивы сумматоров

Аналогично массивам счетчиков, широко применяются и массивы сумматоров.

Использование переменных-сумматоров связано, как правило, с необходимостью подсчитывать общее количество каких-то значений:

- Суммы всех четных и суммы всех нечетных чисел в серии значений.
- Общего количества мальчиков и общего количества девочек в школе, — при введении соответствующих данных по каждому классу.
- Общего числа проданных билетов на всех аттракционах парка, — при введении данных по каждому аттракциону.
- Общую стоимость закупки разными покупателями, причем каждый из покупателей покупает несколько разных по стоимости товаров.

В случае переменных-сумматоров, так же, как и в ситуации с переменными-счетчиками, ими удобно пользоваться, если речь идет о подсчете одной, двух, максимум трех разных сумм. Как и в ситуации со счетчиками, при необходимости подсчитывать достаточно большое количество разных сумм, следует использовать массив, каждая ячейка которого исполняет функцию переменной-сумматора.

Раздел 25

Массив массивов

Несомненно, что использование таких элементов, как массив, позволяет очень эффективным способом хранить и обрабатывать наборы значений. Существует огромный круг задач, решение которых без использования массивов становится настолько громоздким, что не имеет почти никакого смысла приступать к написанию классов.

Но это совершенно не означает, что массив (а точнее, одномерный массив) является инструментом, который достаточно легко позволяет решить любую задачу, связанную с хранением и обработкой групп данных. Существует достаточно много задач, для решения которых приходится прибегать к использованию нескольких массивов, — а это, в свою очередь, возвращает нас к ситуации, с которой мы столкнулись, когда вместо многих переменных одного и того же типа стали использовать один массив.

Приведем несколько примеров, в которых становится необходимым использовать более одного массива.

Скажем, нам надо хранить информацию о недельном расписании уроков какого-то класса или ученика. В этом случае нам потребуется использовать 5 или 6 массивов — по одному на каждый день занятий. Это могут быть массивы типа «строковое значение» (если мы решим хранить в них названия предметов), а могут быть и массивы с целочисленными значениями (если мы пронумеруем все школьные предметы).

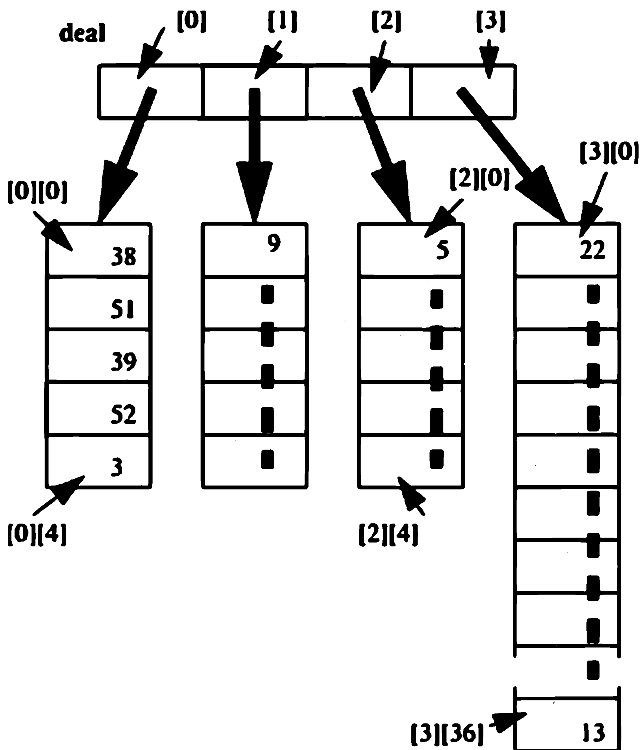
Другой пример — транспортная компания, обслуживающая несколько городских маршрутов: в этом случае, если нам требуется хранить информацию о каждом из маршрутов (названия остановок, время движения по маршруту, количество пассажиров, входящих и выходящих на каждой остановке, и так далее) — придется для каждого маршрута использовать свой массив.

Можно привести еще множество аналогичных задач, при решении которых потребуется использовать несколько (точнее, много...) однотипных массивов. Это и задачи, связанные с расписанием, и задачи, связанные с отчетами (о рабочем времени на предприятии, в конторе, в учебном заведении — по каждому «участнику», отделу, классу, цеху...), с финансовой информацией (движение денег на счетах клиентов)... и прочая, и прочая, и прочая...

В свое время, когда мы оказались перед проблемой «слишком много переменных» — ее решение оказалось в использовании массива: «одно имя, много значений в пронумерованных ячейках».

Теперь, когда мы оказались перед проблемой «слишком много массивов», ее решение мы найдем в использовании аналогичного подхода: создании массива массивов.

Массив массивов — это структура, которая представляет из себя «обычный» массив, каждая из ячеек которого — тоже «обычный» массив. Вот так и получится у нас — массив массивов.



А для того чтобы меньше путаться с тем, о каком из массивов идет речь, будем использовать термины «внешний массив» и «внутренние массивы».

Во многих языках в этих случаях принято говорить о «двумерных массивах», — отделяя и отличая их таким образом от массивов «одномерных». Кроме того, часто используется и термин «табличный массив», поскольку двумерных массив состоит как бы из столбцов и строк.

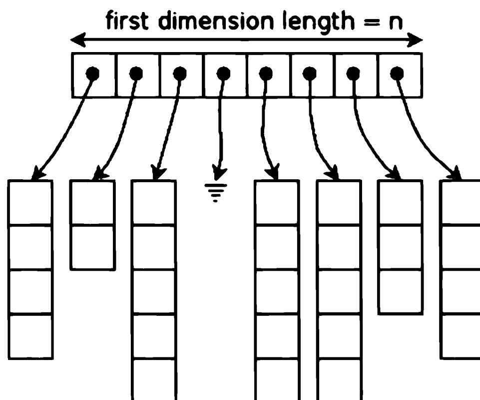
Следует сказать, что термины «одномерный массив» и «двумерный массив» довольно часто используются и в применении к языку Java, но, строго говоря, это очень и очень спорно с точки зрения смысла.

Дело в том, что в таких языках, как, например, BASIC или Pascal, размер массива надо было обязательно указывать с самого начала в конкретных числовых значениях — и не было совершенно никакой(!), просто категорически никакой возможности устанавливать размер массива гибко, то есть в процессе работы программы, а не в процессе написания программы.

В языке Java — ситуация принципиально иная. Мы еще при изучении темы «Массивы» познакомились с этой замечательной возможностью языка Java — устанавливать размер массива гибко, при исполнении класса. Эта возможность избавляет нас от указания в фиксированном числовом виде еще в кодах указывать размер массива (хотя и это можно сделать, — можно, но совершенно не обязательно).

Именно благодаря этой возможности «двумерный» массив в языке Java — совсем не то, что в «старых» языках: в каждой ячейке массива можно «разместить» внутренний массив любого размера, причем в разных ячейках — разного размера.

Мы не случайно взяли в кавычки слово «разместить» в предыдущем абзаце — поскольку в ячейках «внешнего» массива на самом деле располагаются не «внутренние» массивы, а ссылки на эти внутренние массивы. И именно потому, что в ячейках «внешнего» массива располагаются не сами внутренние массивы, а ссылки на них, — «внутренние» массивы могут быть разных размеров.



Мы в дальнейшем будем стараться использовать именно термин «массив массивов», — но следует иметь в виду, что (очевидно, в силу привычки) довольно часто используется выражение «двумерный массив».

Объявление и инициализация массива массивов

Объявление массива массивов чрезвычайно похоже на объявление «обычного», «одномерного» массива:

int [] a; //это — одномерный массив
int [] [] b; //это — массив массивов, «двумерный» массив

Тот факт, что «двумерный» массив по сути дела является именно массивом массивов, позволяет инициализировать в каждой ячейке «внешнего» массива «внутренний» массив любого размера.

Например, класс:

```
import java.util.*;
class primer_arr2_1
{
static Scanner reader = new Scanner(System.in);
public static void main(String[] args)
{
   int [] []a=new int[5][];
   for (int i=0;i<a.length;i++)
      a[i]=new int[i];
   for (int i=0;i<a.length;i++)
      System.out.println(Arrays.toString(a[i]));
}
}
```

выполняет следующие действия:

int [] []a=new int[5][];	Объявляет и инициализирует массив массивов размером в 5 ячеек; при этом уже на этапе объявления указано, что это будет массив массивов. Однако при этом внутренние массивы объявлены (так как дважды использованы квадратные скобки) — но еще не инициализированы. Именно поэтому внешний массив имеет размер (5 ячеек), а внутренние массивы не имеют пока никаких размеров (повторим еще раз — они объявлены, но не инициализированы)

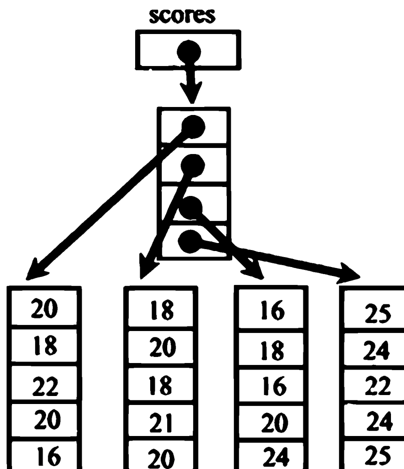
Окончание таблицы

for (int i=0;i<a.length;i++) a[i]=new int[i];	Инициализируются внутренние массивы, при этом размер каждого массива устанавливается равным номеру ячейки, с которой он связан. Между прочим, это позволяет создать в ячейке номер 0 ссылку на внутренний массив размером 0...
for (int i=0;i<a.length;i++) System.out.println (Arrays.to String(a[i]));	Значения, содержащиеся во внутренних массивах, выводятся на экран. Поскольку «сознательно» значения в инициализированные внутренние массивы не заносились, они — «пустые», то есть содержат значения 0 (потому что речь идет о массиве типа **int**)

Результат вывода информации на экран при исполнении этого класса:

```
[]
[0]
[0, 0]
[0, 0, 0]
[0, 0, 0, 0]
```

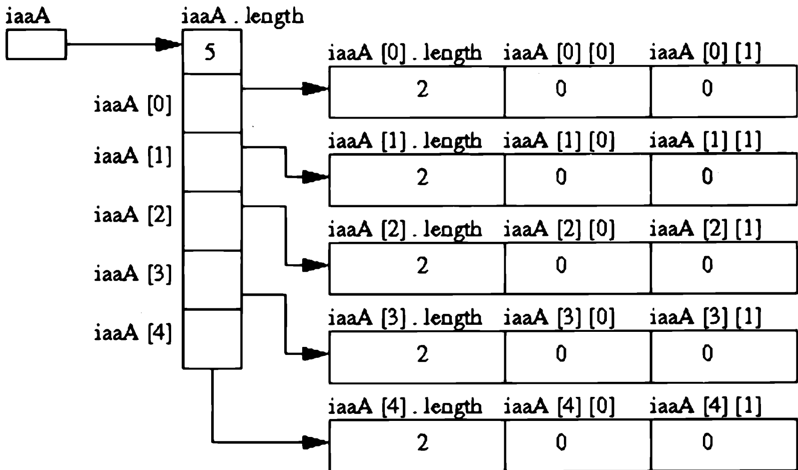
Как обращаться к ячейкам?



В принципе, ничего специфического в смысле обращения к ячейкам массива массивов нет: надо указывать два номера ячеек — первый относится к ячейке внешнего массива, а второй — к ячейке внутреннего, массива, связанного с указанной ячейкой внешнего.

Например, запись **a[3] [2]** означает, что речь идет о ячейке номер 2 внутреннего массива, который связан с ячейкой номер 3 внешнего массива.

Во многих случаях, по традиции, используются термины «строка» и «столбец», которые сохранились со времен языков вроде BASIC и Pascal. В этих языках массивы и в самом деле были «двумерными», то есть «прямоугольными таблицами», в которых легко было отыскать и строки, и столбцы.



Применимо же к языку Java термин «строка означает соответствующую ячейку внешнего массива, а термин «столбец» — соответствующую ячейку внутреннего массива, связанного с ячейкой внешнего. А поскольку до сих пор в достаточно большом количестве задач рассматривается ситуация, при которой размеры всех внутренних массивов одинаковы, то использование терминов «строка» и «столбец» продолжает оставаться распространенным.

Квадратный массив массивов

Очень часто встречаются задачи, в которых речь идет о «квадратном» массиве. В этих случаях речь идет о массиве массивов, в котором размер внешнего массива равен размеру каждого из внутренних массивов.

