ГОСУДАРСТВЕННЫЙ КОМИТЕТ СВЯЗИ, ИНФОРМАТИЗАЦИИ И ТЕЛЕКОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ
РЕСПУБЛИКИ УЗБЕКИСТАН

САМАРКАНДСКИЙ ФИЛИАЛ ТАШКЕНТСКОГО УНИВЕРСИТЕТА ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

ФАКУЛЬТЕТ КОМПЬЮТЕР ИНЖИНИРИНГ

КАФЕДРА КОМПЬЮТЕРНЫЕ СИСТЕМЫ

ВЫПУСКНАЯ

КВАЛИФИКАЦИОННАЯ РАБОТА

для получения академической степени бакалавриата по направлению 5521900 - “Информатика и информационные технологии ”

Тема: “Генерация всех n-кортежей”

Работа рекомендовано к защите

заседанием кафедры № __

от ___ мая 2014 года

И.О.Заведующего кафедрой

_____доц. Кубаев С.Т.

“_____” _____ 2014 г.

Выполнил: студент 4-курса

_____Шокиров Ш.

Научный руководитель:

_____доц. Абдукаримов А.

САМАРКАНД – 2014

СОДЕРЖАНИЕ

Введение

Актуальность темы: Указом Президента Республики Узбекистан от 30 мая 2002 г. N УП-3080 "О дальнейшем развитии компьютеризации и внедрении информационно-коммуникационных технологий" министерствами и ведомствами, общественными объединениями Узбекистана проводится целенаправленная работа по развитию информационно-коммуникационных технологий и услуг в сфере образования, духовно-нравственного воспитания молодежи. В связи с чем, задача подготовки высокопрофессиональных кадров, способных развивать новые информационные технологии и эффективно использовать их на практике становится стратегически важной для прогресса нашего общества.

Научный поиск новых подходов обеспечивающих требуемый уровень профессиональной компетентности будущих специалистов, становится, сегодня особенно актуальным. Немаловажная роль в решении этой проблемы принадлежит разработке и использованию инновационных технологий обучения будущего специалиста, обеспечивающих формирование компетенций студентов в определенной профессиональной сфере.

Основные задачи исследования. «Трамвайные билеты», «Монеты» при помощи этих примеров раскроем суть нашей программы;

1. Трамвайные билеты имеют номера из шести цифр от 000000 до 999999. Билет называется "счастливым", если сумма 1-й, 3-й и 5-й цифр равна сумме 2-й, 4-й и 6-й цифр. Вычислить и напечатать количество "счастливых" билетов.
2. Даны положительные целые числа A_1, A_2, A_3, A_4, M. Найти и напечатать все четверки целых положительных чисел X_1, X_2, X_3, X_4, удовлетворяющих уравнению:

$$A_1X_1 + A_2X_2 + A_3X_3 + A_4X_4 = M.$$

Цель работы. Разработка средств проектирования: методики, моделей, методов и алгоритмов для специализированных устройств генерации полных комбинаторных перестановок элементов. Наша цель состоит в изучении

методов обработки всех возможных объектов в некой комбинаторной вселенной, поскольку часто приходится сталкиваться с проблемами, при решении которых необходимо или желательно тщательно изучить все возможные случаи. Например, часто требуется изучить все перестановки заданного множества.

Методы исследования. В качестве теоретической и методологической основы исследования использованы элементы теории автоматизированного проектирования, множеств, дискретной математики, методы и средства структурно-функционального моделирования, технологии объектно-ориентированного и событийно-ориентированного программирования, а также мета - программирования.

В выпускной квалификационной работе рассмотрены общие подходы к реализации микропроцессорной системы на базе микроконтроллера для терморегулятора помещений. В процессе написания квалификационной работы автором велась разработка реализации микропроцессорной системы на базе микроконтроллера PIC16F84A.

Данная выпускная квалификационная работа состоит из 77 страниц, включающих в себя введение, три глав, заключение и приложения 1-3, включающего в себя SQL-скрипт для генерации базы данных, исходные тексты коммуникационного сервиса являющегося основой выполненной выпускной квалификационной работы.

Глава 1. Основные понятия теории генерации кортежей и Java технологии. В этой главе объясняются основные понятия и термины, используемые в данной дипломной работе. Что собой вообще представляет теория кортежей, их назначение, область применения и их архитектура.

Глава 2. Создание программного обеспечения для генерации объектов любого рода. Здесь рассмотрены некоторые из веб-служб, предоставляемые концепцией облачных вычислений. Инфраструктура является услугой в концепции облачных вычислений. «Инфраструктура как Сервис» (Infrastructure-as-a-Service, IaaS) в основном предоставляется по запросу на

базе современных вычислительных технологий и высокоскоростных сетей. «Коммуникаций как Сервис» (Communication-as-a-Service, CaaS). «Программное обеспечение как Сервис» (Software-as-a-Service, SaaS), такие как Amazon.com с их эластичной платформой облака, характеристики, преимущества, и архитектурный уровень обслуживания. Исследуем ключевые особенности использования внешних источников/ресурсов (outsourcing), доступные как "Платформы как Сервис" (Platforms-as-a-Service, PaaS).

Глава 3. Описание программного обеспечения и порядок его использования. Эта часть посвящена разработке программного обеспечения облачного вычисления, выбору, описанию и расчета элементной базы. Приводится порядок использования программного обеспечения.

В приложениях дано короткое описание выпускной работы и листинг.

ГЛАВА 1. ОСНОВНЫЕ ПОНЯТИЯ ТЕОРИИ ГЕНЕРАЦИИ КОРТЕЖЕЙ И JAVA ТЕХНОЛОГИИ.

Кортеж. В математике кортеж или n-ка (упорядоченная n-ка) — упорядоченный конечный набор длины n(где n-любое натуральное число 0), каждый из элементов которого x_n принадлежит некоторому множеству X_i, $1 \leq i \leq n$.Элементы картежа могут повторяться в нём любое число раз (этим, в частности, он отличается от упорядоченного множества, куда каждый элемент может входить только в одном экземпляре).

Множество — тип и структура данных в информатике, является реализацией математического объекта множество. Данные типа множество позволяют хранить ограниченное число значений определённого типа без определённого порядка. Повторение значений, как правило, недопустимо. За исключением того, что множество в программировании, конечно, оно в общем соответствует концепции математического множества. Для этого типа в языках программирования обычно предусмотрены стандартные операции над множествами.

Понятие **множества** - простейшее математическое понятие, оно не определяется, а лишь поясняется при помощи примеров: множество книг на полке, множество точек на прямой(точечное множество), множество языков программирования и т.д.

Взаимно однозначные соответствия, т. е. при котором каждому элементу одного множества отвечает один и только один элемент другого, и наоборот. Ясно, что взаимно однозначное соответствие между двумя конечными множествами можно установить тогда и только тогда, когда число элементов в них одинаково. Например, чтобы проверить, одинаково ли число студентов в группе и стульев в аудитории, можно, не пересчитывая ни тех, ни других, посадить каждого студента на определенный стул. Если мест хватит всем и не останется ни одного лишнего стула, т. е. если будет

установлена биекция(ВОС) между этими двумя множествами, то это и будет означать, что число элементов в них одинаково.

Счетные множества. Простейшим среди бесконечных множеств является множество натуральных чисел. Назовем *счетным множеством* всякое множество, элементы которого можно биективно сопоставить со всеми натуральными числами. Иначе говоря, счетное множество — это такое множество, элементы которого можно занумеровать в бесконечную последовательность. Приведем примеры счетных множеств:

Множество всех целых чисел. Установим соответствие между всеми целыми и всеми натуральными числами по следующей схеме:

0-1 1 -22...,

1 2 3 4 5 **...,**

Конечными называются — множество, количество элементов которого конечно, то есть, существует неотрицательное целое число k, равное количеству элементов этого множества.

Бесконечными называются — множество, не являющееся конечным. Примеры: множества натуральных чисел N, целых чисел Z, рациональных чисел Q, действительных чисел R, комплексных чисел C — являются бесконечными множествами.

Рассматривая различные множества, мы замечаем, что иногда можно, если не фактически, то хотя бы примерно, указать число элементов в данном множестве. Таковы, например, множество всех вершин некоторого многогранника, множество всех простых чисел, не превосходящих данного числа, множество всех молекул воды на Земле и т. д. Каждое из этих множеств содержит конечное, хотя, быть может, и неизвестное нам число элементов. С другой стороны, существуют множества, состоящие из бесконечного числа элементов. Два конечных множества мы можем сравнивать по числу элементов и судить, одинаково это число или же в одном из множеств элементов больше, чем в другом. Спрашивается, можно ли подобным же образом сравнивать бесконечные множества? Иначе говоря,

имеет ли смысл, например, вопрос о том, чего больше: кругов на плоскости или рациональных точек на прямой, функций, определенных на отрезке [0, 1], или прямых в пространстве, и т. д.?

Посмотрим, как мы сравниваем между собой два конечных множества. Можно, например, сосчитать число элементов в каждом из них и, таким образом, эти два множества сравнить. Но можно поступить и иначе, именно, попытаться установить ***биекцию***, т. е. взаимно однозначное соответствие между элементами этих множеств, иначе говоря, такое соответствие, при котором каждому элементу одного множества отвечает один и только один элемент другого, и наоборот. Ясно, что взаимно однозначное соответствие между двумя конечными множествами можно установить тогда и только тогда, когда число элементов в них одинаково.

Заметим теперь, что если первый способ (подсчет числа элементов) годится лишь для сравнения конечных множеств, второй (установление взаимно однозначного соответствия) пригоден и для бесконечных.

Объектно-ориентированное программирование. Объе́ктно-ориенти́рованное, или объектное, программирование (в дальнейшем ООП) — парадигма программирования, в которой основными концепциями являются понятия объектов и классов.

Парадигма программирования — это система идей и понятий, определяющих стиль написания компьютерных программ, а также образ мышления программиста. Это способ концептуализации, определяющий организацию вычислений и структурирование работы, выполняемой компьютером. Важно отметить, что парадигма программирования не определяется однозначно языком программирования; практически все современные языки программирования в той или иной мере допускают использование различных парадигм (мультипарадигмальное программирование). Так на языке Си, который не является объектно-ориентированным, можно работать в соответствии с принципами объектно-ориентированного программирования, хотя это и сопряжено с

определёнными сложностями; функциональное программирование можно применять при работе на любом императивном языке, в котором имеются функции (для этого достаточно не применять присваивание)

Система Java создавалась объектно-ориентированной с самого начала. Объектно-ориентированная парадигма наиболее удобна при создании программного обеспечения типа клиент-сервер, а также для организации распределенных вычислений.

Одна из черт, присущих объектам, заключается в том, что объекты обычно переживают процедуру, их создающую. Они затем могут перемещаться по сети, храниться в базах данных и т.д. Идейными наследниками Java являются такие языки, как C++, Eiffel, Smalltalk и Objective C. За исключением примитивных типов данных, практически все в языке является объектом.

Определение класса. Класс в программировании — набор методов и функций. Наряду с понятием «объекта» класс является ключевым понятием в ООП (хотя существуют и бесклассовые объектно-ориентированные языки, например, JavaScript. Суть отличия классов от других абстрактных типов данных состоит в том, что при задании типа данных класс определяет одновременно и интерфейс, и реализацию для всех своих экземпляров, а вызов метода-конструктора обязателен.