Это позволяет говорить о «диагоналях», то есть о группах ячеек, у которых оба индекса — одинаковы.
Например, в классе:

```
import java.util.*;
class primer_arr2_2
{
static Scanner reader =  new Scanner(System.in);
static final int n=6; //Объяснение этой записи, встречающейся впервые —
ниже!
public static void main(String[] args)
{
   int [] []a=new int[n][n];
   for (int i=0;i<a.length;i++)
   {
    a[i][i]=1;
    a[i][n-i-1]=2;
   }
   for(int i=0;i<a.length;i++)
      System.out.println(Arrays.toString(a[i]));
}
}
```

Одна «диагональ» заполняется значениями 1, а вторая — значениями 2. На экран это будет выводиться вот в таком виде:

```
[1, 0, 0, 0, 0, 2]
[0, 1, 0, 0, 2, 0]
[0, 0, 1, 2, 0, 0]
[0, 0, 2, 1, 0, 0]
[0, 2, 0, 0, 1, 0]
[2, 0, 0, 0, 0, 1]
```

Обратите внимание еще на одну строку в классе:

static final int n=6;

Она не просто устанавливает значение переменной **n** равным 6, но еще и описывает эту переменную как «недопустимую к изменениям значения»

(**final**). Это ограничение связано с уже упоминавшимся правилом: размер массива можно устанавливать гибко, но после установления нельзя менять.

Мы приведем еще несколько примеров для методов, использующих в качестве параметра (или, по крайней мере, одного из параметров) «прямоугольный» двумерный массив (массив, в котором во всех строках одинаковое количество ячеек). Дело в том, что «прямоугольные» двумерные массивы фактически являются стандартными таблицами данных — а работа с такими таблицами является (и еще долго будет оставаться) одной из главных потребностей современных информационных систем.

«Напишите метод, который принимает в качестве параметра „прямоугольный" двумерный массив и возвращает сумму значений в этом массиве».

Разумеется, мы будем исходить из предположения, что массив уже содержит какие-то значения; кроме того, для вычисления суммы значений в массиве следует знать, какого типа эти значения. Однако, поскольку с точки зрения того, как метод будет написан, нет никакой разницы между числовыми типами — будем писать метод для массива типа int.

```
public static int sum_arr(int [][] t)
{
 int s=0;
 for(int i=0;i<t.length;i++)
   for(int j=0;j<t[i].length;j++)
     s+=t[i][j];
 return s;
}
```

Обратите внимание, как с помощью t.length определяется количество «строк», а с помощью t[i].length — количество ячеек в «строке». Напомним: двумерный массив, по сути, является массивом массивов.

«Напишите метод, который принимает в качестве параметра „прямоугольный" двумерный массив и выводит на печать суммы значений в каждой из строк массива».

Написать этот метод можно двумя способами: либо создать метод типа void, который будет выводить на экран нужные значения, — либо использовать вспомогательный метод для вычисления сумм, а нужный нам метод будет только выводить на экран требуемые значения.
Приведем решение по первому способу.

```
public static void print_sum(int [][] t)
```

```
{
 int s;
 for(int i=0;i<t.length;i++)
 {
  s=0;
  for(int j=0;j<t[i].length;j++)
    s+=t[i][j];
  System.out.print(s+" ");
 }
}
```

«Напишите метод, который принимает в качестве параметра „прямоугольный" двумерный массив и выводит на печать индексы всех ячеек, которые содержат максимум массива».

В этом примере тоже есть два варианта решения: можно в методе использовать обращение к дополнительному методу, который возвращает максимальное значение, содержащееся в массиве, а можно передавать в метод два параметра — и сам массив, и уже найденное предварительно максимальное значение.

```
public static void max_pos(int [][] t, int max)
{
 for(int i=0;i<t.length;i++)
  for(int j=0;j<t[i].length;j++)
    if (t[i][j]==max)
          System.out.print(i+" & "+j+" : ");
 }
}
```

*«Напишите метод, который принимает в качестве параметра „прямоугольный" двумерный массив и проверяет, являются ли все его нечетные строки упорядоченными по возрастанию. Метод должен возвращать одно из значений, **true** или **false**, в зависимости от результатов проверки».*

Поскольку в решении использовано несколько заслуживающих внимания приемов, мы опишем их с помощью комментариев к соответствующим строкам метода.

```
public static boolean is_up(int [][] t)
{
 boolean res=true;
//Устанавливаем исходное значение возвращаемой переменной
 for(int i=1;i<t.length;i+=2)
```

```
  {
   int [] b=new int[t[1].length];
// Создаем вспомогательный массив размером, равным размеру строки
   for (int j=0;j<t[i].length;j++)
              b[j]=t[i][j];
// Копируем строку во вспомогательный массив
   Arrays.sort(b);
// Сортируем вспомогательный массив
      if(t[i].equals(b)==false)
          res=false;
/*
Если строка и вспомогательный массив неодинаковы, значит строка не упорядочена по возрастанию, и возвращаемая переменная получает соответствующее значение
*/
  }
 return res;
}
```

«Напишите метод, который принимает в качестве параметров „прямоугольный“ двумерный массив и две пары чисел, каждая из которых указывает на одну из ячеек. Метод должен возвратить среднее арифметическое значений из всех ячеек, расположенных „построчно“ между указанными».

Поясним пример рисунком, на котором закрашены фоном те ячейки, среднее арифметическое которых следует найти. Для приведенного ниже рисунка первая пара чисел должна быть 1 и 3 (строка № 1 и в ней ячейка № 3), а вторая пара чисел — 3 и 1.

```
public static double sum_part_arr(int [][] t, int row1, int col1, int row2, int col2)
{
 double s=0;
```

```
  int count=0;

  return s;
}
```

Одним из частных случаев «прямоугольного» двумерного массива является массив «квадратный», в котором количество строк равно количеству столбцов. Особенностью такого типа двумерных массивов является существование в нем «диагоналей»: «главная» — из левой верхней ячейки в правую нижнюю, «вторичная» — из правой верхней ячейки в левую нижнюю. Эта особенность «квадратных» двумерных массивов порождает дополнительную группу задач. Разберем некоторые из них.

«Напишите метод, который принимает в качестве параметра двумерный „квадратный" массив и возвращает сумму элементов, находящихся на главной диагонали».

В методе использован факт, что индексы ячеек, находящихся на главной диагонали, являются парами одинаковых чисел.

```
public static int sum_diag(int [][] t)
{
  int s=0,n=t[0].length;
  for(int i=0;i<n;i++)
      s+=t[i][i];
  return s;
}
```

«Напишите метод, который принимает в качестве параметров двумерный „квадратный" массив и целое число X. Метод возвращает количество ячеек, которые содержат значение X и находятся на одной из диагоналей».

В этом методе используются два цикла: один подсчитывает количество подходящих ячеек на главной диагонали, а второй — количество таких же ячеек на вторичной диагонали.

```
public static int count_diags(int [][] t, int x)
{
  int count=0,n=t[0].length;
  for(int i=0;i<n;i++)
      if(t[i][i]==x) count++;
  for(int i=0;i<n;i++)
      if(t[i][n-i-1]==x) count++;
return count;
}
```

*«Напишите метод, который принимает в качестве параметра двумерный „квадратный“ массив и проверяет, содержат ли все ячейки над главной диагональю одно и то же значение. Метод должен возвращать одно из значений, **true** или **false**, в зависимости от результатов проверки».*

В приводимом решении следует обратить внимание на то, что внутренний цикл (подсчитывает сумму значений в нужных ячейках соответствующей строки) зависит от цикла внешнего. Кроме того, внешний цикл заканчивается на предпоследней строке, потому что в последней строке нет ни одной ячейки над диагональю.

```
public static boolean allow(int [][] t)
{
  boolean res=true;
  int x=t[0][1], n=t[0].length;
  for(int i=0;i<t.length-1;i++)
    for(int j=i+1;j<n;j++)
       if(t[i][j]!=x) return false;
  return res;
}
```

*«Напишите метод, который принимает в качестве параметров двумерный „квадратный“ массив M и три целых числа — row, col, long. Метод должен проверить, существует ли в массиве M квадратный подмассив, верхняя левая ячейка которого расположена в позиции [row, col] и размер которого long X long. Метод должен возвращать одно из значений, **true** или **false**, в зависимости от результатов проверки».*

```
public static boolean is_found(int [][] m, int row, int col, int long)
{
  boolean res=true;
  n=t[0].length;
  if (row+long>n) return false;
  if (col+long>n) return false;
  return res;
}
```

Объекты в языке Java

Объекты «вообще»

Необходимым требованием для овладения программированием для системы Android является понимание понятия «объект» и владение основными навыками написания в языке Java классов, использующих объекты. На профессиональном языке это называется объектно-ориентированным программированием, — и перед тем, как мы приступим непосредственно к овладению навыками работы для Android, следует накопить хотя бы минимальный опыт работы с объектами в языке Java.

Для того, кто овладел базовыми навыками написания классов в языке Java, термины «объект» и «ориентированное на использование объектов программирование» выглядят в первый момент чем-то если не пугающим, то наверняка искусственно придуманным компьютерщиками.

В действительности же ситуация совершенно противоположна: нет ничего более естественного для окружающего нас мира, чем объекты и, соответственно, действия, связанные с использованием объектов.

Показательно, что очень понятное определение мы нашли как раз не у программистов, а в работе психоаналитика Ханы Сегал: *«Что такое объект? В привычном смысле слова объектом является все, что имеет в нашем мире материальные качества».*

После этого, на наш взгляд, гораздо понятнее выглядит и определение, используемое в специфической, компьютерной сфере: *«Объект — некоторая сущность в виртуальном пространстве, обладающая определенным состоянием и поведением, имеет заданные значения свойств (атрибутов) и операций над ними (методов)».*

Например, объектом является «ученик»:

- у него имеются определенные свойства (рост, вес, имя-фамилия, номер класса, набор оценок и так далее);
- над ним можно выполнять различные операции (методы) (переводить из класса в класс, оставлять на второй год, выставлять ему оценки изменять имя-фамилию и прочие).

Поскольку все в языке Java создается классами, то конкретного ученика (Вася Иванов, ученик 10А класса, 1.80 ростом, 70 кг весом, имеющий 5 по

математике...) в профессиональных терминах называют «экземпляром класса».

Уверены, что аналогичные примеры без труда в достаточном количестве может привести каждый.

Объект какого типа?

Объект может быть любого типа — например, типа «целое число».

Например, строка **int x** объявляет «просто» переменную типа «целое число».

А вот строка **Integer y=new Integer(5)** создает переменную, указывающую на объект типа «целое число».

Законно будет задать вопрос: а какая разница будет между этими переменными? Не вдаваясь в подробности (мы еще ведь ничего не изучили про объекты, какие уж тут подробности...), можно сказать так: разница в возможностях использования этих переменных, — у переменной, которая указывает на объект, их значительно больше.

Кроме того, объекты, как правило, используются в тех случаях, когда невозможно или крайне неудобно использовать простые (в частности — уже известные нам) типы, вроде «целое число».

Что значит — «указывает на объект»?

Надеемся, что вы уже обратили внимание на дважды повторенное выражение: «переменная, указывающая на объект». На профессиональном языке говорят о «переменной ссылочного типа».

Не углубляясь в излишние (пока!) подробности, сформулируем разницу между «просто» переменной и переменной, работающей с объектами:

- «Простая» переменная (например, описанная как **int x=4**) впрямую и непосредственно связана с определенным значением. Часто в таких случаях говорят: «переменная имеет значение» или «переменная хранит значение». Это значит, что описание **int a=4,b=4** не только создает две переменные, но и создает два(!) значения, каждое из которых хранится в своей переменной.
- Переменная, указывающая на объект, не хранит никакого «настоящего» значения, а только «ссылку на объект», — то есть фактически адрес той ячейки памяти, начиная с которой записаны данные, образующие объект. Поэтому на одно и то же место в памяти, содержащее данные объекта, могут указывать несколько переменных-ссылок.

Зачем нужны объекты

Использование именно объектов, а не ставших уже привычными «простых» переменных легко понять, если обратить внимание, что в качестве характеристик (данных) чаще всего приходится использовать данные разных типов.

В частности, в использованном выше примере с учеником, при внимательном рассмотрении, легко обратить внимание на то, что:

- Имя и фамилия — значения (данные) типа String.
- Оценки — значения типа int.
- Рост — значение типа double.
- Буквенное обозначение класса (А, Б и так далее) — значение типа char.

Вот для того чтобы собрать все эти значения «под одной крышей» (под одним именем), и используются объекты.

А для этого в первую очередь необходимо уметь создавать в языке Java такие типы, которые будут содержать в качестве составляющих их частей данные разных типов (как в приведенном только что примере про ученика).

Объекты в Java

Поскольку в языке Java код всегда пишется в виде класса, то и объекты создаются и обрабатываются классами, а посему наравне с термином «объект» используется термин «объект класса».