На практике объектно-ориентированное программирование сводится к созданию некоторого набора классов, включая интерфейс и реализацию, и последующему их использованию. Объектно-ориентированный подход за время своего развития накопил множество рекомендаций (паттернов) по созданию классов.

Класс — это шаблон для создания объекта. Класс определяет структуру объекта и его методы, образующие функциональный интерфейс. В процессе выполнения Java-программы система использует определения классов для создания представителей классов. Представители являются реальными

объектами. Термины “представитель”, “экземпляр” и “объект” взаимозаменяемы.

Имя исходного файла Java должно соответствовать имени хранящегося в нем класса. Регистр букв важен и в имени класса, и в имени файла.

Базовым элементом объектно-ориентированного программирования в языке Java является класс. Напомню, что классы в Java не обязательно должны содержать метод main. Единственное назначение этого метода — указать интерпретатору Java, откуда надо начинать выполнение программы. Для того, чтобы создать класс, достаточно иметь исходный файл, в котором будет присутствовать ключевое слово class, и вслед за ним — допустимый идентификатор и пара фигурных скобок для его тела.

Па́ттерн (англ. *'pattern* — образец, шаблон, система) — заимствованное слово. Слово «pattern» используется как термин в нескольких западных дисциплинах и технологиях, откуда оно и проникло в русскоязычную среду. Смысл термина «паттерн» больше уже чем просто «образец», и варьируется в зависимости от области знаний, в которой используется.

Паттерн — эффективный способ решения характерных задач проектирования, в частности проектирования компьютерных программ.

Объект в программировании — некоторая сущность в виртуальном пространстве, обладающая определённым состоянием и поведением, имеющая заданные значения свойств (атрибутов) и операций над ними (методов). Как правило, при рассмотрении объектов выделяется то, что объекты принадлежат одному или нескольким классам, которые определяют поведение (являются моделью) объекта. Термины «**экземпляр класса**» и «объект» взаимозаменяемы.

Объект, наряду с понятием *класс*, является важным понятием объектно-ориентированного подхода. Объекты обладают свойствам и наследования, инкапсуляции и полиморфизма.

Время жизни объекта — время с момента создания объекта (конструкция) до его уничтожения (деструкция).

Экземпляр класса (англ. *instance*) — это описание конкретного объекта в памяти. Класс описывает свойства и методы, которые будут доступны у объекта, построенного по описанию, заложенному в классе. Экземпляры используют для представления (моделирования) конкретных сущностей реального мира.

Interface. Интерфе́йс (от лат. *inter* — «между», и *face* — «поверхность») — семантическая и синтаксическая конструкция в коде программы, используемая для специфицирования услуг, предоставляемых классом или компонентом. Интерфейс определяет границу взаимодействия между классами или компонентами, специфицируя определенную абстракцию, которую осуществляет реализующая сторона. В отличие от многих других видов интерфейсов, интерфейс в ООП является строго формализованным элементом объектно-ориентированного языка и в качестве семантической конструкции широко используется кодом программы.

Понятие интерфейса - одна из главных особенностей в Java. Интерфейс позволяет классу обойти наследование от одного предка. Программы Java одновременно могут унаследовать один класс, но могут осуществить несколько интерфейсов. Интерфейс не может иметь никаких конкретных методов. Интерфейсы также используются, чтобы определить набор констант, которые могут использоваться классами. Короче говоря, интерфейс - шаблон поведения (в форме методов), который должны осуществить другие классы. Это означает, что мы имеем пустое тело метода. Интерфейс очень похож на определение класса.

Приложения Java обычно компилируются в специальный байт-код, поэтому они могут работать на любой виртуальной Java-машине (JVM) вне зависимости от компьютерной архитектуры.

Интерфейс — это контракт, который обязуется выполнить класс, *реализующий* его, с другой стороны, интерфейс — это тип данных, потому

что его описание достаточно четко определяет свойства объектов, чтобы наравне с классом типизировать переменные.

Объявление интерфейсов очень похоже на упрощенное объявление классов.

Оно начинается с заголовка. Сначала указываются модификаторы. Инерфейс может быть объявлен как public и тогда он будет доступен для общего использования, либо модификатор доступа может не указываться, в этом случае интерфейс доступен только для типов своего пакета. Модификатор abstract для интерфейса не требуется, поскольку все интерфейсы являются абстрактными классами. Его можно указать, но делать этого не рекомендуется, чтобы не загромождать код.

```
public interface Drawable extends Colorable, Resizable {
}
```

Представленный в релизе JDK 1.2 языка Java интерфейс java.util.Iterator обеспечивает итерацию контейнерных классов. Каждый Iterator реализует методыnext() и hasNext() и дополнительно может поддерживать метод remove(). Итераторы создаются соответствующими контейнерными классами, как правило методом iterator().

Метод **next()** переводит итератор на следующее значение и возвращает указываемое значение итератору. При первоначальном создании итератор указывает на специальное значение, находящееся перед первым элементом, поэтому первый элемент можно получить только после первого вызова **next().** Для определения момента, когда все элементы в контейнере были перебраны, используется тестовый метод **hasNext().** Следующий пример демонстрирует простое использование итераторов:

```
Iterator iter = list.iterator();
//Iterator<MyType> iter = list.iterator();
while (iter.hasNext())
   System.out.println(iter.next());
```

1.2. Java– технология программирования, синтаксис языка.

Все на языке программирования Java.

Java - это "объектно-ориентированный, распределенный, интерпретируемый, устойчивый, безопасный, архитектурно-нейтральный, переносимый, высокопроизводительный, многопоточный и динамический. Одно из главных преимуществ языка Java - его независимость от платформы, на которой выполняются программы. Это способствует тому, что один и тот же код можно запускать под управлением операционных систем Windows, Linux, FreeBSD, Solaris, Apple Mac и др. Это становится очень важным, когда программы загружаются посредством глобальной сети интернет и используются на различных платформах.

Другим, не менее важным преимуществом Java, является большая схожесть с языком программирования C++. Поэтому тем программистам, которые знакомы с синтаксисом C и C++ будет просто освоить Java. Кроме того, Java - полностью объектно-ориентированный язык, даже в большей степени, чем C++. Все сущности в языке Java являются объектами, за исключением немногих основных типов (primitive types), например чисел. В свое время объектно-ориентированное программирование (ООП) заменило структурное программирование.

Java-платформа обладает следующими преимуществами:

• переносимость, или кросс-платформенность;

• объектная ориентированность, создана эффективная объектная модель;

• привычный синтаксис C/C++;

• встроенная и прозрачная модель безопасности;

• ориентация на Internet-задачи, сетевые распределенные приложения;

• динамичность, легкость развития и добавления новых возможностей;

• простота освоения.

Важно и то, что разрабатывать на Java программы, которые не содержат ошибок, значительно легче, чем на C++.

В отличие от C++, Java не позволяет наследовать больше одного класса. В качестве альтернативы множественному наследованию, существуют интерфейсы. Каждый класс в Java может реализовать любой набор интерфейсов. Порождать объекты от интерфейсов в Java нельзя.

Исходный файл на языке Java - это текстовый файл, содержащий в себе одно или несколько описаний классов. Транслятор Java предполагает, что исходный текст программ хранится в файлах с расширениями Java. Получаемый в процессе трансляции код для каждого класса записывается в отдельном выходном файле, с именем совпадающем с именем класса, и расширением class.

Язык Java требует, чтобы весь программный код был заключен внутри поименованных классов.

Лексические основы

Теперь, когда мы подробно рассмотрели минимальный Java-класс, давайте вернемся назад и рассмотрим общие аспекты синтаксиса этого языка. Программы на Java — это набор пробелов, комментариев, ключевых слов, идентификаторов, литеральных констант, операторов и разделителей.

Пробелы

Java — язык, который допускает произвольное форматирование текста программ. Для того, чтобы программа работала нормально, нет никакой необходимости выравнивать ее текст специальным образом. Например, класс HelloWorld можно было записать в двух строках или любым другим способом, который придется вам по душе. И он будет работать точно так же при условии, что между отдельными лексемами (между которыми нет операторов или разделителей) имеется по крайней мере по одному пробелу, символу табуляции или символу перевода строки.

Комментарии

Хотя комментарии никак не влияют на исполняемый код программы, при правильном использовании они оказываются весьма

существенной частью исходного текста. Существует три разновидности комментариев: комментарии в одной строке, комментарии в нескольких строках и, наконец, комментарии для документирования. Комментарии, занимающие одну строку, начинаются с символов // и заканчиваются в конце строки. Такой стиль комментирования полезен для размещения кратких пояснений к отдельным строкам кода:

```
a = 42; // если 42 - ответ, то каков же был вопрос?
```

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

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

Зарезервированные ключевые слова

Зарезервированные ключевые слова — это специальные идентификаторы, которые в языке Java используются для того, чтобы идентифицировать встроенные типы, модификаторы и средства управления выполнением программы. На сегодняшний день в языке Java имеется 59 зарезервированных слов (см. таблицу 2). Эти ключевые слова совместно с синтаксисом операторов и разделителей входят в описание языка Java. Они

могут применяться только по назначению, их нельзя использовать в качестве идентификаторов для имен переменных, классов или методов.

оответствующие сторонам прямоугольного треугольника, а затем *с* помощью теоремы Пифагора вычисляется длина гипотенузы, в данном случае числа 5, величины гипотенузы классического прямоугольного треугольника со сторонами 3-4-5.

class Variables {

public static void main (String args []) {

double a = 3;

double b = 4;

double c;

c = Math.sqrt (a* a + b* b);

System.out.println ("c = "+ c);

} }

Типы

В этой главе вы познакомитесь со всеми основными типами языка Java и увидите, как надо объявлять переменные, присваивать им значения и использовать выражения со смешанными типами. В данной главе мы и обсудим простые типы языка Java, оставив сложные типы до главы 7.

Простые типы

Простые типы в Java не являются объектно-ориентированными, они аналогичны простым типам большинства традиционных языков программирования. В Java имеется восемь простых типов: — byte, short, int, long, char, float, double и boolean. Их можно разделить на четыре группы:

1. Целые. К ним относятся типы byte, short, int и long. Эти типы предназначены для целых чисел со знаком.
2. Типы с плавающей точкой — float и double. Они служат для представления чисел, имеющих дробную часть.
3. Символьный тип char. Этот тип предназначен для представления элементов из таблицы символов, например, букв или цифр.
4. Логический тип boolean. Это специальный тип, используемый для

представления логических величин.

В Java, в отличие от некоторых других языков, отсутствует автоматическое приведение типов. Несовпадение типов приводит не к предупреждению при трансляции, а к сообщению об ошибке. Для каждого типа строго определены наборы допустимых значений и разрешенных операций.

Целые числа

В языке Java понятие беззнаковых чисел отсутствует. Все числовые типы этого языка — знаковые. Например, если значение переменной типа byte равно в шестнадцатиричном виде 0x80, то это — число -1.

ЗАМЕЧАНИЕ

Единственная реальная причина использования беззнаковых чисел — это использование иных, по сравнению со знаковыми числами, правил манипуляций с битами при выполнении операций сдвига. Пусть, например, требуется сдвинуть вправо битовый массив mask, хранящийся в целой переменной и избежать при этом расширения знакового разряда, заполняющего старшие биты единицами. Стандартный способ выполнения этой задачи в C — ((unsigned) mask) >> 2. В Java для этой цели введен новый оператор беззнакового сдвига вправо. Приведенная выше операция записывается с его помощью в виде mask>>>2. Детально мы обсудим все операторы в следующей главе.