В случае если речь идет о необходимости создания объекта такого типа, который в стандартном виде не существует в языке Java, то для создания нового типа и, соответственно, объектов этого типа, следует написать отдельный, специальный класс. Именно его экземплярами и будут создаваемыми на его же основе объекты.

Например, если нам требуется обрабатывать данные учеников, и мы хотим для этого использовать объект типа **Student** (который, разумеется, в стандартных типах языка Java не присутствует!), то и описание этого типа должно быть сделано в классе с тем же именем — **public class Student**. Обратите внимание на то, что перед словом **class** (как правило!.. но возможны и другие варианты...) используется указание **public**: это делает класс «общедоступным», то есть разрешает его использование другими классами.

Свойства

Свойства объекта нужного типа (имя-фамилия, класс, рост — какие нужны) описываются в том же классе, который будет описывать тип и создавать экземпляры класса. Описываются они так же, как описываются все переменные (каковыми, по сути, свойства и являются), но при этом желательно (хотя и — не обязательно) использовать указание **private**: использование private скрывает от внешнего класса подробности внутреннего строения класса, при этом доступ к значениям свойств производится не напрямую, а через методы.

Методы

Все методы, в сущности, составляют несколько групп:

- методы-конструкторы — они создают экземпляры объекта (экземпляры класса);
- методы, устанавливающие значение свойств экземпляра объекта (set);
- методы, возвращающие значение свойств экземпляра объекта (get);
- метод, выводящий на экран сгруппированную в строковое значение информацию о «состоянии объекта» (его параметрах);
- различные дополнительные методы (по мере необходимости в зависимости от потребностей каждой отдельно взятой задачи).

Все эти методы описываются в одном классе, но перед написанием собственно кода этого класса правильно и полезно в общих чертах провести предварительную подготовку.

В общих чертах

Для того чтобы не импровизировать при написании кода класса, будет верным предварительно написать таблицу, в которой удобно заранее, и в общем виде (без кода) описать словами и свойства, и методы, которые затем будем описывать кодами языка Java.

Существуют общие требования к оформлению класса:

- Конструктор (или конструкторы) не имеют типа (они ведь «создают» новый экземпляр класса!)
- Для каждого свойства объекта есть соответствующий метод getИмя_Свойства(), например, String getName() или int getMark(). Обратите внимание, что у методов нет параметров, точнее, есть пустой список параметров.

- Для каждого свойства объекта есть соответствующий метод void set*Имя_Свойства*(*тип_свойства Новое_Значение*), например, void setMark(int newMark) или void setName(String newName).
- Метод со стандартным названием toString() возвращает строку, содержащую значения всех свойств данного объекта (экземпляра класса). Это позволяет узнать содержимое объекта, используя стандартный вывод на экран System.out.println, точно так же, как мы распечатывали значения простых переменных.

Один из наиболее удобных вариантов такой таблицы выглядит (в общих чертах) так:

<table>
<tr><th colspan="3">Имя класса</th></tr>
<tr><td>Тип и имя свойства</td><td>Описание свойства</td><td rowspan="4">Свойства</td></tr>
<tr><td>Тип и имя свойства</td><td>Описание свойства</td></tr>
<tr><td>Тип и имя свойства</td><td>Описание свойства</td></tr>
<tr><td>...</td><td>...</td></tr>
<tr><td>Имя конструктора</td><td>Описание конструктора</td><td rowspan="4">Конструкторы</td></tr>
<tr><td>Имя конструктора</td><td>Описание конструктора</td></tr>
<tr><td>Имя конструктора</td><td>Описание конструктора</td></tr>
<tr><td>...</td><td>...</td></tr>
<tr><td>Тип и имя возвращающего метода</td><td>Описание возвращающего метода</td><td rowspan="4">Возвращающие методы</td></tr>
<tr><td>Тип и имя возвращающего метода</td><td>Описание возвращающего метода</td></tr>
<tr><td>Тип и имя возвращающего метода</td><td>Описание возвращающего метода</td></tr>
<tr><td>...</td><td>...</td></tr>
<tr><td>Тип и имя устанавливающего метода</td><td>Описание устанавливающего метода</td><td rowspan="4">Устанавливающие методы</td></tr>
<tr><td>Тип и имя устанавливающего метода</td><td>Описание устанавливающего метода</td></tr>
<tr><td>Тип и имя устанавливающего метода</td><td>Описание устанавливающего метода</td></tr>
<tr><td>...</td><td>...</td></tr>
<tr><td>Тип и имя информационного метода</td><td>Описание информационного метода</td><td>Информационный метод (toString)</td></tr>
<tr><td>Тип и имена дополнительных методов</td><td>Описание дополнительных методов</td><td>Дополнительные методы</td></tr>
</table>

Приведем в качестве примера описание класса, который можно будет использовать для создания объектов типа *«точка на плоскости»:*

Точка (Point)		
Point (char name, double x, double y)	Создает (строит, конструирует) точку с именем **name** и координатами **x** и **y**	**Метод-конструктор**
double getX()	Возвращает координату **x** точки	**Методы, возвращающие значения**
double getY()	Возвращает координату **y** точки	
char getName()	Возвращает имя точки	
void setName(char name)	Устанавливает значение имени точки, в соответствии с указанным параметром, который получает метод	**Методы, устанавливающие значения**
void setX(double x)	Устанавливает значение координаты **x** точки, в соответствии с указанным параметром, который получает метод	
void setY(double y)	Устанавливает значение координаты **y** точки, в соответствии с указанным параметром, который получает метод	
String toString()	Метод возвращает строковое значение, «изображающее» информацию о точке	**Информационный метод**

А теперь — класс! Я сказал — класс!

Вот как будет выглядеть класс, описанный в общем виде в таблице:

```
public class Point // точка на плоскости
{
//Сначала описываем три свойства для нового типа
  private char name;
  private double x;
  private double y;
```

```
 public Point( char name, double a, double b)
  {
   this.name=name;
   this.x=a;
   this.y=b;
  }

 public double getX()
  {
   return this.x;
  }

 public double getY()
  {
   return this.y;
  }

 public char getName()
  {
   return this.name;
  }

public void setX(double a)
  {
  this.x=a;
  }

 public void setY(double b)
  {
  this.y=b;
  }

 public String toString()
  {
   return "This is a point: "+this.name+" ("+this.x+","+this.y+")";
  }
}
```

Это this — это что?

Для того, кто уже прошел курс «Компьютерные науки» и владеет основами написания и отладки классов на языке Java, в приведенном только что выше примере практически все должно быть знакомо и понятно.

Да, пожалуй, единственным исключением: использование служебного элемента **this** («этот»).

Точнее всего, пожалуй, будет сказать так: **this** — это указание, что данный метод следует приметить к тому объекту, относительно которого в данный момент используется этот самый метод.

А чтобы более подробно объяснить цель использования указателя **this**, попробуйте ответить на очень простой вопрос: глядя на класс **Point**, можно ли сказать, какое имя имеет объект, создаваемый или обрабатываемый методами этого класса? Может, этот объект называется **to4ka**? А может — **position**? Или — **mesto_na_ploskosti**? Или как-то еще?

И ответ получается единственным: да нет в классе **Point** вообще никакой информации о том, каким должно быть имя экземпляра объекта **Point** — и поэтому имя это может быть любым (в соответствии с требованиями к именам языка Java).

Вот для того чтобы сохранить эту свободу имени, для того чтобы класс **Point** можно было приметить к **любому** экземпляру этого класса, — и используется в самом классе указание **this**: «примени этот метод к тому объекту, о котором идет сейчас речь — и не суть важно, какое у него имя».

А чего тут нет?

Посмотрите еще раз на класс, который мы привели выше в качестве примера: какой привычной части в нем не хватает? Обратили внимание? Совершенно верно: в нем отсутствует главный метод, то самый, который всегда имеет стандартный заголовок — **public static void main(String[] args)**.

Все дело в том, что в нем — в данном классе — просто нет никакой необходимости. Ведь, собственно, класс **Point** не предназначен для выполнения каких-то действий в главном методе, а для того чтобы методы и свойства, описанные в этом классе, использовали другие классы.

А как использовать?

Разумеется, немедленно возникает вопрос: так как же использовать написанный класс?

Приведем пример такого класса, использующего класс **Point**, а затем разберем этот пример.

```
class testPoint
{

  public static int quarta(Point a)
  {
/*
Метод получает в качестве параметра объект a типа Point и проверяет, в
какой четверти находится объект. В случае если объект находится на од-
ной из осей, метод возвращает значение 0
*/
      if (a.getX()>0 && a.getY()>0) return 1;
      if (a.getX()<0 && a.getY()>0) return 2;
      if (a.getX()<0 && a.getY()<0) return 3;
      if (a.getX()>0 && a.getY()<0) return 4;
      return 0;
  }

  public static boolean isAxis(Point b)
  {
/*
Метод получает в качестве параметра объект b типа Point и проверяет,
находится ли объект на одной из осей. В случае если находится, метод
возвращает true, в противном случае метод возвращает false
*/
    if (b.getX()==0 || b.getY()==0) return true;
    return false;
  }

  public static boolean PandP(Point a, Point b)
  {
/*
Метод получает в качестве параметра объекты a и b типа Point и возвра-
щает true, если обе точки находятся в одной четверти системы координат;
в противном случае метод возвращает false
*/
    if (quarta(a)==quarta(b)) return true;
    return false;
  }
```

```
   public static double howLong(Point a, Point b)
   {
  {
/*
Метод получает в качестве параметра объекты a и b типа Point и возвра-
щает расстояние между ними
*/
         double dx= ((a.getX()-b.getX())*(a.getX()-b.getX());
         double dy= ((a.getY()-b.getY())*(a.getY()-b.getY()))
         double d= Math.sqrt(dx+dy);
         return d;
   }
  public static void main(String[]args)
  {
   Point x=new Point('A',6,8);
   Point y=new Point('B',5,8);
   System.out.println(x);
   System.out.println(quarta(x));
   System.out.println(isAxis(x));
   System.out.println(PandP(x,y));
   System.out.println(howLong(x,y));
  }
}
```

А теперь разберем команды главного метода.

Команда	Описание
Point x=new Point('A',6,8);	Осуществляется вызов конструктора Point с параметрами 'A' (имя), 6 и 8 (координатами по осям x и y соответственно), и этот вызов создает объект типа Point с соответствующими параметрами. Затем используется действие new, которое создает переменную-ссылку по имени **x** на только что созданный объект. Иными словами — создается экземпляр **x** класса Point

Окончание таблицы

Команда	Описание
Point y=new Point('B',5,8);	Осуществляется вызов конструктора Point с параметрами 'B' (имя), 5 и 8 (координатами по осям x и y соответственно), и этот вызов создает объект типа Point с соответствующими параметрами. Затем используется действие new, которое создает переменную-ссылку по имени **y** на только что созданный объект. Иными словами — создается экземпляр **y** класса Point
System.out.println(x);	Разумеется, нет никакой возможности «вывести на экран объект»! Эта команда вызывает (автоматически) метод toString класса Point — и на экран выводится та информация, и в том виде, как это описано в методе toString
System.out.println(quarta(x));	Отправляет переменную-ссылку **x** (экземпляр **x** класса Point) в качестве параметра в метод quatra класса **testPoint**, чтобы проверить, в какой четверти находится точка
System.out.println(isAxis(x));	Отправляет переменную-ссылку **x** (экземпляр **x** класса Point) в качестве параметра в метод isAxis класса **testPoint**, чтобы проверить, находится ли точка хотя бы на одной из осей координат
System.out.println(PandP(x,y));	Отправляет переменные-ссылки **x** и **y** (экземпляры **x** и **y** класса Point) в качестве параметров в метод PandP класса **testPoint**, чтобы проверить, находится ли точки в одной и той же четверти
System.out.println(howLong(x,y));	Отправляет переменные-ссылки **x** и **y** (экземпляры **x** и **y** класса Point) в качестве параметров в метод howLong класса **testPoint**, чтобы определить расстояние между точками

Работа с объектами классов

Поскольку значительная часть процесса создания аппликации для Android является работой с разными объектами и классами, необходимо получить определенный навык работы в этой области.

Мы предлагаем вам разобрать и самостоятельно сделать некоторое количество заданий на написание классов и работу с объектами, что, вне всякого сомнения, поможет вам в дальнейшем при работе с аппликациями для Android.

Прежде чем вы перейдете к самостоятельному выполнению заданий по работе с объектами, разберем несколько примеров.

Пример 1: «Простая дробь»

Напишем класс, который позволит нам работать с простыми дробями — как известно, такого типа значений в языке Java не существует, а вот потребность в работе с простыми дробями достаточно часто встречается в математических задачах.

Простая дробь — это, по сути дела, два целых числа, одно из которых является числителем дроби, а второе — знаменателем.