Отсутствие в Java беззнаковых чисел вдвое сокращает количество целых типов. В языке имеется 4 целых типа, занимающих 1, 2, 4 и 8 байтов в памяти. Для каждого типа — byte, short, int и long, есть свои естественные области применения.

byte

Тип byte — это знаковый 8-битовый тип. Его диапазон — от -128 до 127. Он лучше всего подходит для хранения произвольного потока байтов, загружаемого из сети или из файла.

byte b;

byte c = 0x55;

Если речь не идет о манипуляциях с битами, использования типа byte, как правило, следует избегать. Для нормальных целых чисел, используемых в качестве счетчиков и в арифметических выражениях, гораздо лучше подходит тип int.

short

short — это знаковый 16-битовый тип. Его диапазон — от -32768 до 32767. Это, вероятно, наиболее редко используемый в Java тип, поскольку он определен, как тип, в котором старший байт стоит первым.

short s;

short t = Ox55aa;

ЗАМЕЧАНИЕ

Случилось так, что на ЭВМ различных архитектур порядок байтов в слове различается, например, старший байт в двухбайтовом целом short может храниться первым, а может и последним. Первый случай имеет место в архитектурах SPARC и Power PC, второй — для микропроцессоров Intel x86. Переносимость программ Java требует, чтобы целые значения одинаково были представлены на ЭВМ разных архитектур.

int

Тип int служит для представления 32-битных целых чисел со знаком. Диапазон допустимых для этого типа значений — от -2147483648 до 2147483647. Чаще всего этот тип данных используется для хранения обычных целых чисел со значениями, достигающими двух миллиардов. Этот тип прекрасно подходит для использования при обработке массивов и для счетчиков. В ближайшие годы этот тип будет прекрасно соответствовать машинным словам не только 32-битовых процессоров, но и 64-битовых с поддержкой быстрой конвейеризации для выполнения 32-битного кода в режиме совместимости. Всякий раз, когда в одном выражении фигурируют переменные типов byte, short, int и целые литералы, тип всего выражения перед завершением вычислений приводится к int.

int i;

int j = 0x55aa0000;

long

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

long m;

long n = Ox55aa000055aa0000;

Не надо отождествлять *разрядность* целочисленного типа с занимаемым им количеством памяти. Исполняющий код Java может использовать для ваших переменных то количество памяти, которое сочтет нужным, лишь бы только их поведение соответствовало *поведению* типов, заданных вами. Фактически, нынешняя реализация Java из соображений эффективности хранит переменные типа byte и short в виде 32-битовых значений, поскольку этот размер соответствует машинному слову большинства современных компьютеров (СМ – 8 бит, 8086 – 16 бит, 80386/486 – 32 бит, Pentium – 64 бит).

Ниже приведена таблица разрядностей и допустимых диапазонов для различных типов целых чисел.

Имя	Разрядность	Диапазон
long	64	-9, 223, 372, 036, 854, 775, 808.. 9, 223, 372, 036, 854, 775, 807
Int	32	-2, 147, 483, 648.. 2, 147, 483, 647
Short	16	-32, 768.. 32, 767
byte	8	-128.. 127

Числа с плавающей точкой

Числа с плавающей точкой, часто называемые в других языках вещественными числами, используются при вычислениях, в которых требуется использование дробной части. В Java реализован стандартный (IEEE-754)набор типов для чисел с плавающей точкой — float и double и операторов для работы с ними. Характеристики этих типов приведены в таблице.

Имя	Разрядность	Диапазон
double	64	1. 7e-308.. 1. 7e+ 308
float	32	3. 4e-038.. 3. 4e+ 038

float

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

float f;

float f2 = 3. 14F; // обратите внимание на F, т.к. по умолчанию все литералы double

double

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

double d;

double pi = 3. 14159265358979323846;

Приведение типа

Приведение типов (type casting) — одно из неприятных свойств C++, тем не менее приведение типов сохранено и в языке Java. Иногда возникают ситуации, когда у вас есть величина какого-то определенного типа, а вам нужно ее присвоить переменной другого типа. Для некоторых типов это можно проделать и без приведения типа, в таких случаях говорят об автоматическом преобразовании типов. В Java автоматическое преобразование возможно только в том случае, когда точности представления чисел переменной-приемника достаточно для хранения исходного значения. Такое преобразование происходит, например, при занесении литеральной константы или значения переменной типа byte или short в переменную типа int. Это называется *расширением (widening)* или *повышением (promotion),* поскольку тип меньшей разрядности расширяется (повышается) до большего совместимого типа. Размера типа int всегда достаточно для хранения чисел из диапазона, допустимого для типа byte, поэтому в подобных ситуациях оператора явного приведения типа не требуется. Обратное в большинстве случаев неверно, поэтому для занесения значения типа int в переменную типа byte необходимо использовать оператор приведения типа. Эту процедуру иногда называют *сужением (narrowing),* поскольку вы явно сообщаете транслятору, что величину необходимо преобразовать, чтобы она уместилась в переменную нужного вам типа. Для приведения величины к определенному типу перед ней нужно указать этот тип, заключенный в круглые скобки. В приведенном ниже фрагменте кода демонстрируется приведение типа источника (переменной типа int) к типу приемника (переменной типа byte). Если бы при такой операции целое значение выходило за границы допустимого для типа byte диапазона, оно было бы уменьшено путем деления по модулю на допустимый для byte диапазон

(результат деления по модулю на число — это остаток от деления на это число).

int a = 100;

byte b = (byte) a;

Автоматическое преобразование типов в выражениях

Когда вы вычисляете значение выражения, точность, требуемая для хранения промежуточных результатов, зачастую должна быть выше, чем требуется для представления окончательного результата.

byte a = 40;

byte b = 50;

byte c = 100;

int d = a* b / c;

Результат промежуточного выражения (a* b) вполне может выйти за диапазон допустимых для типа bytезначений. Именно
поэтому Java автоматически повышает тип каждой части выражения до типа int, так что для промежуточного результата (a* b) хватает места.

Автоматическое преобразование типа иногда может оказаться причиной неожиданных сообщений транслятора об ошибках. Например, показанный ниже код, хотя и выглядит вполне корректным, приводит к сообщению об ошибке на фазе трансляции. В нем мы пытаемся записать значение 50* 2, которое должно прекрасно уместиться в тип byte, в байтовую переменную. Но из-за автоматического преобразования типа результата в int мы получаем сообщение об ошибке от транслятора — ведь при занесении int в byte может произойти потеря точности.

byte b = 50;

b = b* 2;

^ Incompatible type for =. Explicit cast needed to convert int to byte.

(Несовместимый тип для =. Необходимо явное преобразование int в byte)

Исправленный текст :

byte b = 50;

b = (byte) (b* 2);

что приводит к занесению в b правильного значения 100.

Если в выражении используются переменные типов byte, short и int, то во избежание переполнения тип всего выражения автоматически повышается до int. Если же в выражении тип хотя бы одной переменной —long, то и тип всего выражения тоже повышается до long. Не забывайте, что все целые литералы, в конце которых не стоит символ L (или 1), имеют тип int.

Если выражение содержит операнды типа float, то и тип всего выражения автоматически повышается dofloat. Если же хотя бы один из операндов имеет тип double, то тип всего выражения повышается до double.По умолчанию Java рассматривает все литералы с плавающей точкой, как имеющие тип double.Приведенная ниже программа показывает, как повышается тип каждой величины в выражении для достижения соответствия со вторым операндом каждого бинарного оператора.

class Promote {

public static void main (String args []) { byte b = 42;

char c = 'a';

short s = 1024;

int i = 50000;

float f = 5.67f;

double d =.1234;

double result = (f* b) + (i/ c) - (d* s);

System. out. println ((f* b)+ "+ "+ (i / c)+ " - " + (d* s));

System. out. println ("result = "+ result);

}

}

Подвыражение f* b — это число типа float, умноженное на число типа byte. Поэтому его тип автоматически повышается до float. Тип следующего подвыражения i / c (int, деленный на char)повышается

до int. Аналогично этому тип подвыражения d* s (double, умноженный на short) повышается до double. На следующем шаге вычислений мы имеем дело с тремя промежуточными результатами типовfloat, int и double. Сначала при сложении первых двух тип int повышается до float и получается результат типа float. При вычитании из него значения типа double тип результата повышается до double.Окончательный результат всего выражения — значение типа double.

Символы

Поскольку в Java для представления символов в строках используется кодировка Unicode, разрядность типаchar в этом языке — 16 бит. В нем можно хранить десятки тысяч символов интернационального набора символов Unicode. Диапазон типа char — 0..65536. Unicode — это объединение десятков кодировок символов, он включает в себя латинский, греческий, арабский алфавиты, кириллицу и многие другие наборы символов.

char c;

char c2 = Oxf132;

char c3 = ' a';

char c4 = '\n';

Хотя величины типа char и не используются, как целые числа, вы можете оперировать с ними так, как если бы они были целыми. Это дает вам возможность сложить два символа вместе, или инкрементировать значение символьной переменной. В приведенном ниже фрагменте кода мы, располагая базовым символом, прибавляем к нему целое число, чтобы получить символьное представление нужной нам цифры.

int three = 3;

char one = '1';

char four = (char) (three+ one);

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

Тип boolean

В языке Java имеется простой тип boolean, используемый для хранения логических значений. Переменные этого типа могут принимать всего два значения — true (истина) и false (ложь). Значения типа booleanвозвращаются в качестве результата всеми операторами сравнения, например (a < b) — об этом разговор пойдет в следующей главе. Кроме того, в главе 6 вы узнаете, что boolean — это тип, *требуемый* всеми условными операторами управления — такими, как if, while, do.

boolean done = false;

Завершая разговор о простых типах...

Теперь, когда мы познакомились со всеми простыми типами, включая целые и вещественные числа, символы и логические переменные, давайте попробуем собрать всю информацию вместе. В приведенном ниже примере создаются переменные каждого из простых типов и выводятся значения этих переменных.

class SimpleTypes {

public static void main(String args []) {

byte b = 0x55;

short s = 0x55ff;

int i = 1000000;

long l = 0xffffffffL;

char c = ' a' ;

float f = .25f;

double d = .00001234;

boolean bool = true;

System.out.println("byte b = " + b);

System.out.println("short s = " +s);

System.out.println("int i = " + i);

System.out.println("long l = " + l);

System.out.println("char c = " + c);

System.out.println("float f = " + f);

System.out.println("double d = " + d);

System.out.println("boolean bool = " + bool);

} }

Запустив эту программу, вы должны получить результат, показанный ниже:

C: \> java SimpleTypes

byte b = 85

short s = 22015

int i = 1000000

long l = 4294967295

char c = a

float f = 0.25

double d = 1.234e-005

boolean bool = true

Обратите внимание на то, что целые числа печатаются в десятичном представлении, хотя мы задавали значения некоторых из них в шестнадцатиричном формате. В главе 12 вы узнаете, как можно форматировать выводимые числовые значения.

Массивы

Для объявления типа массива используются квадратные скобки. В приведенной ниже строке объявляется переменная month_days, тип которой — “массив целых чисел типа int”.

int month_days [];

Для того, чтобы зарезервировать память под массив, используется специальный оператор new. В приведенной ниже строке кода с помощью

оператора new массиву month_days выделяется память для хранения двенадцати целых чисел.

```
month_days = new int [12];
```

Итак, теперь month_days — это ссылка на двенадцать целых чисел. Ниже приведен пример, в котором создается массив, элементы которого содержат число дней в месяцах года (невисокосного).

```
class Array {

public static void main (String args []) {

int month_days[];

month_days = new int[12];

month_days[0] = 31;

month_days[1] = 28;

month_days[2] = 31;

month_days[3] = 30;

month_days[4] = 31;

month_days[5] = 30;

month_days[6] = 31;

month_days[7] = 31;

month_days[8] = 30;

month_days[9] = 31;

month_days[10] = 30;

month_days[11] = 31;

System.out.println("April has " + month_days[3] + " days.");

} }
```

При запуске эта программа печатает количество дней в апреле, как это показано ниже. Нумерация элементов массива в Java начинается с нуля, так

что число дней в апреле — это month_days [3].

C: \> java Array

April has 30 days.

Имеется возможность автоматически инициализировать массивы способом, во многом напоминающим инициализацию переменных простых типов. Инициализатор массива представляет собой список разделенных запятыми выражений, заключенный в фигурные скобки. Запятые отделяют друг от друга значения элементов массива. При таком способе создания массив будет содержать ровно столько элементов, сколько требуется для хранения значений, указанных в списке инициализации.

class AutoArray {

public static void main(String args[]) {

int month_days[] = { 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31 };

System.out.println("April has " + month_days[3] + " days.");

} }

В результате работы этой программы, вы получите точно такой же результат, как и от ее более длинной предшественницы.

Java строго следит за тем, чтобы вы случайно не записали или не попытались получить значения, выйдя за границы массива. Если же вы попытаетесь использовать в качестве индексов значения, выходящие за границы массива — отрицательные числа либо числа, которые больше или равны количеству элементов в массиве, то получите сообщение об ошибке времени выполнения. В главе 10 мы подробно расскажем о том, что делать при возникновении подобных ошибок.

Многомерные массивы

На самом деле, настоящих многомерных массивов в Java не существует. Зато имеются массивы массивов, которые ведут себя подобно многомерным массивам, за исключением нескольких незначительных отличий. Приведенный ниже код создает традиционную матрицу из шестнадцати элементов типа double, каждый из которых инициализируется нулем. Внутренняя реализация этой матрицы — массив массивов double.

double matrix [][] = new double [4][4];

Следующий фрагмент кода инициализирует такое же количество памяти, но память под вторую размерность отводится вручную. Это сделано для того, чтобы наглядно показать, что матрица на самом деле представляет собой вложенные массивы.

```
double matrix [][] = new double [4][];

matrix [0] = new double[4];

matrix[1] = new double[4];

matrix[2] = new double[4], matrix[3] = { 0, 1, 2, 3 };
```

В следующем примере создается матрица размером 4 на 4 с элементами типа double, причем ее диагональные элементы (те, для которых x==y) заполняются единицами, а все остальные элементы остаются равными нулю.

```
class Matrix {

public static void main(String args[]) { double m[][];

m = new double[4][4];

m[0][0] = 1;

m[1][1] = 1;

m[2][2] = 1;

m[3][3] = 1;

System.out.println(m[0][0] +" "+ m[0][1] +" "+ m[0][2] +" "+ m[0][3]);

System.out.println(m[1][0] +" "+ m[1][1] +" "+ m[1][2] +" "+ m[1][3]);

System.out.println(m[2][0] +" "+ m[2][1] +" "+ m[2][2] +" "+ m[2][3]);

System.out.println(m[3][0] +" "+ m[3][1] +" "+ m[3][2] +" "+ m[3][3]);

}

}
```

Запустив эту программу, вы получите следующий результат:

C : \> Java Matrix

1 0 0 0

0 1 0 0

0 0 1 0

0 0 0 1

Обратите внимание — если вы хотите, чтобы значение элемента было нулевым, вам не нужно его инициализировать, это делается автоматически.

Для задания начальных значений массивов существует специальная форма инициализатора, пригодная и в многомерном случае. В программе, приведенной ниже, создается матрица, каждый элемент которой содержит произведение номера строки на номер столбца. Обратите внимание на тот факт, что внутри инициализатора массива можно использовать не только литералы, но и выражения.

```
class AutoMatrix {

public static void main(String args[]) { double m[][] = {

{ 0*0, 1*0, 2*0, 3*0 }, { 0*1, 1*1, 2*1, 3*1 }, { 0*2. 1*2, 2*2, 3*2 },

{ 0*3, 1*3. 2*3, 3*3 } }:

System.out.println(m[0][0] +" "+ m[0][1] +" "+ m[0][2] +" "+ m[0][3]);

System.out.println(m[1][0] +" "+m[1][1] +" "+ m[1][2] +" "+ m[1][3]);

System.out.println(m[2][0] +" "+m[2][1] +" "+ m[2][2] +" "+ m[2][3]);

System.out.println(m[3][0] +" "+m[3][1] +" "+ m[3][2] +" "+ m[3][3]);

} }
```

Запустив эту программу, вы получите следующий результат:

C: \> Java AutoMatrix

0 0 0 0

0 1 2 3

0 2 4 6

0 3 6 9

Операторы

Операторы в языке Java — это специальные символы, которые сообщают транслятору о том, что вы хотите выполнить операцию с некоторыми операндами. Некоторые операторы требуют одного операнда, их называют*унарными*. Одни операторы ставятся перед операндами и называются *префиксными,* другие — после, их называют*постфиксными* операторами. Большинство же операторов ставят между двумя операндами, такие операторы называются *инфиксными* бинарными операторами. Существует тернарный оператор, работающий с тремя операндами.

В Java имеется 44 встроенных оператора. Их можно разбить на 4 класса - *арифметические, битовые, операторы сравнения* и *логические.*

Арифметические операторы

Арифметические операторы используются для вычислений так же как в алгебре (см. таблицу со сводкой арифметических операторов ниже). Допустимые операнды должны иметь числовые типы. Например, использовать эти операторы для работы с логическими типами нельзя, а для работы с типом char можно, поскольку в Java тип char — это подмножество типа int.

Оператор	Результат	Оператор	Результат
+	Сложение	+ =	сложение с присваиванием
-	вычитание (также унарный минус)	-=	вычитание с присваиванием
*	Умножение	* =	умножение с присваиванием
/	Деление	/=	деление с присваиванием
%	деление по модулю	%=	деление по модулю с присваиванием
++	Инкремент	--	декремент

Четыре арифметических действия

Ниже, в качестве примера, приведена простая программа, демонстрирующая использование операторов. Обратите внимание на то, что операторы работают как с целыми литералами, так и с переменными.

```
class BasicMath  { public static void int a = 1 + 1;

int  b = a *  3;

main(String args[]) {

int c = b / 4;

int d = b - a;

int e = -d;

System.out.println("a =  " +  a);

System.out.println("b =  " +  b);

System.out.println("c =  " +  c);

System.out.println("d =  " +  d);

System.out.println("e =  " +  e);

} }
```

Исполнив эту программу, вы должны получить приведенный ниже результат:

C: \> java BasicMath

a = 2

b = 6

c = 1

d = 4

e = -4

Оператор деления по модулю

Оператор деления по модулю, или оператор mod, обозначается символом %. Этот оператор возвращает остаток от деления первого операнда на второй. В отличие от C++, функция mod в Java работает не только с целыми, но и с вещественными типами. Приведенная ниже программа иллюстрирует работу этого оператора.

```
class Modulus {

public static void main (String args []) {

int x = 42;

double y = 42.3;

System.out.println("x mod 10 = " + x % 10);

System.out.println("y mod 10 = " + y % 10);

} }
```

Выполнив эту программу, вы получите следующий результат:

C:\> Modulus

x mod 10 = 2

y mod 10 = 2.3

Арифметические операторы присваивания

Для каждого из арифметических операторов есть форма, в которой одновременно с заданной операцией выполняется присваивание. Ниже приведен пример, который иллюстрирует использование подобной

разновидности операторов.

```
class OpEquals {
public static void main(String args[]) {
int a = 1;
int b = 2;
int c = 3;
a += 5;
b *= 4;
c += a * b;
c %= 6;
System.out.println("a = " + a);
System.out.println("b = " + b);
System.out.println("c = " + c);
} }
```

А вот и результат, полученный при запуске этой программы:

C:> Java OpEquals

```
a = 6
b = 8
c = 3
```

Инкремент и декремент

В C существует 2 оператора, называемых операторами инкремента и

декремента (++ и --) и являющихся сокращенным вариантом записи для сложения или вычитания из операнда единицы. Эти операторы уникальны в том плане, что могут использоваться как в префиксной, так и в постфиксной форме. Следующий пример иллюстрирует использование операторов инкремента и декремента.

```
class IncDec {

public static void main(String args[]) {

int a = 1;

int b = 2;

int c = ++b;

int d = a++;

c++;

System.out.println("a = " + a);

System.out.println("b = " + b);

System.out.println("c = " + c);

System.out.println("d = " + d);

} }
```

Результат выполнения данной программы будет таким:

C:\ java IncDec

a = 2

b = 3

c = 4

d = 1

Целочисленные битовые операторы

Для целых числовых типов данных — long, int, short, char и byte, определен дополнительный набор операторов, с помощью которых можно проверять и модифицировать состояние отдельных битов соответствующих значений. В таблице приведена сводка таких операторов. Операторы битовой арифметики работают с каждым битом как с самостоятельной величиной.

Оператор	Результат	Оператор	Результат
~	побитовое унарное отрицание (NOT)		
&	побитовое И (AND)	&=	побитовое И (AND) с присваиванием
|	побитовое ИЛИ (OR)	|=	побитовое ИЛИ (OR) с присваиванием
^	побитовое исключающее ИЛИ (XOR)	^=	побитовое исключающее ИЛИ (XOR) с присваиванием
>>	сдвиг вправо	>> =	сдвиг вправо с присваиванием
>>>	сдвиг вправо с заполнением нулями	>>>=	сдвиг вправо с заполнением нулями с присваиванием
<<	сдвиг влево	<<=	сдвиг влево с присваиванием

Пример программы, манипулирующей с битами

В таблице, приведенной ниже, показано, как каждый из операторов битовой арифметики воздействует на возможные комбинации битов своих операндов. Приведенный после таблицы пример иллюстрирует использование этих операторов в программе на языке Java.

A	B	OR	AND	XOR	NOT A
0	0	0	0	0	1
1	0	1	0	1	0
0	1	1	0	1	1
1	1	1	1	0	0

```
class Bitlogic {

public static void main(String args []) {

String binary[] = { "OOOO", "0001", "0010", "0011", "0100", "0101",
"0110", "0111", "1000", "1001",     "1010", "1011", "1100", "1101",

"1110", "1111" };

int a = 3;   //     0+2+1  или двоичное 0011

int b = 6;   //     4+2+0  или двоичное 0110

int c = a | b;

int d = a & b;

int e = a ^ b;

int f = (~a & b) | (a & ~b);

int g = ~a & 0x0f;

System.out.println(" a = " + binary[a]);

System.out.println(" b = " + binary[b]);

System.out.println(" ab = " + binary[c]);

System.out.println(" a&b = " + binary[d]);

System.out.println(" a^b = " + binary[e]);

System.out.println("~a&b|a^~Ь = " + binary[f]);

System.out.println(" ~a = " + binary[g]);

} }
```

Ниже приведен результат, полученный при выполнении этой программы:

C: \> Java BitLogic

a = 0011

b = 0110

a | b = 0111

a & b = 0010

a ^ b = 0101

~a & b | a & ~b = 0101

~a = 1100

Сдвиги влево и вправо

Оператор << выполняет сдвиг влево всех битов своего левого операнда на число позиций, заданное правым операндом. При этом часть битов в левых разрядах выходит за границы и теряется, а соответствующие правые позиции заполняются нулями. В предыдущей главе уже говорилось об автоматическом повышении типа всего выражения до int в том случае если в выражении присутствуют операнды типа int или целых типов меньшего размера. Если же хотя бы один из операндов в выражении имеет тип long, то и тип всего выражения повышается до long.