Вот класс, который описывает новый тип данных — «простая дробь».

```
import java.util.*;
class Rational
{
 static Scanner in=new Scanner(System.in);

 private int num_up;  //числитель дроби
 private int num_down;  //знаменатель дроби

 public Rational(int x, int y)
// конструктор объекта "простая дробь"
  {
   this.num_up=x;
   this.num_down=y;
  }

 public Rational()
// конструктор объекта "простая дробь"
  {
   this.num_up=in.nextInt();;
   this.num_down=in.nextInt();
  }
```

```
 public String toString()
//информационный метод
 {
  return this.num_up+"/"+this.num_down;
 }

 public int getNum_up()
//метод, возвращающий значение числителя
 {
  return this.num_up;
 }

 public int getNum_down()
//метод, возвращающий значение знаменателя
 {
  return this.num_down;
 }

 public Rational multiply(Rational num)
//произведение данной дроби и дроби-параметра
 {
  int x=this.num_up*num.num_up;
  int y=this.num_down*num.num_down;
  Rational c=new Rational(x,y);
 return c;
 }

 public Rational sum(Rational num)
//сумма данной дроби и дроби-параметра
 {
   int x=this.num_up*num.num_down+num.num_up*this.num_down;
   int y=this.num_down*num.num_down;
   Rational c=new Rational(x,y);
   return c;
  }

public Rational devide(Rational num)
//новая дробь, как результат деления данной дроби на дробь-параметр
 {
  int x=this.num_up*num.num_down;
  int y=this.num_down*num.num_up;
  Rational c=new Rational(x,y);
 return c;
 }
```

```
public Rational hisur(Rational num)
//новая дробь, как результат вычитания из данной дроби дроби-параметра
  {
    int x=this.num_up*num.num_down-num.num_up*this.num_down;
    int y=this.num_down*num.num_down;
    Rational c=new Rational(x,y);
    return c;
  }

public boolean equals(Rational num)
/*
Сравнение данной дроби с дробью-параметром
Возвращает true, если дроби равны; возвращает false, если дроби не равны
*/
    {
      return this.num_up==num.num_up && this.num_down==num.num_down;
    }

public int compareTo(Rational num)
/*
Сравнение данной дроби с дробью-параметром
Возвращает 0, если дроби равны
Возвращает 1, если данная дробь больше дроби-параметра
Возвращает -1, если данная дробь меньше дроби-параметра

*/
    {
     if(this.num_up*num.num_down>num.num_up*this.num_down)
       if(this.num_down*num.num_down>0)return 1; else return -1;

       if(this.num_up*num.num_down<num.num_up*this.num_down)
         if(this.num_down*num.num_down>0)return -1; else return 1;
       return 0;
    }

}
```

Обратите внимание, что в классе используются два метода-конструктора: один создает объект через параметры, передаваемые в метод-конструктор, а второй — через ввод значений числителя и знаменателя непосредственно в методе-конструкторе.

И еще — в классе, как вы заметили, отсутствуют метод set: нет смысла менять дробь, после того, как она создана.

А теперь — пример тестового класса, демонстрирующего использование объектов класса Rational.

```
class testRational
{

 public static void main(String[] args)
 {
  Rational a=new Rational();
  System.out.println("a="+a);
  Rational b=new Rational(2,4);
  System.out.println("b="+b);



    Rational x=a.multiply(b);
    System.out.println("a*b="+x);
    x=a.sum(b);
    System.out.println("a+b="+x);
     x=a.devide(b);
     System.out.println("a/b="+x);
     x=a.hisur(b);
     System.out.println("a-b="+x);
     System.out.println("a==b ? "+a.equals(b));
     System.out.println("a>b? "+a.compareTo(b));

 }
}
```

Пример 2: «Дата»

Напишем класс, который будет создавать объекты «дата» (день, месяц, год), — этот тип тоже отсутствует в языке Java, и с данными такого типа тоже часто приходится работать.

```
class Date
{
 private int day; //день — как целое число
 private int month; //месяц — как целое число
 private int year; //год — как целое число
```

```
 public Date(int d, int m, int y)
// конструктор, создающий объект по трем числовым параметрам
 {
  this.day=d;
  this.month=m;
  this.year=y;
 }
 public Date(Date other)
// конструктор, создающий объект по объекту-параметру типа "дата"
 {
  this.day=other.getDay();
  this.month=other.getMonth();
  this.year=other.getYear();
 }

 public int getDay()
// метод, возвращающий день
 {
  return this.day;
 }
 public int getMonth()
// метод, возвращающий месяц
 {
  return this.month;
 }
 public int getYear()
// метод, возвращающий год
 {
  return this.year;
 }
 public void setDay(int x)
// метод, устанавливающий день
 {
  this.day=x;
 }
 public void setMonth(int x)
// метод, устанавливающий месяц
 {
  this.month=x;
 }
```

```
  public void setYear(int x)
// метод, устанавливающий год
   {
    this.year=x;
   }

  public int compareTo(Date other)
/*
Метод, сравнивающий данную дату с датой параметром
Возвращает 0, если даты совпадают
Возвращает -1, если данная дата предшествует дате-параметру
Возвращает 1, если дата-параметр предшествует данной дате
*/
  {
   if(this.year<other.year)return -1;
     if(this.year>other.year)return 1;
       if(this.month<other.month)return -1;
     if(this.month>other.month)return 1;
         if(this.day<other.day)return -1;
     if(this.day>other.day)return 1;
     return 0;
  }

  public String toString()
//информационный метод
  {
   return this.day+"."+this.month+"."+this.year;
  }
}
```

На этот раз мы приведем в качестве примеров тестовых программ два примера: простой и посложнее.

```
class testData
{
   public static void main(String[] args)
  {
     Date d=new Date(12,5,2008);
     Date d1=new Date(12,5, 2007);
     System.out.println(d.toString()+" & "+d1.toString());
     System.out.println(d.compareTo(d1));
  }
}
```

```
import java.util.*;
class testDate1
{
 public static boolean same(Date d1,Date d2)
 {
  int a1=d1.getMonth();
  int b1=d2.getMonth();
  int a2=d1.getDay();
  int b2=d2.getDay();
  if(a1!=b1)return false;
  if(a2!=b2)return false;
  return true;
 }

 public static void main(String[]args)
 {
  Scanner in=new Scanner(System.in);
  Date x;
  Date y;
  int a=in.nextInt();
  int b=in.nextInt();
  int c=in.nextInt();
  x=new Date(a,b,c);
  y=new Date(21,06,1995);
  System.out.println(x);
  System.out.println(y);
  int z=y.getYear();
  System.out.println(z);
  y.setYear(2005);
  System.out.println(y);
  System.out.println(y.toString());
  x.setDay(21);
  x.setMonth(6);
  x.setYear(1995);
  System.out.println(x);
  boolean f=same(x,y);
  System.out.println(f);
  f=x.equals(y);
  System.out.println(f);
  System.out.println(x.compareTo(y));
 }
}
```

Пример 3: «Паспортные данные»

Третий пример — класс, описывающий паспортные данные:

```
class Passport
{
 private String name;  //имя владельца паспорта
 private int number;  //номер паспорта
 private Date limitDate;  //дата завершения срока действия паспорта

public Passport(String n, int num, Date d)
// конструктор, создающий объект-паспорт
  {
   this.name=n;
   this.number=num;
   this.limitDate=d;
  }

public void setLimitDate(Date other)
/*
 установка срока окончания действия паспорта с помощью объекта-
параметра типа Data
*/
  {
    this.limitDate=other;
  }

public boolean isValid(Date other)
/*
Проверяет, будет ли паспорт действителен на дату, указанную в объ-
екте-параметре типа Data
*/
 {
  if(this.limitDate.compareTo(other)>0)return true;
  else return false;
 }

public String toString()
// информационный метод
  {
   String s="";
   s=s+this.name+"\n";
   s=s+this.number+"\n";
```

```
  s=s+this.limitDate.toString();
  return s;
 }
}
```

Обратите внимание, что в этом примере используется в качестве одного из составляющих описание типа Passport тип, созданный в предыдущем примере (Data). Ну и, разумеется — пример тестового класса.

```
class testPassport
{
  public static void main(String[] args)
  {
   Date d=new Date(12,5,2008);
   Passport p=new Passport("Сергей", 121,d);

   Date d1=new Date(12,5, 2009);
   System.out.println(p.isValid(d1));

   System.out.println(p);
   d.setYear(2013);
    System.out.println(p);
    System.out.println(p.isValid(d1));
  }
}
```

Задания на рассмотренные примеры

Общее замечание: разумеется, что при работе разобранными выше примерами вполне допустимо использование, например, и массивов: Date[] d=new Date[10], да и вообще всего известного вам инструментария.

Задание 1

Напишите тестовый класс, который будет заполнять массив случайными датами рождения (массив типа Data). Заполнение можно выполнить с использование генератора случайных чисел (дни — от 1 до 31, месяцы — от 1 до 12, годы — от 1900 до 2011), а можете выбрать и другие варианты заполнения массива, — например, через введение данных с клавиатуры.

Затем добавьте в класс следующие методы:

1. Метод, выводящий на экран содержание массива.
2. Метод, проверяющий, сколько дат в массиве являются ошибочными.
3. Метод, исправляющий ошибочные даты.

4. Метод, подсчитывающий, сколько человек родились в определенное временя года.
5. Метод, подсчитывающий, сколько человек рождены в один и тот же определенный месяц.
6. Метод, подсчитывающий, сколько человек рождены в XX-м, и, сколько — в XXI вв.
7. Метод, подсчитывающий, сколько человек рождены ранее определенной даты.
8. Метод, подсчитывающий, сколько человек на определенную дату достигли определенного возраста.

Задание 2

Напишите тестовый класс, который заполняет массив типа Rational положительными дробями (размер массива и дроби выберите либо произвольно, с помощью генератора случайных чисел, либо введите с клавиатуры).

Затем добавьте в класс следующие методы:

1. Метод, выводящий на экран содержание массива.
2. Метод, вычисляющий сумму массива.
3. Метод, проверяющий, сколько в массиве неправильных дробей (каждую такую дробь надо вывести на экран с указанием на ее место в массиве).
4. Метод, принимающий в качестве параметра дробь, и проверяющий, сколько в массиве дробей с таким же знаменателем.
5. Метод, заполняющий новый массив произведениями смежных дробей из массива, который метод получает в качестве параметра.
6. Метод, заполняющий новый массив делением смежных дробей из массива, который метод получает в качестве параметра.
7. Метод, который проверяет, имеются ли в «параллельных» (с одинаковыми индексами) ячейках массивов, заполненных методами 5 и 6 одинаковые дроби.

Задание 3

Напишите тестовый класс, который заполняет массив типа Point точками, лежащими в прямоугольной системе координат (размер массива и дроби выберите либо произвольно, с помощью генератора случайных чисел, либо введите с клавиатуры).

Затем добавьте в класс следующие методы:

1. Метод, выводящий на экран содержание массива.
2. Метод, выводящий на экран список точек, лежащих в определенной четверти (указание на четверть метод принимает в качестве параметра).
3. Метод, выводящий на экран список точек, лежащих на одной из осей системы координат.

4. Метод, принимающий в качестве параметра радиус окружности с центром в начале системы координат, и выводящий на экран список точек, лежащих внутри этой окружности.
5. Метод, принимающий в качестве параметра некое число, и выводящий на экран пары точек, расстояние между которыми равно величине-параметру.

Задание 4

Для обработки статистической информации о школе решено написать класс на языке Java, который будет использовать объекты типа learning_group (не будем использовать слово class во избежание проблем), характеризуемые следующими свойствами:

- Номер параллели (целое число от 1 до 11).
- Буквенное обозначение класса (символ, буква).
- Специализацию класса (строковое значение). В случае если класс не имеет специализации, этот параметр содержит значение «отсутствует».
- Число мальчиков в классе (целое число).
- Число девочек в классе (целое число).
- Среднюю годовую оценку по классу (десятичная дробь).

A. Напишите класс для объектов описанного типа.

B. Напишите тестовую программу, которая будет содержать следующие методы:

   1. Метод, выводящий на экран информацию по всем классам (каждая параллель — в отдельной строке).
   2. Метод, определяющий, в какой параллели больше всего классов.
   3. Метод, определяющий, в какой параллели больше всего учеников.
   4. Метод, определяющий, в каких параллелях больше всего девочек.
   5. Метод, вычисляющий среднюю общешкольную годовую оценку.
   6. Метод, определяющий, сколько в каждой параллели классов, в которых средняя годовая оценка выше средней годовой оценки по школе.

Задание 5

Данные типа «песня» (Song) описывают следующие свойства объекта:

- название песни (name);
- исполнитель (performer);
- продолжительность (length), в секундах.

Дана таблица (Unified Modeling Language — унифицированный язык моделирования) для класса Song:

String name	Свойства
String perfomer	
int length	
Song(String n, String p, int len)	Метод-конструктор
String getName()	Методы, возвращающие значения
String getPerfomer() менее 2-х минут	
int getLength()	
void setName(String n)	Методы, устанавливающие значения
void setPefomer(String p)	
void setLength(int len)	
String toString()	Информационный метод

1. Напишите класс для объектов указанного типа.
2. Напишите метод Category, который получает в качестве параметра объект Song и возвращает одно из следующих значений: short (если продолжительность песни менее 2-х минут), long (если продолжительность песни более 4-х минут), medium (если продолжительность менее 4-х, но более 2-х минут).
3. Напишите метод, который получает в качестве параметров два объекта типа Song и проверяет, относятся ли они к одной и той же категории (возвращает либо true, либо false).

Задание 6

Данные типа «диск» (Disk) описывают следующие свойства объекта:

- название диска (name);
- массив объектов Song.

Дана таблица (Unified Modeling Language — унифицированный язык моделирования) для класса Song:

String name	Свойства
Song [] songs	
Disc (String n, Song[] s)	Метод-конструктор
String getName()	Методы, возвращающие значения
Song[] getSongs()	
void setName(String n)	Методы, устанавливающие значения
void setSongs(Song[] s)	
String toString()	Информационный метод

1. Напишите класс для объектов указанного типа.
2. Напишите метод diskLength, который возвращает продолжительность звучания всех песен диска.
3. Напишите метод nameOfLongestSong, который возвращает название самой длинной песни диска.
4. Напишите метод displaySongsInCategory(String cat), который выводит на экран все песни диска, относящиеся к категории cat.
5. Напишите тестовый класс, который будет создавать объект типа Disk, заполнит его данными, выведет их на экран, а также реализует методы из предыдущих пунктов задания.

Задание 7

Данные типа «бусинка» Bead описывают следующие свойства объекта:

- Цвет (одно из строковых значений — «красный», «зеленый», «желтый», «черный»).
- Номер материала, из которого она сделана (целое число от 1 до 10).
- Отшлифованная или нет (true/false).

Дана таблица (Unified Modeling Language — унифицированный язык моделирования) для класса Song:

String name	Свойства
int material	
boolean polish	
Bead (String n, int[] m, boolean p)	Метод-конструктор
String getName()	Методы, возвращающие значения
int getMaterial()	
boolean getPolish()	
void setName(String n)	Методы, устанавливающие значения
void setMaterial(int m)	
Void setPolish(Boolean p)	
String toString()	Информационный метод

1. Напишите класс для объектов указанного типа.

Задание 8

Назовем «ожерельем» массив объектов типа «бусинка».

Ожерелье считается «совершенным», если выполняются следующие условия:

- Цвета первой и последней бусинок совпадают.
- Нет ни одной пары соседних бусинок одного цвета.
- Рядом с отшлифованной бусинкой должна быть неотшлифованная — и наоборот.

Ожерелье называется «разноцветным», если в нем встречаются бусинки всех цветов.

Ожерелье называется «скромным», если в нем есть бусинки только одного цвета.

Напишите тестовый класс, который заполняет массив типа Bead соответствующими значениями (создает «ожерелье»), выводит его на экран, а затем проверяет, соответствует ли «ожерелье» описанным выше определениям.

Иерархия и наследование

Разумеется, в таком широко используемом и современном языке как Java должны присутствовать возможности не только написания каких-то классов решения каких-то задач, но и возможности использования аналогичных решений, написанных прежде.

Одним из вариантов такого использования «прежнего опыта» является особенность языка Java, известная как «наследование классов», или просто «наследование» (inheritance). Основано наследование на принципе иерархии, — то есть на том факте, что одни классы являются подчиненными (или, как принято говорить — потомками) другого класса (или — других классов).

В обычной жизни с иерархией мы сталкиваемся постоянно, в определенном смысле можно сказать, что иерархия в значительной степени делает нашу жизнь более упорядоченной, более понятной, даже четче работающей.

Примеры иерархии привести легко: семья, офис, школа, армия, государственное устройство, спортивная команда... Все эти примеры являются примера «иерархии прав и обязанностей», но, кроме этого, можно привести примеры «иерархии наследования», когда «предки» являются более общим представлением «потомков».

Один из наиболее часто используемых в качестве иллюстрации «иерархии наследования» примеров — классификация животного мира.

Вот, например, классификация (частичная) ядовитых животных:

Вот еще один пример — классификация авиационных двигателей:

АВИАЦИОННЫЕ ДВИГАТЕЛИ
По способу создания тяги
Винтовые
Реактивные
Поршневые (ПД)
Турбовинтовые (ТВД)
Воздушные
Ракетные
Охлаждение цилиндров
Воздушное
Жидкостное
Расположение цилиндров
Расположение рядов цилиндров
Звездообразное
Рядное
V-образное
W-образное
X-образное
Многорядная звезда
Компрес-сорные
Бескомп-рессорные
По виду привода
Мотореактивные
Турбореактивные (ТРД)
Пульсирующие (ПуД)
Прямоточные (ПД)
Тип компрессора
Центробеж-ный
Осевой
Смешанной конструкции
Жидкостные ракетные (ЖРД)
Ракетные твердого топлива (РДТТ)
На самолетах используются в качестве вспомогательных двигателей (стартовые ускорители)

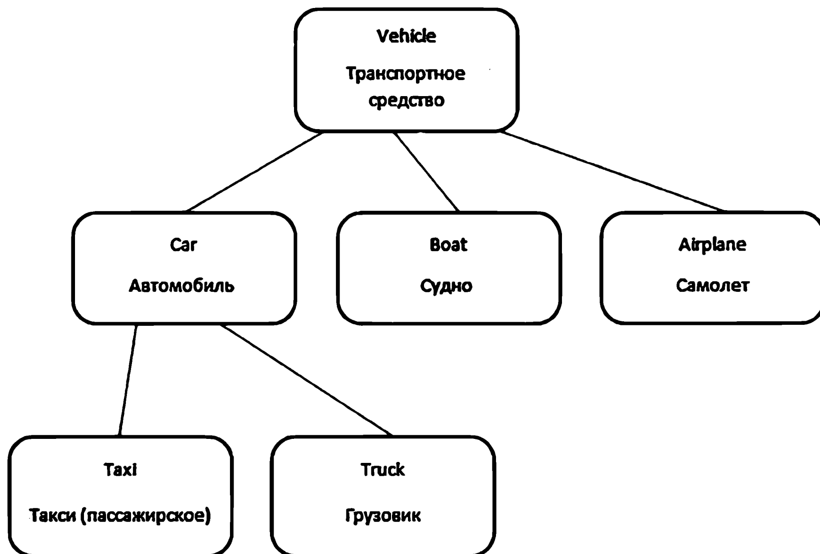
Вы можете самостоятельно составить аналогичные схемы.

Пример «Транспортные средства»

Для того чтобы научиться использовать наследование, разберем пример, с помощью которого покажем, как это реализуется в Java.

Компания грузопассажирских перевозок «Бери и вези» использует следующие транспортные средства: легковые и грузовые автомобили, речные суда и самолеты.

Первым делом составим иерархическую таблицу.



С точки зрения языка Java такая иерархическая схема означает следующее:

1. Надо будет создать следующие классы: Vehicle, Car, Boat, Airplane, Taxi, Truck.
2. Классы Car, Boat, Airplane наследуют классу Vehicle (являются его наследниками). В этом случае класс Vehicle является «суперклассом» (superclass), а классы Car, Boat, Airplane являются «подклассами» (subclass, субклассами).
3. В свою очередь класс Car является суперклассом для подклассов Taxi и Truck.

Для суперкласса Vehicle введем следующие свойства:

- type (среда, в которое передвигается данное транспортное средство — это будет свойства типа String, которое будет принимать одно из следующих значений: land, sky, water);

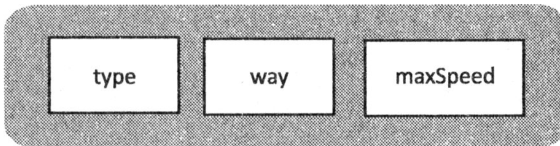
- way (вид транспортной инфраструктуры, которую будет использовать транспортное средство — это будет свойства типа String, которое будет принимать одно из следующих значений: road, river, sea, air);
- maxSpeed (максимальная скорость транспортного средства — это будет свойство типа int, выражающее скорость в км/час).

Напишем на языке Java класс для работы с объектами Vehicle, и внесем в него пока только один метод — конструктор.

```
public class Vehicle
{
        private String type;
        private String way;
        private int maxSpeed;

        public Vehicle (String type, String way, int maxSpeed)
        {
                this.type=type;
                this.way=way;
                this.maxSpeed=maxSpeed;
        }
}
```

Для наглядности можно изображать экземпляры класса Vehicle примерно так:



Для каждого конкретного экземпляра в прямоугольниках вместо названий свойств будем писать значения этих свойств.

Созданный нами класс пока позволяет делать только одно: создавать объекты типа Vehicle. В случае если у вас будет желание дополнить этот класс get/set-методами и методом toString — сделайте это. Для задачи, которую мы сейчас перед собой поставили — разобраться с наследованием — эти методы не существенны (пока).

Для демонстрации того, как можно создавать экземпляры класса Vehicle — напишем простой тестовый класс (Test_0):

```
import java.util.*;
class Test_0
{
 static Scanner reader=new Scanner(System.in);
```

```
  public static void main(String[] args)
  {
    String type=reader.next();
    String way=reader.next();
    int maxSpeed=reader.nextInt();
    Vehicle auto=new Vehicle(type, way, maxSpeed);
    Vehicle transport=new Vehicle("land","road",100);

    int v=reader.nextInt();
    int n=reader.nextInt();
    Car mobil=new Car(v,n);

  }
}
```

Обратите внимание, что при вводе данных с клавиатуры, проверка вводимых значений не выполняется — разумеется, в этом случае можно ввести любое значение, в том числе и не соответствующее тем, которые были оговорены выше. Мы предлагаем вам дополнить класс Test_0 соответствующими проверками.

Субклассы Car, Boat, Airplane мы будем строить по следующим правилам:

- Каждый из них будет наследовать свойства своего суперкласса (Vehicle). Для этого в заголовках субклассов мы используем служебное слово extends и указание на имя суперкласса.
- У каждого субкласса будут свои собственные свойства (хотя бы по одному), характеризующие специфику этого субкласса:
  - у субкласса Car будет свойство numOfWheel (число колес);
  - у субкласса Boat не будет собственных (дополнительных) свойств;
  - у субкласса Airplane будет свойство maxHeight (максимальная высота полета).

Напишем класс для объектов Car:

```
public class Car extends Vehicle
{
 private int numOfWheel;

 public Car(int maxSpeed, int numOfWheel)
 {
   super("land","road", maxSpeed);
   this.numOfWheel=numOfWheel;
 }
}
```

В этот класс мы пока включили только метод-конструктор Car, который имеет следующие особенности:

- Получает в качестве параметра два значения, первое из которых — скорость «конкретного» автомобиля (экземпляра класса Car), а второе — количество колес этого автомобиля.
- «Автомобиль» должен иметь значения параметров type, way, maxSpeed, причем параметры эти он должен унаследовать от своего суперкласса, от Vehicle.

  Однако мы не можем воспользоваться в этом случае способами this.type, this.way, this.maxSpeed, потому что соответствующие свойства (type, way, maxSpeed) описаны в классе Vehicle как private, — то есть к ним нет доступа из других классов.

  Для разрешения возникающего противоречия («нужны параметры — их нельзя установить») в языке Java используется оператор super.
- Количество колес устанавливается в соответствии с полученным методом параметром numOfWheel.

Добавим в тестовый класс Test_0 строки кода для создания экземпляра класса car.

```
int v=reader.nextInt();
int n=reader.nextInt();
Car mobil=new Car(v,n);
```

Теперь напишем класс для объектов Boat:

```
public class Boat extends Vehicle
{
  public Boat(String way, int maxSpeed)
  {
    super("water",way,maxSpeed);
  }
}
```

Дополним класс Test_0 командами для создания экземпляров и этого класса:

```
way=reader.next();
maxSpeed=reader.nextInt();
Boat ship_1=new Boat(way, maxSpeed);
Boat ship_2=new Boat("Volga",10);
```

Поскольку у объектов класса Boat, в соответствие с нашим решением, нет собственных свойств (все свои свойства он наследует от своего класса-

«родителя», то есть суперкласса Vehicle) — обратите внимание, что в методе-конструкторе присутствует только использование метода super.