Оператор >> означает в языке Java сдвиг вправо. Он перемещает все биты своего левого операнда вправо на число позиций, заданное правым операндом. Когда биты левого операнда выдвигаются за самую правую позицию слова, они теряются. При сдвиге вправо освобождающиеся старшие (левые) разряды сдвигаемого числа заполняются предыдущим содержимым знакового разряда. Такое поведение называют расширением знакового разряда.

В следующей программе байтовое значение преобразуется в строку, содержащую его шестнадцатиричное представление. Обратите внимание - сдвинутое значение приходится маскировать, то есть логически умножать на значение 0x0f, для того, чтобы очистить заполняемые в результате расширения знака биты и понизить значение до пределов, допустимых при индексировании массива шестнадцатиричных цифр.

class HexByte {

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

char hex[] = { '0', '1, '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c', 'd', 'e', 'f };

byte b = (byte) 0xf1;

System.out.println("b = 0x" + hex[(b >> 4) & 0x0f] +   hex[b & 0x0f]);

} }
```

Ниже приведен результат работы этой программы:

C:\> java HexByte

b = 0xf1

Беззнаковый сдвиг вправо

Часто требуется, чтобы при сдвиге вправо расширение знакового разряда не происходило, а освобождающиеся левые разряды просто заполнялись бы нулями.

```
class ByteUShift   {

static public void main(String args[]) {

char hex[] = { '0', '1', '2', '3', '4',    '5', '6', '7', '8', '9', 'a', 'b', 'c', 'd',       'e', 'f'
};

byte b = (byte) 0xf1;

byte c = (byte) (b >> 4);

byte d = (byte) (b >> 4);

byte e = (byte) ((b & 0xff) >> 4);

System.out.println(" b = 0x" + hex(b >> 4) & 0x0f] + hex[b & 0x0f]);

System.out.println(" b >> 4 =  0x" + hex[(c >> 4) & 0x0f] + hex[c & 0x0f]);
```

System.out.println(“b >>> 4 = 0x" + hex[(d >> 4) & 0x0f] + hex[d & 0x0f]);

System.out.println(“(b & 0xff) >> 4 = 0x" + hex[(e >> 4) & 0x0f] + hex[e & 0x0f]);

} }

Для этого примера переменную b можно было бы инициализировать произвольным отрицательным числом, мы использовали число с шестнадцатиричным представлением 0xf1. Переменной c присваивается результат знакового сдвига b вправо на 4 разряда. Как и ожидалось, расширение знакового разряда приводит к тому, что 0xf1 превращается в 0xff. Затем в переменную d заносится результат беззнакового сдвига b вправо на 4 разряда. Можно было бы ожидать, что в результате d содержит 0x0f, однако на деле мы снова получаем 0xff. Это — результат расширения знакового разряда, выполненного при автоматическом повышении типа переменной b до int перед операцией сдвига вправо. Наконец, в выражении для переменной e нам удается добиться желаемого результата — значения 0x0f. Для этого нам пришлось перед сдвигом вправо логически умножить значение переменной b на маску 0xff, очистив таким образом старшие разряды, заполненные при автоматическом повышении типа. Обратите внимание, что при этом уже нет необходимости использовать беззнаковый сдвиг вправо, поскольку мы знаем состояние знакового бита после операции AND.

C: \> java ByteUShift

b = 0xf1

b >> 4 = 0xff

b >>> 4 = 0xff

b & 0xff) >> 4 = 0x0f

Битовые операторы присваивания

Так же, как и в случае арифметических операторов, у всех бинарных битовых операторов есть родственная форма, позволяющая автоматически присваивать результат операции левому операнду. В следующем примере создаются несколько целых переменных, с которыми с помощью операторов, указанных выше, выполняются различные операции.

```
class OpBitEquals {

public static void main(String args[]) {

int a = 1;

int b = 2;

int c = 3;

a |= 4;

b >>= 1;

c <<= 1;

a ^= c;

System.out.println("a = " + a);

System.out.println("b = " + b);

System.out.println("c = " + c);

} }
```

Результаты исполнения программы таковы:

C:\> Java OpBitEquals

a = 3

b = 1

c = 6

Операторы отношения

Для того, чтобы можно было сравнивать два значения, в Java имеется набор операторов, описывающих отношение и равенство. Список таких операторов приведен в таблице.

Оператор	Результат
==	равно

!=	не равно
>	больше
<	меньше
>=	больше или равно
<=	меньше или равно

Значения любых типов, включая целые и вещественные числа, символы, логические значения и ссылки, можно сравнивать, используя оператор проверки на равенство == и неравенство !=. Обратите внимание — в языке Java, так же, как в C и C++ проверка на равенство обозначается последовательностью (==). Один знак (=) — это оператор присваивания.

Булевы логические операторы

Булевы логические операторы, сводка которых приведена в таблице ниже, оперируют только с операндами типа boolean. Все бинарные логические операторы воспринимают в качестве операндов два значения типа boolean и возвращают результат того же типа.

Оператор	Результат	Оператор	Результат
&	логическое И (AND)	&=	И (AND) с присваиванием
|	логическое ИЛИ (OR)	=	ИЛИ (OR) с присваиванием
^	логическое исключающее ИЛИ (XOR)	^=	исключающее ИЛИ (XOR) с присваиванием
||	оператор OR быстрой оценки выражений (short circuit OR)	==	равно
&&	оператор AND быстрой оценки выражений (short circuit AND)	!=	не равно
!	логическое унарное отрицание (NOT)	?:	тернарный оператор if-then-else

Результаты воздействия логических операторов на различные комбинации значений операндов показаны в таблице.

A	B	OR	AND	XOR	NOT A
false	false	false	false	false	true

true	false	true	false	true	false
false	true	true	false	true	true
true	true	true	true	false	false

Программа, приведенная ниже, практически полностью повторяет уже знакомый вам пример BitLogic. Только но на этот раз мы работаем с булевыми логическими значениями.

```
class BoolLogic {

public static void main(String args[]) {

boolean a = true;

boolean b = false;

boolean c = a | b;

boolean d = a & b;

boolean e = a ^ b;

boolean f = (!a & b) | (a & !b);

boolean g = !a;

System.out.println(" a = " + a);

System.out.println(" b = " + b);

System.out.println(" a|b = " + c);

System.out.println(" a&b = " + d);

System.out.println(" a^b = " + e);

System.out.println("!a&b|a&!b = " + f);

System.out.println(" !a = " + g);

} }
```

C: \> Java BoolLogic

a = true

b = false

a|b = true

a&b = false

a^b = true

!a&b|a&!b = true

!a = false

Операторы быстрой оценки логических выражений (short circuit logical operators)

Существуют два интересных дополнения к набору логических операторов. Это — альтернативные версии операторов AND и OR, служащие для быстрой оценки логических выражений. Вы знаете, что если первый операнд оператора OR имеет значение true, то независимо от значения второго операнда результатом операции будет величина true. Аналогично в случае оператора AND, если первый операнд — false, то значение второго операнда на результат не влияет — он всегда будет равен false. Если вы в используете операторы && и || вместо обычных форм & и |, то Java не производит оценку правого операнда логического выражения, если ответ ясен из значения левого операнда. Общепринятой практикой является использование операторов && и || практически во всех случаях оценки булевых логических выражений. Версии этих операторов & и | применяются только в битовой арифметике.

Тернарный оператор if-then-else

Общая форма оператора if-then-use такова:

выражение1? выражение2: выражение3

В качестве первого операнда — «выражение1» — может быть использовано любое выражение, результатом которого является значение типа boolean. Если результат равен true, то выполняется оператор, заданный вторым

операндом, то есть, «выражение2». Если же первый операнд равен false, то выполняется третий операнд — «выражение3». Второй и третий операнды, то есть «выражение2» и «выражение3», должны возвращать значения одного типа и не должны иметь тип void.

В приведенной ниже программе этот оператор используется для проверки делителя перед выполнением операции деления. В случае нулевого делителя возвращается значение 0.

```
class Ternary {

public static void main(String args[]) {

int a = 42;

int b = 2;

int c = 99;

int d = 0;

int e = (b == 0) ? 0 : (a / b);

int f = (d == 0) ? 0 : (c / d);

System.out.println("a = " + a);

System.out.println("b = " + b);

System.out.println("c = " + c);

System.out.println("d = " + d);

System.out.println("a / b = " + e);

System.out.println("c / d = " + f);

} }
```

При выполнении этой программы исключительной ситуации деления на нуль не возникает и выводятся следующие результаты:

C: \>java Ternary

a = 42

b = 2

c = 99

d = 0

a / b = 21

c / d = 0

Приоритеты операторов

В Java действует определенный порядок, или приоритет, операций. В элементарной алгебре нас учили тому, что у умножения и деления более высокий приоритет, чем у сложения и вычитания. В программировании также приходится следить и за приоритетами операций. В таблице указаны в порядке убывания приоритеты всех операций языка Java.

Высший			
( )	[ ]	.	
~	!		
*	/	%	
+	-		
>>	>>>	<<	
>	>=	<	<=
==	!=		
&			
^			
|			

&&			
||			
?:			
=	op=		
Низший			

В первой строке таблицы приведены три необычных оператора, о которых мы пока не говорили. Круглые скобки ()используются для явной установки приоритета. Как вы узнали из предыдущей главы, квадратные скобки []используются для индексирования переменной-массива. Оператор . (точка) используется для выделения элементов из ссылки на объект — об этом мы поговорим в главе 7. Все же остальные операторы уже обсуждались в этой главе.

Явные приоритеты

Поскольку высший приоритет имеют круглые скобки, вы всегда можете добавить в выражение несколько пар скобок, если у вас есть сомнения по поводу порядка вычислений или вам просто хочется сделать свои код более читабельным.

a >> b + 3

Какому из двух выражений, a >> (b + 3) или (a >> b) + 3, соответствует первая строка? Поскольку у оператора сложения более высокий приоритет, чем у оператора сдвига, правильный ответ — a>> (b + *a).* Так что если вам требуется выполнить операцию (a>>b)+ *3* без скобок не обойтись.

Условные операторы

Они хорошо Вам знакомы, давайте познакомимся с каждым из них в Java.

if-else

В обобщенной форме этот оператор записывается следующим образом:

```
if (логическое выражение) оператор1; [ else оператор2;]
```

Раздел else необязателен. На месте любого из *операторов* может стоять *составной оператор,* заключенный в фигурные скобки. *Логическое выражение* — это любое выражение, возвращающее значение типа boolean.

```
int bytesAvailable;

// ...

if (bytesAvailable > 0) {

ProcessData();

bytesAvailable -= n;

} else

waitForMoreData();
```

А вот полная программа, в которой для определения, к какому времени года относится тот или иной месяц, используются операторы if-else.