И, наконец, напишем класс для объектов Airplane:

```
public class Airplane extends Vehicle
{
 private int maxHeight;

 public Airplane(int maxSpeed, int maxHeight)
 {
   super("sky","air", maxSpeed);
   this.maxHeight =maxHeight;
 }
}
```

Обратите внимание, что конструктор в этом классе построен аналогично конструктору класса Car.

Дополним тестовый класс строками кода и для экземпляров класса Airplane.

```
maxSpeed=reader.nextInt();
int maxHeight=reader.nextInt();
Airplane plane_1=new Airplane(maxSpeed, maxHeight);
Airplane plane_2=new Airplane(500, 2000);
```

Разумеется, нам хотелось бы посмотреть, как вся эта система классов будет работать, — то есть получить возможность выводить параметры каждого экземпляра каждого класса на экран.

Для этого дополним каждый из классов методом toString:

Класс	Метод toString

Vehicle	public String toString()
	{
	return "("+type+" & "+way+" & "+maxSpeed+")";
	}

Car	public String toString()
	{
	return "Car: "+super.toString()+"^"+this.numOfWheel;
	}

Окончание таблицы

Класс	Метод toString

Boat	public String toString()
	{
	return "Boat: "+super.toString();
	}

Airplane	public String toString()
	{
	return "Airplane: "+super.toString()+"^"+this.maxHeight;
	}

Дополним наш тестовый класс Test_0 соответствующими командами — и теперь он будет выглядеть следующим образом:

```
import java.util.*;
class Test_0
{
 static Scanner reader=new Scanner(System.in);

  public static void main(String[] args)
  {
    String type=reader.next();
    String way=reader.next();
    int maxSpeed=reader.nextInt();
    Vehicle auto=new Vehicle(type, way, maxSpeed);
    Vehicle transport=new Vehicle("land","road",100);

    System.out.println(auto.toString()+" "+transport.toString());

    int v=reader.nextInt();
    int n=reader.nextInt();
    Car mobil=new Car(v,n);

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

    way=reader.next();
    maxSpeed=reader.nextInt();
    Boat ship_1=new Boat(way, maxSpeed);
    Boat ship_2=new Boat("Volga",10);

    System.out.println(ship_1.toString()+" "+ship_2.toString());
```

```
        maxSpeed=reader.nextInt();
        int maxHeight=reader.nextInt();
       Airplane plane_1=new Airplane(maxSpeed, maxHeight);
       Airplane plane_2=new Airplane(500, 2000);

       System.out.println(plane_1.toString()+" "+plane_2.toString());
   }
}
```

Замещение (overriding)

Для того чтобы познакомиться с еще одним чрезвычайно важным и полезным механизмом, присутствующим в языке Java, — «замещением» («подавлением») — продолжим работу с нашими классами.

Рассмотрим следующие строки кода:

```
Car auto = new Car(100,6);
Vehicle v_auto = new Car(300,2);
System.out.println(auto.toString());
System.out.println(v_auto.toString());
```

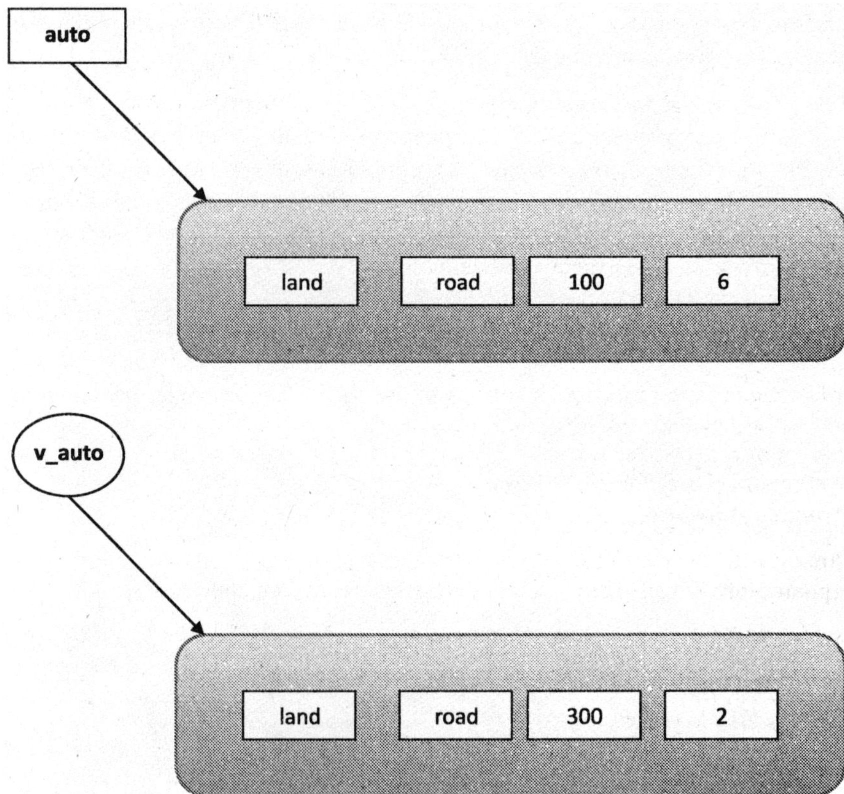
Опишем подробно события, которые происходят при исполнении первой строки этого кода:

1. Создается объект класса Car.
2. При этом два свойства объекта (land и road) наследуются от суперкласса, потому что именно они указаны в конструкторе класса Car в команде super.
3. Третье свойство тоже наследуется от суперкласса, но при этом его значение устанавливается (заменяется, «подавляется») собственным параметром субкласса (100).
4. У объекта также формируется дополнительное, четвертое, свойство (6).

События, которые происходят при исполнении второй строки, от перечисленных выше четырех пунктов совершенно ничем не отличаются, по крайней мере, на первый взгляд:

1. Создается объект класса Car
2. При этом два свойства объекта (land и road) наследуются от суперкласса, потому что именно они указаны в конструкторе класса Car в команде super.
3. Третье свойство тоже наследуется от суперкласса, но при этом оно заменяется («подавляется») собственным параметром субкласса (300).
4. У объекта также формируется дополнительное, четвертое, свойство (2).

Однако легко заметить и разницу: в первом случае на объект класса Car указывает переменная типа Car, а во втором случае на объект класса Car указывает переменная типа Vehicle (поясним это рисунками).



Обратите внимание: переменную v_auto мы заключили в овал — этим мы подчеркиваем, что она принадлежит к иному(!) классу, чем объект, на который она указывает.

С третьей строкой в нашем примере тоже никаких проблем нет: и переменная, и объект принадлежат к одному и тому же классу (Car) и достаточно очевидно, что исполняться будет метод toString, описанный в этом же классе.

Но вот в отношении четвертой строки в нашем примере возникает вопрос: при исполнении **System.out.println(v_auto.toString());** — будет исполняться метод toString, описанный в классе Vehicle, или метод toString, описанный в классе Car?

Напишем еще один тестовый класс (Test_1) и впишем в него эти строки кода.

Сделайте компиляцию класса Test_1 и запустите его на исполнение: результат свидетельствует о том, что в строке **System.out.println(v_auto.toString());** используется метод toString класса Car, а не класса Vehicle.

Такой механизм называется в языке Java «замещение» («подавление»): субкласс, наследуя свойства у суперкласса, использует собственные методы, если в субклассе и в суперклассе есть два метода с одинаковыми заголовками. При этом если на объект субкласса указывает переменная суперкласса (**Vehicle v_auto = new Car(300,2);**), предпочтение при выборе — классу объекта, а не классу переменной.

Обратите внимание: механизм замещения включается в том случае, если речь идет о методах с **одинаковыми заголовками**.

Для иллюстрации этого добавим в классы Car и Vehicle по методу с одинаковым заголовком, но разным содержанием.

В класс Vehicle добавим метод

```
public void Message()
   {
      System.out.println("This is a message from Vehicle class");
   }
```

а в класс Car добавим метод

```
public void Message()
   {
      System.out.println("This is a message from Car class");
   }
```

Теперь в класс Test_1 добавим следующие строки кода:

Vehicle[] v = {new Vehicle("land","road",200),new Vehicle("sky","air",500),new Vehicle("water","river",50)}; // массив объектов класса Vehicle

auto.Message(); //применить метод Message() к переменной типа Car

v_auto.Message();//применить метод Message() к переменной типа Vehicle

v[0].Message();//применить метод Message() к переменной типа Vehicle

При исполнении класса Test_1 мы обнаружим, что для этих строк вывод на экран будет выглядеть так:

This is a message from Car class
This is a message frcm Car class
This is a message from Vehicle class

Чрезвычайно важно обратить внимание на то, что при исполнении строки **v_auto.Message();** метод класса Vehicle (к которому принадлежит переменная v_auto) оказался замещенным методом класса Car (к которому принадлежит объект, на который указывает переменная v_auto)!

Однако такое однозначное замещение выполняется только в том случае, когда в суперклассе и субклассе присутствуют методы с **одинаковыми заголовками**.

А что будет происходить, если в тестовой программе осуществляются вызовы методов с разными заголовками, описанные в разных классах? Разберемся с этим с помощью дополнительных методов.

Добавим в класс Vehicle вот такой метод:

```
public void Message_vehicle()
  {
    System.out.println("I'm Vehicle: "+this.maxSpeed");
  }
```

А в класс Test_1 впишем вот такие строки:

```
auto.Message_vehicle();
v_auto.Message_vehicle();
v[0].Message_vehicle();
```

Напомним, к каким типам принадлежат переменные, и на объекты какого класса они указывают.

Переменная	Тип переменной	Указывает на объект класса
auto	Car	Car
v_auto	Vehicle	Car
v[0]	Vehicle	Vehicle

Код **auto.Message_vehicle();** исполняется для переменной **auto**, поэтому в поисках метода **Message_vehicle()** язык Java первым делом выполняет обращение к классу **Car**. Не обнаружив этого метода в классе **Car,** язык выполняет обращение к суперклассу, то есть к классу **Vehicle**, в котором указанный метод и обнаруживается. В результате на экран выводится сообщение **I'm Vehicle: 100**, в котором значение 100 взято из свойств того объекта, на который указывает переменная **auto**.

Код **v_auto.Message_vehicle();** исполняется для переменной **v_auto**, поэтому в поисках метода **Message_vehicle()** язык Java первым делом выполняет обращение к классу **Vehicle**, в котором указанный метод и обнаруживается. В результате на экран выводится сообщение **I'm Vehicle: 300**, в котором значение 300 взято из свойств того объекта, на который указывает переменная **v_auto**.

Код **v[0].Message_vehicle();** исполняется для переменной **v[0]**, поэтому в поисках метода **Message_vehicle()** язык Java первым делом выполняет обращение к классу **Vehicle**, в котором указанный метод и обнаруживается. В результате на экран выводится сообщение **I'm Vehicle: 200** — в котором значение 300 взято из свойств того объекта, на который указывает переменная **v[0]**.

Обратите внимание: обращение из субкласса к суперклассу в поисках нужного метода язык Java выполняет автоматически.

А теперь в класс Car добавим похожий, но все же иной метод:

```
public void Message_car()
  {
    System.out.println("I'm Car: "+this.maxSpeed);
  }
```

Отличие, как видите, в заголовке.

Теперь добавим в класс Test_1 еще три строки кода — тоже очень похожие на те, которые мы только что разобрали:

```
auto.Message_car();
v_auto.Message_car();
v[0].Message_car();
```

Однако уже на этапе компиляции мы получим (для строки **v_auto.Message_car();**) сообщение об ошибке, смысл которой: для переменной **v_auto** не существует метода **Message_car()**!

И если мы уберем это строку, то совершенно такое же сообщение об ошибке мы получим для строки **v[0].Message_car();**!

Причина проблемы в том, что язык **никогда** не выполняет поиск метода в субклассе, если метод отсутствует в суперклассе. К суперклассу **Vehicle** обращаются вторая и третья строки, потому что переменные **v_auto** и **v[0]** относятся к этому суперклассу.

Поэтому категорически недопустимо применять к переменным суперкласса методы, описанные в субклассе и не описанные в суперклассе.

Еще один пример замещения можно продемонстрировать следующим образом:

1. Создадим в классе Vehicle get-методы (возвращающие значения соответствующих свойств).
2. Создадим в классе Car еще два метода-конструктора, которые будут получать в качестве одного из параметров объект (экземпляр) класса Vehicle.
3. Используя метод toString, выведем (в тестовом классе Test_1) на экран значения свойств соответствующих экземпляров класса Car.

Мы приведем полные тексты (коды) соответствующих классов и предоставим вам самостоятельно разобраться с тем, как они построены и исполняются.

Задания для самостоятельной работы

Как вы помните, в нашем примере у суперкласса Vehicle, кроме субкласса Car, есть еще два субкласса: Boat и Airplane.