```
class IfElse {

public static void main(String args[]) { int month = 4;

String season;

if (month == 12 || month == 1 || month == 2) {

season = "Winter";

} else if (month ==3 || month == 4 || month == 5) {

season = "Spring";

} else if (month == 6 || month == 7 || month == 8) {

season = "Summer";

} else if (month == 9 || month == 10 || month == 11) {

season = "Autumn";

} else {

season = "Bogus Month";

}

System.out.println( "April is in the " + season  + ".");

} }
```

После выполнения программы вы должны получить следующий результат:

```
C: \> java IfElse

April is in the Spring.
```

break

В *языке* Java отсутствует оператор goto. Для того, чтобы в некоторых случаях заменять goto, в Java предусмотрен оператор break. Этот оператор сообщает исполняющей среде, что следует прекратить выполнение именованного блока и передать управление оператору, следующему за данным блоком. Для именования блоков в языке Java используются метки. Оператор break при работе с циклами и в операторах switch может использоваться без метки. В таком случае подразумевается выход из текущего блока.

Например, в следующей программе имеется три вложенных блока, и у каждого своя уникальная метка. Оператор break, стоящий во внутреннем блоке, вызывает переход на оператор, следующий за блоком b. При этом пропускаются два оператора println.

```
class Break {

public static void main(String args[]) { boolean t = true;

a:      { b:         { c:             {

System.out.println("Before the break"); // Перед break

        if (t)

        break b;

        System.out.println("This won't execute");  // Не будет  выполнено    }

        System.out.println("This won't execute"); // Не будет  выполнено    }

        System.out.println("This is after b"); //После b

} } }
```

В результате исполнения программы вы получите следующий результат:

```
C:\> Java Break

Before the break

This is after b
```

ВНИМАНИЕ

Вы можете использовать оператор break только для перехода за один из текущих вложенных блоков. Это отличает break от оператора goto языка C, для которого возможны переходы на произвольные метки.

switch

Оператор switch обеспечивает ясный способ переключения между различными частями программного кода в зависимости от значения одной переменной или выражения. Общая форма этого оператора такова:

```
switch ( выражение ) { case значение1:

break;

case значение2:

break;

case значением:

break;

default:

}
```

Результатом вычисления *выражения* может быть значение любого простого типа, при этом каждое из значений, указанных в операторах case, должно быть совместимо по типу с выражением в операторе switch. Все эти значения должны быть уникальными литералами. Если же вы укажете в двух операторах case одинаковые значения, транслятор выдаст сообщение об ошибке.

Если же значению выражения не соответствует ни один из операторов case, управление передается коду, расположенному после ключевого слова default. Отметим, что оператор default необязателен. В случае, когда ни один из операторов case не соответствует значению выражения и в switch отсутствует оператор default выполнение программы продолжается с оператора, следующего за оператором switch.

Внутри оператора switch (а также внутри циклических конструкций, но об этом — позже) break без метки приводит к передаче управления на код, стоящий после оператора switch. Если break отсутствует, после текущего раздела case будет выполняться следующий. Иногда бывает удобно иметь в операторе switch несколько смежных разделов case, не разделенных оператором break.

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

int month = 4;

String season;

switch (month) {

case 12: // FALLSTHROUGH

case 1: // FALLSTHROUGH

case 2:

season = "Winter";

break;

case 3: // FALLSTHROUGH
```

```
case 4: // FALLSTHROUGH

case 5:

season = "Spring";

break;

case 6: // FALLSTHROUGH

case 7: // FALLSTHROUGH

case 8:

season = "Summer";

break;

case 9: // FALLSTHROUGH

case 10: // FALLSTHROUGH

case 11:

season = "Autumn";

break;

default:

season = "Bogus Month";

}

System.out.println("April is in the " +    season +     ".");

} }
```

Ниже приведен еще более полезный пример, где оператор switch используется для передачи управления в соответствии с различными кодами символов во входной строке. Программа подсчитывает число строк, слов и символов в текстовой строке.

```
class WordCount {

static String text = "Now is the tifne\n" +

                     "for all good men\n" +

                     "to come to the aid\n" +

                     "of their country\n"+

                     "and pay their due taxes\n";

static int len = text.length();

public static void main(String args[]) {

boolean inWord = false;
```

```
int numChars = 0;

int numWords = 0;

int numLines = 0;

for (int i=0; i < len; i++) {

     char c = text.charAt(i);

     numChars++;

     switch (c) {

              case '\n': numLines++; // FALLSTHROUGH

              case '\t': // FALLSTHROUGH

              case ' ' : if (inWord) {

                                 numWords++;

                                 inWord = false;

                        }

                        break;

            default: inWord = true;

                }

     }

System.out.println("\t" + numLines +"\t" + numWords + "\t" + numChars);

} }
```

В этой программе для подсчета слов использовано несколько концепций, относящихся к обработке строк. Подробно эти вопросы будут рассмотрены в главе 9.

return

В следующей главе вы узнаете, что в Java для реализации процедурного интерфейса к объектам классов используется разновидность подпрограмм, называемых методами. Подпрограмма main, которую мы использовали до сих пор — это статический метод соответствующего класса-примера. В любом месте программного кода метода можно поставить оператор return, который приведет к немедленному завершению работы и передаче управления коду, вызвавшему этот метод. Ниже приведен пример, иллюстрирующий использование оператора return для немедленного возврата управления, в данном случае — исполняющей среде Java.

```
class ReturnDemo {

public static void main(String args[]) {

boolean t = true;
```

```
System.out.println("Before the return"); //Перед оператором return

if (t) return;

System.out.println("This won't execute"); //Это не будет выполнено

} }
```

Замечание

Зачем в этом примере использован оператор if (t)? Дело в том, не будь этого оператора, транслятор Java догадался бы, что последний оператор println никогда не будет выполнен. Такие случаи в Java считаются ошибками, поэтому без оператора if оттранслировать этот пример нам бы не удалось.

Циклы

Любой цикл можно разделить на 4 части — *инициализацию, тело, итерацию и условие завершения.* В Java есть три циклические конструкции: while (с пред-условием), do-while (с пост-условием) и for (с параметровм).

while

Этот цикл многократно выполняется до тех пор, пока значение логического выражения равно true. Ниже приведена общая форма оператора while:

```
[ инициализация; ]

while ( завершение )     {

тело;

[итерация;] }
```

Инициализация и итерация необязательны. Ниже приведен пример цикла while для печати десяти строк «tick».

```
class WhileDemo {

public static void main(String args[]) {

int n = 10;

while (n > 0) {

        System.out.println("tick " + n);

        n--;

        }

} }
```

do-while

Иногда возникает потребность выполнить тело цикла по крайней мере один раз — даже в том случае, когда логическое выражение с самого начала принимает значение false. Для таких случаев в Java используется циклическая конструкция do-while. Ее общая форма записи такова:

```
[ инициализация; ] do { тело; [итерация;] } while ( завершение );
```

В следующем примере тело цикла выполняется до первой проверки условия завершения. Это позволяет совместить код итерации с условием завершения:

```
class DoWhile {

public static void main(String args[]) {

int n = 10;

do {

    System.out.println("tick " + n);

   } while (--n > 0);

} }
```

for

В этом операторе предусмотрены места для всех четырех частей цикла. Ниже приведена общая форма оператора записи for.

```
for ( инициализация; завершение; итерация ) тело;
```

Любой цикл, записанный с помощью оператора for, можно записать в виде цикла while, и наоборот. Если начальные условия таковы, что при входе в цикл условие завершения не выполнено, то операторы тела и итерации не выполняются ни одного раза. В каноническая форме цикла for происходит увеличение целого значения счетчика с минимального значения до определенного предела.

```
class ForDemo {

public static void main(String args[]) {

for (int i = 1; i <= 10; i++)

System.out.println("i = " + i);

} }
```

Следующий пример — вариант программы, ведущей обратный отсчет.

```
class ForTick {

public static void main(String args[]) {

for (int n = 10; n > 0; n--)

System.out.println("tick " + n);

} }
```

Обратите внимание — переменные можно объявлять внутри раздела инициализации оператора for. Переменная, объявленная внутри оператора for, действует в пределах этого оператора.

А вот — новая версия примера с временами года, в которой используется оператор for.

```
class Months {

static String months[] = {

"January", "February", "March", "April", "May", "June", "July", "August", "September",
"October", "November", "December" };

static int monthdays[]   = { 31, 28,  31, 30, 31,  30, 31, 31, 30, 31, 30, 31 };

static String spring = "spring";

static String summer = "summer";

static String autumn = "autumn";

static String winter = "winter";

static String seasons[]  = { winter, winter, spring, spring, spring, summer, summer,
summer, autumn, autumn, autumn, winter };

public static void main(String args[]) {

for (int month = 0; month < 12; month++) {

System.out.println(months[month] + " is a "      +

seasons[month] + " month with " + monthdays[month] + " days.");

} } }
```

При выполнении эта программа выводит следующие строки:

```
C:\> Java Months

January is a winter month with 31 days.

February is a winter month with 28 days.

March is a spring month with 31 days.

April is a spring month with 30 days.
```

```
May is a spring month with 31 days.

June is a summer month with 30 days.

July is a summer month with 31 days.

August is a summer month with 31 days.

September is a autumn month with 30 days.

October is a autumn month with 31 days.

November is a autumn month with 30 days.

December a winter month with 31 days.
```

Оператор запятая

Иногда возникают ситуации, когда разделы инициализации или итерации цикла for требуют нескольких операторов. Поскольку составной оператор в фигурных скобках в заголовок цикла for вставлять нельзя, Java предоставляет альтернативный путь. Применение запятой (,) для разделения нескольких операторов допускается только внутри круглых скобок оператора for. Ниже приведен тривиальный пример цикла for, в котором в разделах инициализации и итерации стоит несколько операторов.

```
class Comma {

public static void main(String args[]) {

int a, b;

for (a = 1, b = 4; a < b; a++, b--) {

     System.out.println("a = " + a);

     System.out.println("b = " + b);

     }

} }
```

Вывод этой программы показывает, что цикл выполняется всего два раза.

```
C: \> java Comma

a = 1

b = 4

a = 2

b = 3
```

continue

В некоторых ситуациях возникает потребность досрочно перейти к выполнению следующей итерации, проигнорировав часть операторов тела цикла, еще не выполненных в текущей итерации. Для этой цели в Java предусмотрен оператор continue. Ниже приведен пример, в котором оператор continue используется для того, чтобы в каждой строке печатались два числа.

```
class ContinueDemo {

public static void main(String args[]) {

for (int i=0; i < 10; i++) {

     System.out.print(i + " ");

     if (i % 2 == 0) continue;

     System.out.println("");

     }

}}
```

Если индекс четный, цикл продолжается без вывода символа новой строки. Результат выполнения этой программы таков:

```
C: \> java ContinueDemo

0 1

2 3

4 5

5 7

8 9
```

Как и в случае оператора break, в операторе continue можно задавать метку, указывающую, в каком из вложенных циклов вы хотите досрочно прекратить выполнение текущей итерации. Для иллюстрации служит программа, использующая оператор continue с меткой для вывода треугольной таблицы умножения для чисел от 0 до 9:

```
class ContinueLabel {

public static void main(String args[]) {

outer:   for (int i=0; i < 10; i++) {

              for (int j = 0; j < 10; j++) {

                   if (j > i) {
```

```
                    System.out.println("");
                    continue outer;
                }
            System.out.print(" " + (i * j));
            }
        }
}}
```

Оператор continue в этой программе приводит к завершению внутреннего цикла со счетчиком j и переходу к очередной итерации внешнего цикла со счетчиком i. В процессе работы эта программа выводит следующие строки:

```
C:\> Java ContinueLabel
0
0 1
0 2 4
0 3 6 9
0 4 8 12 16
0 5 10 15 20 25
0 6 12 18 24 30 36
0 7 14 21 28 35 42 49
0 8 16 24 32 40 48 56 64
0 9 18 27 36 45 54 63 72 81
```

Исключения

Последний способ вызвать передачу управления при выполнении кода — использование встроенного в Java механизма обработки исключительных ситуаций. Для этой цели в языке предусмотрены операторы try, catch, throw и finally.