Задание 1

Напишите для класса Boat:

- метод toString;
- метод, выводящий на экран текстовое сообщение **Скорость судна** = и значение максимальной скорости судна.

Задание 2

Напишите для класса Airplane:

- метод toString;
- метод, выводящий на экран текстовое сообщение **Высота полета** = и значение максимальной высоты полета.

Задание 3

Напишите отдельный тестовый класс, который будет создавать переменные типа Vehicle, Boat и Airplane, и объекты этих же классов. Часть переменных и связанных с ними объектов должны быть одного типа (класса), часть — разного.

Тестовый класс должен продемонстрировать исполнение разных методов.

Задание 4

Класс Car является суперклассом для классов Taxi и Truck. Напишите для этих трех классов методы, аналогичные методам, сформулированным в заданиях 1, 2.
Напишите для этих трех классов тестовый класс.

Задание 5

Напишите тестовый класс для классов Vehicle, Car, Taxi.

Полиморфизм

Одной из самых сильных возможностей при работе с объектами (наравне с наследованием) является полиморфизм, позволяющий к одному и тому же объекту отнестись по-разному, в зависимости от того, с помощью каких переменных и каких методов мы используем объект.

Говоря простыми словами, все зависит от «точки зрения», потому что каждый объект может быть не только объектом своего класса, но и соответствующих суперклассов.

Например, конкретный «грузовик» (объект класса Truck) одновременно является и «автомобилем» (суперкласс Car для класса Truck), и «транспортным средством» (суперкласс Vehicle для класса Car).

Полиморфизм позволяет по-разному отнестись и к группе объектов, в случае если это объекты разных подклассов, связанные с общим суперклассом.

Например, группа, состоящая из «такси», «грузовика» и «самолета», образует группу из 3-х «транспортных средств», но, одновременно, группу из только 2-х «автомобилей».

В случае если мы создаем объект «автомобиль», то к нему можно обращаться и как к объекту «автомобиль», и как к объекту «транспортное средство».

Таким образом, полиморфизм («множественность форм») делает классы на языке Java гораздо более гибкими и эффективными.

Напишем тестовый класс Test_2, с помощью которого мы будем разбираться с тем, как в языке Java выполнять практическое использование полиморфизма.

Пока листинг класса выглядит так:

```
class Test_2
{
  public static void main(String[] args)
  {
    Vehicle[] transport=new Vehicle[4];
    transport[0]=new Vehicle("land","road",200);
    transport[1]=new Vehicle("sky","air",500);
    transport[2]=new Vehicle("water","river",50);
    transport[3]=new Vehicle("wate","sea",99);
     for (int i=0;i<transport.length;i++)
         System.out.println(transport[i].toString());

     System.out.println("=======");

    Vehicle[] carriage=new Vehicle[4];
    carriage[0]=new Vehicle("land","road",150);
    carriage[1]=new Car(250,2);
    carriage[2]=new Boat("sea",88);
    carriage[3]=new Airplane(800,3000);
     for (int i=0;i<carriage.length;i++)
         System.out.println(carriage[i].toString());
  }
}
```

Результат выполнения класса будут выглядеть следующим образом:

```
(land & road & 200)
(sky & air & 500)
(water & river & 50)
(wate & sea & 99)
=======
(land & road & 150)
Car: (land & road & 250)^2
Boat: (water & sea & 88)
Airplane: (sky & air & 800)^3000
```

Обратите внимание, что свойства массива **transport** выводятся на экран как свойства экземпляров класса Vehicle, а свойства массива **carriage** выводятся на экран как свойства экземпляров разных(!) классов. Иными словами, для элементов массива transport каждый раз вызывается метод toString класса Vehicle, а для элементов массива carriage каждый раз вызывается метод toString из иного класса.

Теперь поставим перед собой такую задачу: использовать один из элементов второго массива не как объект подкласса, а как объект суперкласса, — такое действие называется «преобразование вверх», «конверсия вверх», «приведение вверх» (up casting). Например, в экземпляре класса Boat (элемент в ячейке № 2 массива carriage) мы решим изменить свойство way со значения sea на значение Atlantic ocean.

Сделать это, обращаясь к экземпляру класса Boat, — невозможно: напомним, что, хотя такое свойство у этого экземпляра есть (посмотрите, что его можно вывести на экран), но доступ к нему закрыт, — экземпляр его наследует от своего суперкласса, а в суперклассе оно закрыто на доступ, потому что объявлено как private.

Зато мы можем использовать тот факт, что на этот экземпляр класса Boat указывает переменная класса Vehicle, а именно — переменная carriage[2]. И изменить это свойство можно с помощью метода setWay, который мы впишем в класс Vehicle (смотрите листинг класса).

Теперь можно добавить в тестовый класс Test_2 две строки и с их помощью убедиться, что «преобразование вверх» действительно исполняется.

```
carriage[2].setWay("Atlantic ocean");
System.out.println(carriage[2]);
```

Разумеется, кроме «преобразования вверх» существует и «преобразование вниз», когда с объектами суперкласса нужно работать как с объектами подкласса.

В принципе, «преобразование вниз» выполняется по такому же правилу, по которому выполнялись преобразования между переменными типа int и double в базовом курсе «компьютерные науки».

Например, так:

```
Boat temp=(Boat)carriage[i];
```

Но, в отличие от «преобразования вверх», которое принципиально возможно всегда, «преобразование вниз» создает массу проблем. Давайте разберемся, откуда эти проблемы берутся.

Сформулируем два утверждения:

1. Любое транспортное средство является самолетом.
2. Любой самолет является транспортным средством.

Несложно понять, что первое утверждения может быть верным, но далеко не всегда, а тогда и только тогда, когда транспортное средство, в самом деле, является самолетом. Но транспортным средством является и автомобиль, и корабль, — но при этом ни автомобиль, ни корабль самолетом не являются.

А вот второе утверждение — верно всегда.

Что это означает с точки зрения языка Java? Это означает, что:

1. «Приведение вниз», являясь **всегда** теоретически верным, для одних конкретных случаев может оказаться **практически исполнимым**, а для других — **практически неисполнимым.** В более технических терминах: «преобразование вниз» всегда благополучно проходит этап компиляции, но может вызвать срыв на этапе исполнения (Run time error). Разумеется, что такая ситуация никого не может устроить: для чего писать код, про который неизвестно, то ли будет исполняться, то ли нет. О том, как в языке Java можно справиться с этой проблемой — ниже.
2. «Приведение вверх», которое благополучно прошло этап компиляции, **никогда** не создает проблем на этапе исполнения.

Завершим это короткое обсуждение следующим образным сравнением:

1. Если у транспортного средства нет крыльев, то сами они никогда не вырастут.
2. Если у самолета отломать крылья, то самолетом он уже не будет, но как транспортное средство его можно использовать (катить по дороге, например, — колеса-то остались, или использовать гидросамолет с отломанными крыльями как лодку...).

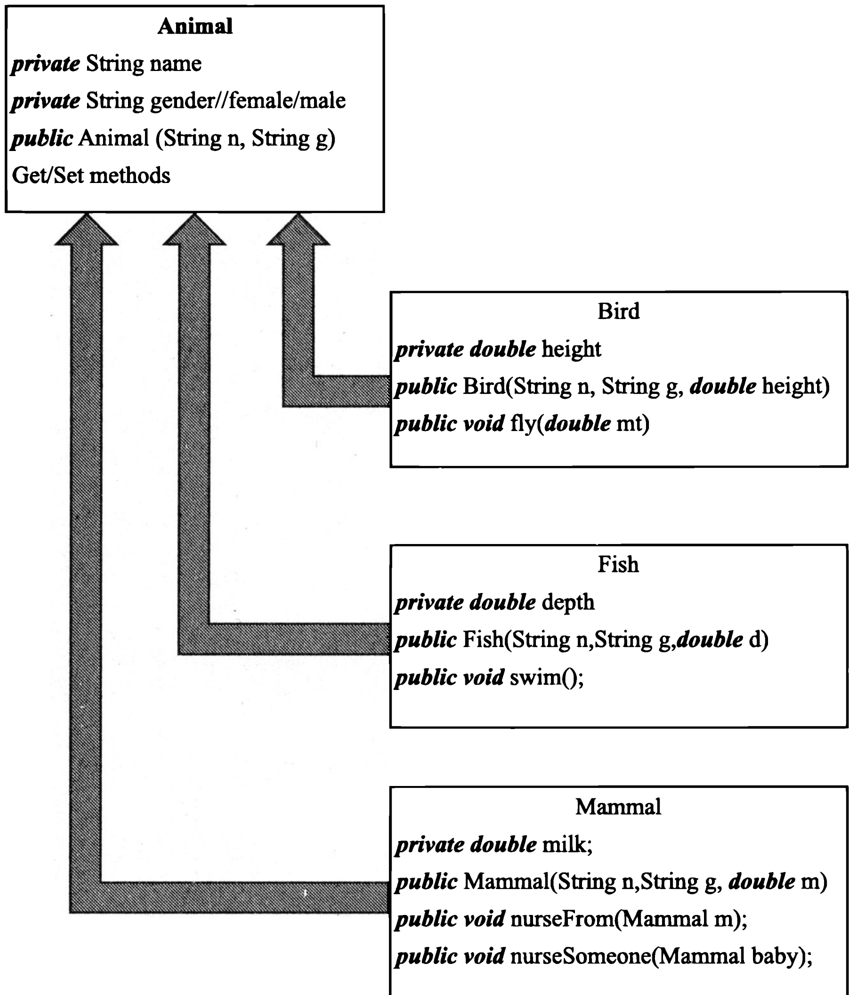
А теперь — об обещанном инструменте языка Java, который позволяет определить, будет ли на этапе исполнения происходить сбой или нет при выполнении «преобразования вниз».

Выглядит конструкция проверки следующим образом:

Алгоритмическая форма	**Запись на языке Java**
Если экземпляр класса А можно «преобразовать вниз» к экземпляру класса В	if (A instanceof B)

В нашем тестовом классе est_2 мы можем применить такую проверку к массиву carriage:

```
for (int i=0;i<carriage.length;i++)
      if ((carriage[i]) instanceof Boat)
                  {
                   Boat temp=(Boat)carriage[i];
                   System.out.println((temp.toString()));
                  }
       else System.out.println("Down casting for carriage["+i+"] not impossible");
```



Абстрактные классы

В языке Java существует возможность создать так называемые *абстрактные* классы.

Большинство словарей русского языка определяют термин «абстрактный» как «отвлеченный, общий, не конкретный, не имеющий реального воплощения». Надо сказать, что это чрезвычайно точно характеризует абстрактные классы в языке Java: в них присутствуют заголовки (сигнатуры) методов, но

отсутствуют описания (тела) этих методов. Из этого следует и еще одна чрезвычайно важная особенность абстрактных классов: невозможность создать экземпляр (объект) такого класса.

При этом, как правило, абстрактный класс является суперклассом, а указанные в нем методы реализуют его субклассы каждый по-своему. Например, суперкласс Animal содержит заголовки методов move и talk, которые означают, что животное может двигаться и говорить. При этом абстрактное животное не может издавать звуков и ходить, — эти свойства в конкретном виде реализуют Cat (кошка) и Horse (лошадь), которые издают разные звуки и по-разному двигаются.

Вот как это реализуется в языке Java.

```
//Абстрактный класс "животное" — суперкласс
public abstract class Animal {
   abstract void move();
   abstract void talk();
}

//Субкласс "кошка"
public class Cat extends Animal {
  protected int progress;
  public Cat() {
 }
 public void move() {
  System.out.println("Stolen");
  this.progress++;
 }
 public void talk() {
  System.out.println("Myau-myau");
 }
}

//Субкласс "лошадь"
public class Horse extends Animal {
  public Horse() {
 }
 public void move() {
  System.out.println("Rides");
 }
 public void talk() {
  System.out.println("Ygo-go");
 }
}
```

Как видите, мы сделали следующие действия:

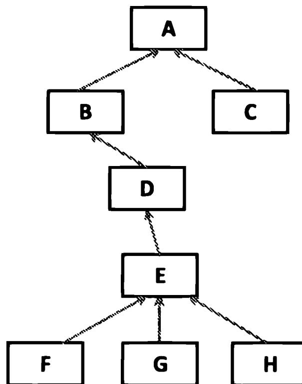
- Объявили класс Animal абстрактным (abstract), и описали в нем два абстрактных же метода. Поскольку методы описаны как абстрактные, в классе присутствуют только их заголовки без конкретного (абстрактные методы!) описания того, как эти методы исполняются.
- Объявили классы Cat и Horse субклассами метода Animal, — использовали в заголовках этих классов указание extends Animal. Это, кстати, автоматически превратило абстрактный класс Animal в суперкласс для этих двух классов.
- В каждом из субклассов описали в конкретном виде те абстрактные методы (move и talk), которые были объявлены в суперклассе Animal. Обратите внимание, что реализации отличаются (пусть и незначительно — по-разному и говорят, и ходят), притом, что каждая реализация «исходит» из одного и того же абстрактного метода.