Классы

Базовым элементом объектно-ориентированного программирования в языке Java является класс. В этой главе Вы научитесь создавать и расширять свои собственные классы, работать с экземплярами этих классов и начнете использовать мощь объектно-ориентированного подхода. Напомним, что классы в Java не обязательно должны содержать метод main. Единственное назначение этого метода — указать интерпретатору Java, откуда надо начинать выполнение программы. Для того, чтобы создать класс, достаточно иметь исходный файл, в котором будет присутствовать ключевое слово class, и вслед за ним — допустимый идентификатор и пара фигурных

скобок для его тела.

```
class Point {

}
```

ЗАМЕЧАНИЕ

Имя исходного файла Java должно соответствовать имени хранящегося в нем класса. Регистр букв важен и в имени класса, и в имени файла.

Как вы помните из главы 2, *класс* — это шаблон для создания объекта. Класс определяет структуру объекта и его*методы,* образующие функциональный интерфейс. В процессе выполнения Java-программы система использует определения классов для создания представителей классов. Представители являются реальными *объектами.*Термины «представитель», «экземпляр» и «объект» взаимозаменяемы. Ниже приведена общая форма определения класса.

```
class имя_класса extends имя_суперкласса { type
переменная1_объекта:

type переменная2_объекта:

type переменнаяN_объекта:

type имяметода1(список_параметров) { тело метода;

}

type имяметода2(список_параметров) { тело метода;

}

type имя методаM(список_параметров) { тело метода;

}

}
```

Ключевое слово extends указывает на то, что *«имя_класса»* — это подкласс класса *«имя_суперкласса».* Во главе классовой иерархии Java стоит единственный ее встроенный класс — Object. Если вы хотите создать подкласс непосредственно этого класса, ключевое слово extends и следующее за ним имя суперкласса можно опустить — транслятор включит их в ваше определение автоматически. Примером может служить

класс Point, приведенный ранее.

Переменные представителей (instance variables)

Данные инкапсулируются в класс путем объявления переменных между открывающей и закрывающей фигурными скобками, выделяющими в определении класса его тело. Эти переменные объявляются точно так же, как объявлялись локальные переменные в предыдущих примерах. Единственное отличие состоит в том, что их надо объявлять вне методов, в том числе вне метода main. Ниже приведен фрагмент кода, в котором объявлен класс Point с двумя переменными типа int.

```
class Point { int x, y;

}
```

В качестве типа для переменных объектов можно использовать как любой из простых типов, описанных в главе 4, так и классовые типы. Скоро мы добавим к приведенному выше классу метод main, чтобы его можно было запустить из командной строки и создать несколько объектов.

Оператор new

Оператор new создает экземпляр указанного класса и возвращает ссылку на вновь созданный объект. Ниже приведен пример создания и присваивание переменной p экземпляра класса Point.

```
Point p = new Point();
```

Вы можете создать несколько ссылок на один и тот же объект. Приведенная ниже программа создает два различных объекта класса Point и в каждый из них заносит свои собственные значения. Оператор точка используется для доступа к переменным и методам объекта.

```
class TwoPoints {

public static void main(String args[]) {

Point p1 = new Point();

Point p2 = new Point();

p1.x = 10;

p1.y = 20;

p2.x = 42;

p2.y = 99;

System.out.println("x = " + p1.x + " y = " + p1.y);
```

```
System.out.println("x =  " + p2.x + " y = " + p2.y);

} }
```

В этом примере снова использовался класс Point, было создано два объекта этого класса, и их переменным x и y присвоены различные значения. Таким образом мы продемонстрировали, что переменные различных объектов независимы на самом деле. Ниже приведен результат, полученный при выполнении этой программы.

```
C:\> Java TwoPoints

x = 10 y = 20

x = 42 y = 99
```

ЗАМЕЧАНИЕ

Поскольку при запуске интерпретатора мы указали в командной строке не класс Point, а класс TwoPoints, метод main класса Point был полностью проигнорирован. Добавим в класс Point метод main и, тем самым, получим законченную программу.

```
class Point { int x, y;

public static void main(String args[]) {

Point p = new Point();

p.x = 10;

p.y = 20;

System.out.println("x =  " + p.x + " y = " + p.y);

} }
```

Объявление методов

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

```
тип имя_метода (список формальных параметров) {

тело метода:
```

```
}
```

Тип результата, который должен возвращать метод может быть любым, в том числе и типом void - в тех случаях, когда возвращать результат не требуется. Список формальных параметров - это последовательность пар тип-идентификатор, разделенных запятыми. Если у метода параметры отсутствуют, то после имени метода должны стоять пустые круглые скобки.

```
class Point { int x, y;

void init(int a, int b) {

x = a;

y = b;

} }
```

Вызов метода

В Java отсутствует возможность передачи параметров *по ссылке* на примитивный тип. В Java все параметры примитивных типов передаются *по значению,* а это означает, что у метода нет доступа к исходной переменной, использованной в качестве параметра. Заметим, что все объекты передаются по ссылке, можно изменять содержимое того объекта, на который ссылается данная переменная. В главе 12 Вы узнаете, как предать переменные примитивных типов по ссылке (через обрамляющие классы-оболочки).

Скрытие переменных представителей

В языке Java не допускается использование в одной или во вложенных областях видимости двух локальных переменных с одинаковыми именами. Интересно отметить, что при этом не запрещается объявлять формальные параметры методов, чьи имена совпадают с именами переменных представителей. Давайте рассмотрим в качестве примера иную версию метода init, в которой формальным параметрам даны имена x и y, а для доступа к одноименным переменным текущего объекта используется ссылка **this**.

```
class Point { int x, y;

void init(int x, int y) {

this.x = x;

this.y = y } }

class TwoPointsInit {

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

```
Point p1 = new Point();

Point p2 = new Point();

p1.init(10,20);

p2.init(42,99);

System.out.println("x = " + p1.x + " y = •• + p-1.y);

System.out.printlnC'x = " + p2.x + " y = •• + p2.y);

} }
```

Конструкторы

Инициализировать все переменные класса всякий раз, когда создается его очередной представитель — довольно утомительное дело даже в том случае, когда в классе имеются функции, подобные методу init. Для этого в Java предусмотрены специальные методы, называемые конструкторами. Конструктор — это метод класса, который инициализирует новый объект после его создания. Имя конструктора всегда *совпадает* с именем класса, в котором он расположен (также, как и в C++). У конструкторов нет типа возвращаемого результата - никакого, даже void.Заменим метод init из предыдущего примера конструктором.

```
class Point { int x, y;

Point(int x, int y) {

this.x = x;

this.y = y;

} }

class PointCreate {

public static void main(String args[]) {

Point p = new Point(10,20);

System.out.println("x = " + p.x + " y = " + p.y);

} }
```

Программисты на Pascal (Delphi) для обозначения конструктора используют ключевое

слово **constructor**.

Совмещение методов

Язык Java позволяет создавать несколько методов с одинаковыми именами, но с разными списками параметров. Такая техника называется совмещением методов (**method overloading**). В качестве примера приведена версия класса Point, в которой совмещение методов использовано для определения альтернативного конструктора, который инициализирует координаты x и y значениями по умолчанию (-1).

```
class Point { int x, y;

Point(int x, int y) {

this.x = x;

this.y = y;

}

Point() {

x = -1;

y = -1;

} }

class PointCreateAlt {

public static void main(String args[]) {

Point p = new Point();

System.out.println("x = " + p.x + " y = " + p.y);

} }
```

В этом примере объект класса Point создается не при вызове первого конструктора, как это было раньше, а с помощью второго конструктора без параметров. Вот результат работы этой программы:

```
C:\> java PointCreateAlt

x = -1 y = -1
```

ЗАМЕЧАНИЕ

Решение о том, какой конструктор нужно вызвать в том или ином случае, принимается в соответствии с количеством и типом параметров, указанных в операторе new. Недопустимо объявлять в классе методы с одинаковыми именами и сигнатурами. В сигнатуре метода не учитываются имена формальных параметров учитываются лишь их типы и количество.

this в конструкторах

Очередной вариант класса Point показывает, как, используя this и совмещение методов, можно строить одни конструкторы на основе других.

```
class Point { int x, y;

Point(int x, int y) {

this.x = x;

this.y = y;

}

Point() {

this(-1, -1);

} }
```

В этом примере второй конструктор для завершения инициализации объекта обращается к первому конструктору.

Методы, использующие совмещение имен, не обязательно должны быть конструкторами. В следующем примере в класс Point добавлены два метода distance. Функция distance возвращает расстояние между двумя точками. Одному из совмещенных методов в качестве параметров передаются координаты точки x и y, другому же эта информация передается в виде параметра-объекта Point.

```
class Point { int x, y;

Point(int x, int y) {

this.x = x;

this. y = y;

}
```

```
double distance(int x, int y) {

int dx = this.x - x;

int dy = this.y - y;

return Math.sqrt(dx*dx + dy*dy);

}

double distance(Point p) {

return distance(p.x, p.y);

} }

class PointDist {

public static void main(String args[]) {

Point p1 = new Point(0, 0);

Point p2 = new Point(30, 40);

System.out.println("p1 = " + pi.x + ", " + p1.y);

System.out.println("p2 = " + p2.x + ", " + p2.y);

System.out.println("p1.distance(p2) = " + p1.distance(p2));

System.out.println("p1.distance(60, 80) = " + p1.distance(60,
80));

} }
```

Обратите внимание на то как во второй фороме метода **distance** для получения результата вызывается его первая форма. Ниже приведен результат работы этой программы:

```
C:\> java PointDist

p1 = 0, 0

p2 = 30, 40

p1.distance(p2) = 50.0

p1.distance(60, 80) = 100.0
```

Наследование

Вторым фундаментальным свойством объектно-ориентированного подхода является наследование (первый – инкапсуляция). Классы-потомки имеют возможность не только создавать свои собственные переменные и методы, но и наследовать переменные и методы классов-предков. Классы-потомки принято называть подклассами. Непосредственного предка данного класса называют его суперклассом. В очередном примере показано, как расширить класс Point таким образом, чтобы включить в него третью координату z.