Интерфейсы

В языке Java наследование классов — линейное: это означает, что у каждого класса может быть один и только один суперкласс. Иногда это формулируют так: в языке Java нет множественного наследования, то есть данный класс может быть напрямую наследником только одного класса.

Например, на приведенной ниже схеме наследования классов (это, кстати, явно перегруженная иерархия, но для примера она подходит), класс H является субклассом для класса E, который является субклассом для класса D, который является субклассом для класса B, который является субклассом для класса A. при это вся эта длинная цепочка расположена на одной линии!

В принципе, можно сказать, что класс H является отдаленным субклассом для класса A, но у каждого субкласса есть только и только один прямой суперкласс.



Однако многие задачи требуют того, что можно было бы назвать множественным наследованием: например, чтобы класс наследовал хотя бы двум (или более того) классам, которые не расположены на одной линии наследования в иерархии.

Такая возможность реализована в языке Java с помощью интерфейсов (interface), которые внешне чрезвычайно похожи на абстрактные классы.

Пример абстрактного класса	**Пример интерфейса**

public abstract class Animal {
  abstract void move();
  abstract void talk();
}
	public interface Animal {
	  void move();
	  void talk();
	}

Существует несколько правил, которые стоит знать:

- Если в заголовке интерфейса стоит public, он будет доступен для общего пользования; в ином случае к нему получат доступ только классы того пакета (package), в котором находится интерфейс. О пакетах — позднее.
- В заголовке интерфейса можно указывать abstract, но это совершенно необязательно, так как по определению каждый интерфейс является абстрактным классом.
- Интерфейс может наследовать другим интерфейсам. В этом случае в заголовке должно быть указано extends и список интерфейсов, которым данный наследует (имена следует указывать через запятую). При этом не допускается такой список интерфейсов, который образует замкнутую (циклическую) иерархию.
- Тело интерфейса состоит из указания постоянных значений (констант) и абстрактных методов. Поскольку все поля интерфейса уже по определению (то есть потому, что это — интерфейс) являются public final static, можно эти модификаторы вообще не указывать (но можно и указывать, разумеется).
- Необходимо инициализировать поля значений.
- Все методы интерфейса по определению public abstract, — и эти модификаторы тоже можно не указывать.
- При использовании (реализации) интерфейса в классе, заголовок класса должен содержать указание implemets и имя интерфейса (или нескольких интерфейсов, — имена их друг от друга в этом случае отделяются запятыми).
- Класс, использующий интерфейс, должен содержать описание (реализацию) всех методов, наследуемых от интерфейса.

Несмотря на то, что список вроде бы получился большим, описание интерфейса, в принципе, выглядит достаточно просто.

У интерфейсов, как и у классов, есть иерархия наследования, но принципиальное ее отличие от иерархии классов в том, что правило «наследовать можно только одному классу» для интерфейсов не обязательно. Это значит, что интерфейс может быть наследником нескольких интерфейсов, то есть реализовать множественность наследования.

Пакеты (package)

Пакет — это своего рода контейнер, в который можно заключить класс или несколько классов, для того чтобы они не вступали в конфликт с классами, имеющими точно такие же имена.

Как правило, в большинстве случаев, мы старались и рекомендовали давать классам уникальные имена. Но совершенно реальной является ситуация, когда, например, на компьютере оказываются несколько классов с одинаковыми именами (например, на компьютере учителя — работы, присланные ему его учениками). Вот для того чтобы такие классы-«тезки» не вступали в конфликт друг с другом, и существует в языке Java инструмент в виде пакетов (package) классов.

Когда в классе в явном виде указано, к какому пакету он относится, это ограничивает «область видимости», и язык Java задействует инструменты только в рамках данного пакета. Выглядит указание на принадлежность класса к определенному пакету достаточно просто: вначале используется служебное слово **package**, после которого указывается имя пакета. Например:

package java.awt.image;

После указания имени пакета, к которому принадлежит класс, но до указания имени класса (то есть — между строками кода, начинающимися с **package** и **class**), можно указывать список «импортов» (**import**), то есть указаний на внешние по отношению к данному пакеты, к которым данный класс из данного пакета может обращаться с целью использования каких-то инструментов (методов, например), описанных в классах, относящихся к другим пакетам.

Мы с вами уже использовали такое обращение в курсе «компьютерные науки», — когда была необходимость использовать действия ввода информации с клавиатуры:

Import java.util.*;

Заключение, которое можно считать вступлением

Вот чего точно нет в любой сфере человеческого знания, так это — достижения стадии, на которой можно сказать: «Теперь я знаю и умею всё».

Книга, с которой вы только что закончили работать, написана с целью дать базовые знания и навыки в программировании на языке Java. Подчеркнем — базовые. Это значит — перед вами еще долгая и интересная работа по совершенствованию уже полученных знаний, по овладению знаниями новыми, по повышению, повышению и снова повышению собственного уровня в сфере, о которой говорилось в этой книге.

Успеха!

Издательская группа
URSS представляет

АПОСТОЛОС ДОКСИАДИС

МАТЕМАТИКО-ХУДОЖЕСТВЕННЫЙ МИРОВОЙ БЕСТСЕЛЛЕР

Дядя Петрос и ПРОБЛЕМА ГОЛЬДБАХА

«Архимеда будут помнить, когда Эсхила уже забудут, потому что языки умирают, а математические идеи — нет. Может быть, «бессмертие» — глупое слово, но что бы оно ни значило, вероятно, лучшие шансы его достигнуть — у математика»

Годфри Харди. Апология математика

Перед читателем — удивительная книга, переведенная на все основные языки мира и ставшая мировым интеллектуальным бестселлером. Она повествует об истории одного математика, Петроса Папахристоса, оставшегося до конца верным великой идее познания истины, всю свою жизнь посвятившего поиску решения одной-единственной научной проблемы, оказавшейся не по силам великим умам прошлого.

Это не просто рассказ о юноше и его дяде, познакомившем племянника с простыми числами, красота которых была столь велика, что предопределила путь всей его жизни; это трагедия математика, напомнившая об играх разума Джона Нэша, Алана Тьюринга, Курта Гёделя и других великих подвижников науки.

Автор книги Апостолос Доксиадис — греческий писатель, яркий представитель современной интеллектуальной прозы. Благодаря его таланту повествование о странном дядюшке, искавшем решение одной из труднейших задач во всей математике, становится увлекательным приглашением в мир научных открытий и математических идей.

Книга адресована самому широкому кругу читателей — от математиков и философов, историков и методологов науки до тех, кто только подходит к выбору дела своей жизни; она напомнит им, что математика признает только величайших, но хорошо помнит каждого, кто был ей предан.

Издательская группа

URSS представляет

МАРТИН ГАРДНЕР

ЗАГАДКИ СФИНКСА

и другие математические головоломки

Перед читателем — впервые переведенная на русский язык работа выдающегося американского математика и популяризатора науки Мартина Гарднера. Автор как всегда остается верен своему уникальному стилю, который характеризуют яркость, доходчивость, тонкий юмор, блеск мысли, постоянное вовлечения читателя в самостоятельное творчество.

В книге представлены занимательные математические задачи и головоломки, публиковавшиеся автором в течение ряда лет на страницах «Журнала научной фантастики Айзека Азимова». Эти задачи, составленные самим М. Гарднером, увлеченными читателями его журнальной колонки, друзьями и коллегами автора, в равной мере относятся как к миру собственно математики, так и к миру логических парадоксов, многие из которых изложены в виде фантастических историй на загадочных далеких планетах.

Материал книги построен таким образом, что к каждой из головоломок дается ответ, который в большинстве случаев порождает новые вопросы, связанные с соответствующей темой. Решение этих новых вопросов приводится уже на следующем уровне ответов.

Всего в книге четыре раздела с ответами и решениями, так что читатель может последовательно переходить с одного раздела на другой, постепенно углубляя свое понимание.

Издательская группа

URSS представляет

Д. М. ЗЛАТОПОЛЬСКИЙ

СИСТЕМЫ СЧИСЛЕНИЯ

УЧЕБНЫЕ И ЗАНИМАТЕЛЬНЫЕ МАТЕРИАЛЫ

- **Более 100 содержательных задач**
- **Фокусы, головоломки, исторические факты**
- **Решение задач из ЕГЭ по информатике**
- **Вопросы для конкурсов «Что? Где? Когда?» и «Брейн-ринг»**

В книге приведены задачи, фокусы, головоломки и другие увлекательнейшие материалы, связанные с недесятичными системами счисления. Ее материалы можно использовать на уроках, в качестве домашних заданий, на кружках и факультативах, во внеклассной работе.

Книга состоит из 18 глав и содержит 13 приложений. В ней приводятся: задачи разного уровня сложности; методика решения типовых задач на системы счисления, представленных в Едином государственном экзамене по информатике; арифметические и геометрические прогрессии чисел в недесятичных системах; логические и сдвиговые операции; основные принципы создания так называемых «помехоустойчивых» кодов; математические фокусы, головоломки, игры с числами в недесятичных системах счисления.

Все задания, представленные в книге, имеют развивающее значение для интеллекта, формируют общеучебные навыки и способствуют повышению интереса учащихся к математике и информатике. Ко всем заданиям даны ответы и разъяснения.

В приложениях описываются различные методы (в том числе малоизвестные) перевода из одной системы счисления в другую целых чисел и правильных дробей, программы (с методикой их разработки) решения задач, связанных с системами счисления, решением головоломок и демонстрацией фокусов, рассмотренных ранее, а также представлены материалы исторического характера (такие материалы имеются и в основной части книги в виде «врезок»).

Евгений Гогаевич КАНЕЛЬ

Родился в 1957 году в Ашхабаде. В 1979 г. окончил математический факультет Туркменского государственного университета им. А. М. Горького. Работал в Институте сейсмологии Академии наук Туркменской ССР. Кандидат физико-математических наук. Репатриировался в Израиль в 1992 г. С 1995 г. работает в системе просвещения. Руководитель отдела преподавания компьютерных наук Образовательного центра «АМИТ», инструктор Министерства просвещения по внедрению информационных технологий в образовании, ответственный за проведение и проверку государственных экзаменов по компьютерным наукам Министерства просвещения Израиля, ведущий лектор курсов повышения квалификации учителей, преподаватель Открытого университета Израиля. В 2012–2014 гг., вместе с коллегой по работе З. Фрайманом, проводил курсы для учителей информатики Пензенской области, разработал учебные курсы «Компьютерные науки. Программирование на языке Java» и «Разработка приложений для ОС Андроид» для школ области.

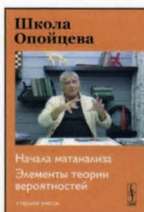
Автор более 30 научных публикаций в области сейсмологии, баз данных, методики преподавания информатики и программирования. Автор и соавтор учебников для средней школы «Введение в компьютерные науки», «Структуры данных» и «Объектно-ориентированное программирование» (на иврите).

Зэев ФРАЙМАН

Родился в 1954 году в Пензе. Окончил физико-математический факультет Пензенского государственного педагогического института им. Белинского (теперь — Пензенский государственный университет), преподавал физику и астрономию в школах города. С 1985 по 1989 гг. преподавал информатику в школе № 1 г. Пензы. В 1989 г. репатриировался в Израиль, с 1990 г. преподает дисциплины «Компьютерные науки» и «Инженерное программирование», проводит курсы повышения квалификации для учителей. В 2012–2014 гг., вместе с коллегой по работе Е. Канелем, проводил курсы для учителей информатики Пензенской области, разработал учебные курсы «Компьютерные науки. Программирование на языке Java» и «Разработка приложений для ОС Андроид» для школ области. Совместно с учителем информатики с. Ленино Пензенской области И. Б. Ермолаевым организует работу российско-израильского онлайн-проекта «Пензенская региональная заочная школа программирования». Награжден медалью «За вклад в развитие образования Пензенской области».

Автор двухтомного учебника «Сделаем и узнаем: практический курс по разработке аппликаций для ОС Андроид» для 10–12 классов израильской школы (на иврите).

Наше издательство предлагает следующие книги:



Издательская группа URSS

117335, Москва, Нахимовский проспект, 56

Телефон / факс (многоканальный) +7 (499) 724 25 45

Каталог изданий в Интернете: http://URSS.ru
E-mail: URSS@URSS.ru

Отзывы о настоящем издании, а также обнаруженные опечатки присылайте по адресу URSS@URSS.ru. Ваши замечания и предложения будут учтены и отражены на web-странице этой книги на сайте http://URSS.ru