```
class Point3D extends Point { int z;

Point3D(int x, int y, int z) {

this.x = x;

this.y = y;

this.z = z; }

Point3D() {

this(-1,-1,-1);

} }
```

В этом примере ключевое слово extends используется для того, чтобы сообщить транслятору о намерении создать подкласс класса Point. Как видите, в этом классе не понадобилось объявлять переменные x и y, поскольку Point3D унаследовал их от своего суперкласса Point.

ВНИМАНИЕ

Вероятно, программисты, знакомые с C++, очевидно ожидают, что сейчас мы начнем обсуждать концепцию множественного наследования. Под множественным наследованием понимается создание класса, имеющего несколько суперклассов. Однако в языке Java ради обеспечения высокой производительности и большей ясности исходного кода множественное наследование реализовано не было. В большинстве случаев, когда требуется множественное наследование, проблему можно решить с помощью имеющегося в Java механизма интерфейсов, описанного в следующей главе.

super

В примере с классом Point3D частично повторялся код, уже имевшийся в суперклассе.

Вспомните, как во втором конструкторе мы использовали **this** для вызова первого конструктора того же класса. Аналогичным образом ключевое слово **super** позволяет обратиться непосредственно к конструктору суперкласса (в Delphi / C++ для этого используется ключевое слово **inherited**).

```
class Point3D extends Point { int z;

Point3D(int x, int y, int z) {

super(x, y);      // Здесь мы вызываем конструктор суперкласса
this.z=z;

public static void main(String       args[]) {

Point3D p = new Point3D(10, 20, 30);

System.out.println( " x = " +  p.x + " y = " + p.y +

                    " z = " + p.z);

} }
```

Вот результат работы этой программы:

```
C:\> java Point3D

x = 10 y = 20 z = 30
```

Замещение методов

Новый подкласс Point3D класса Point наследует реализацию метода distance своего суперкласса (примерPointDist.java). Проблема заключается в том, что в классе Point уже определена версия метода distance(mt x, int y), которая возвращает обычное расстояние между точками на плоскости. Мы должны *заместить* (**override**) это определение метода новым, пригодным для случая трехмерного пространства. В следующем примере проиллюстрировано и *совмещение* (overloading), и *замещение* (overriding) метода distance.

```
class Point { int x, y;

Point(int x, int y) {

this.x = x;
```

```
this.y = y;
}
double distance(int x, int y) {
int dx = this.x - x;
int dy = this.y - y:
return Math,sqrt(dx*dx + dy*dy);
}
double distance(Point p) {
return distance(p.x, p.y);
}
}
class  Point3D extends Point { int z;
Point3D(int x, int y, int z)   {
super(x, y);
this.z = z;
(
double distance(int x, int y,  int z) {
int dx = this.x - x;
int dy = this.y - y;
int dz = this.z -  z;
return Math.sqrt(dx*dx + dy*dy + dz*dz);
}
double distance(Point3D other) {
return distance(other.x, other.y, other.z);
}
double distance(int x, int y)  {
```

```
double dx = (this.x / z) - x;

double dy = (this.y / z) - y;

return Math.sqrt(dx*dx + dy*dy);

}

}

class Point3DDist {

public static void main(String args[]) {

Point3D p1 = new Point3D(30, 40, 10);

Point3D p2 = new Point3D(0, 0, 0);

Point p = new Point(4, 6);

System.out.println("p1 = " + p1.x +  ", " + p1.y + ", " +
p1.z);

System.out.println("p2 = " + p2.x +  ", " + p2.y + ", " +
p2.z);

System.out.println("p = " + p.x + ", " + p.y);

System.out.println("p1.distance(p2) = " + p1.distance(p2));

System.out.println("p1.distance(4, 6) = " + p1.distance(4, 6));

System.out.println("p1.distance(p) = " + p1.distance(p));

} }
```

Ниже приводится результат работы этой программы:

```
C:\> Java Point3DDist

p1 = 30, 40, 10

p2 = 0, 0, 0

p = 4, 6

p1.distance(p2) = 50.9902

p1.distance(4, 6) = 2.23607
```

```
p1.distance(p) = 2.23607
```

Обратите внимание — мы получили ожидаемое расстояние между трехмерными точками и между парой двумерных точек. В примере используется механизм, который называется *динамическим назначением методов* (**dynamic method dispatch**).

Динамическое назначение методов

Давайте в качестве примера рассмотрим два класса, у которых имеют простое родство подкласс / суперкласс, причем единственный метод суперкласса замещен в подклассе.

```
class A { void callme() {

System.out.println("Inside A's callrne method");

class B extends A { void callme() {

System.out.println("Inside B's callme method");

} }

class Dispatch {

public static void main(String args[]) {

A a = new B();

a.callme();

} }
```

Обратите внимание — внутри метода main мы объявили переменную a класса A, а проинициализировали ее ссылкой на объект класса B. В следующей строке мы вызвали метод callme. При этом транслятор проверил наличие метода callme у класса A, а исполняющая система, увидев, что на самом деле в переменной хранится представитель класса B, вызвала не метод класса A, а callme класса B. Ниже приведен результат работы этой программы:

```
C:\> Java Dispatch

Inside B's calime method
```

СОВЕТ

Программистам Delphi / C++ следует отметить, что все Java по умолчанию являются виртуальными функциями (ключевое слово **virtual**).

Рассмотренная форма динамического полиморфизма времени выполнения представляет собой один из наиболее мощных механизмов объектно-ориентированного программирования, позволяющих писать надежный, многократно используемый код.

final

Все методы и переменные объектов могут быть замещены по умолчанию. Если же вы хотите объявить, что подклассы не имеют права замещать какие-либо переменные и методы вашего класса, вам нужно объявить их как final (в Delphi / C++ не писать слово **virtual**).

```
final int FILE_NEW = 1;
```

По общепринятому соглашению при выборе имен переменных типа final — используются только символы верхнего регистра (т.е. используются как аналог препроцерных констант C++). Использование final-методов порой приводит к выигрышу в скорости выполнения кода — поскольку они не могут быть замещены, транслятору ничто не мешает заменять их вызовы *встроенным* (in-line) кодом (байт-код копируется непосредственно в код вызывающего метода).

finalize

В Java существует возможность объявлять методы с именем **finalize**. Методы finalize аналогичны деструкторам в C++ (ключевой знак **~**) и Delphi (ключевое слово **destructor**). Исполняющая среда Java будет вызывать его каждый раз, когда сборщик мусора соберется уничтожить объект этого класса.

static

Иногда требуется создать метод, который можно было бы использовать вне контекста какого-либо объекта его класса. Так же, как в случае main, все, что требуется для создания такого метода — указать при его объявлении модификатор типа **static.** Статические методы могут непосредственно обращаться только к другим статическим методам, в них ни в каком виде не допускается использование ссылок this и super. Переменные также могут иметь тип static, они подобны глобальным переменным, то есть доступны из любого места кода. Внутри статических методов недопустимы ссылки на переменные представителей. Ниже приведен пример класса, у которого есть статические переменные, статический метод и статический блок инициализации.

```
class Static {

static int a = 3;

static int b;

static void method(int x) {

System.out.println("x = " + x);

System.out.println("a = " + a);

System.out.println("b = " + b);

}

static {

System.out.println("static block initialized");

b = a * 4;

}

public static void main(String args[]) {

method(42);

} }
```

Ниже приведен результат запуска этой программы.

```
C:\> java Static static block initialized

X = 42

A = 3

B = 12
```

В следующем примере мы создали класс со статическим методом и несколькими статическими переменными. Второй класс может вызывать статический метод по имени и ссылаться на статические переменные непосредственно через имя класса.

```
class StaticClass {

static int a = 42;

static int b = 99;

static void callme() {

System.out.println("a = " + a);

} }

class StaticByName {

public static void main(String args[]) {

StaticClass.callme();

System.out.println("b = " + StaticClass.b);

} }
```

А вот и результат запуска этой программы:

```
C:\> Java StaticByName

a = 42 b = 99
```

abstract

Бывают ситуации, когда нужно определить класс, в котором задана структура какой-либо абстракции, но полная реализация всех методов отсутствует. В таких случаях вы можете с помощью модификатора типа abstract объявить, что некоторые из методов обязательно должны быть замещены в подклассах. Любой класс, содержащий методы abstract, также должен быть объявлен, как abstract. Поскольку у таких классов отсутствует полная реализация, их представителей нельзя создавать с помощью оператора new. Кроме того, нельзя объявлять абстрактными конструкторы и статические методы. Любой подкласс абстрактного класса либо обязан предоставить реализацию всех абстрактных методов своего суперкласса, либо сам должен быть объявлен абстрактным.

```
abstract class A {

abstract void callme();

void metoo() {

System.out.println("Inside A's metoo method");
```

```
} }

class B extends A {

void callme() {

System.out.println("Inside B's callme method");

} }

class Abstract {

public static void main(String args[]) {

A a = new B():

a.callme():

a.metoo():

} }
```

В нашем примере для вызова реализованного в подклассе класса А метода callme и реализованного в классе А метода metoo используется динамическое назначение методов, которое мы обсуждали раньше.

```
C:\> Java Abstract
```

Inside B's callrne method Inside A's metoo method

Классическое заключение

В этой главе вы научились создавать классы, конструкторы и методы. Вы осознали разницу между совмещением (overloading) и замещением (overriding) методов. Специальные переменные this и super помогут вам сослаться на текущий объект и на его суперкласс. В ходе эволюции языка Java стало ясно, что в язык нужно ввести еще несколько организационных механизмов - возможности более динамичного назначения методов и возможности более тонкого управления пространством имен класса и уровнями доступа к переменным и методам объектов

Глава 2. Iterator и Iterable

Итератор (от англ. *iterator*) — объект, абстрагирующий за единым интерфейсом доступ к элементам коллекции[1]. Итератор иногда также называют курсором, особенно если речь идет о базе данных.
В Обероне он называется также **бегунóк** и представлен как тип данных. В простейшем случае итератором в низкоуровневых языках является указатель.

В 70—80-х годах прошлого столетия, после того как была осознана важность правильной организации данных в определенную структуру, большое внимание уделялось изучению' и' Построению различных структур данных: связанных списков, очередей, деков, стеков, деревьев, сетей

Вместе с развитием структур данных развивались и алгоритмы работы с ними: сортировка, поиск, обход, хэширование.

Этим вопросам посвящена Обширная литература, посмотрите, например, книгу .

В 90-х годах было решено заносить данные в определенную коллекцию, скрыв ее внутреннюю структуру, а для работы с данными использовать методы этой коллекции.

Использование итераторов в обобщённом программировании позволяет реализовать универсальные алгоритмы работы с контейнерами.

Итератор похож на указатель своими основными операциями: указание одного отдельного элемента в коллекции объектов (называется *доступ к элементу*) и изменение себя так, чтобы указывать на следующий элемент (называется *перебор элементов*). Также должен быть определён способ создания итератора, указывающего на первый элемент контейнера, и способ узнать, перебраны ли все элементы контейнера. В зависимости от используемого языка и цели, итераторы могут поддерживать дополнительные операции или определять различные варианты поведения.

Главное предназначение итераторов заключается в предоставлении возможности пользователю обращаться к любому элементу контейнера при сокрытии внутренней структуры контейнера от пользователя. Это позволяет контейнеру хранить элементы любым способом при допустимости работы пользователя с ним как с простой последовательностью или списком. Проектирование класса итератора обычно тесно связано с соответствующим классом контейнера. Обычно контейнер предоставляет методы создания итераторов.

Необходимо отметить, что счётчик цикла иногда называют итератором цикла. Тем не менее, счётчик цикла обеспечивает только перебор элементов, но не доступ к элементу.

Интерфейс Iterator

В частности, задачу обхода возложили на саму коллекцию. В Java API введен интерфейс iterator , описывающий способ обхода всех элементов коллекции. В каждой коллекции есть метод iterator (), возвращающий реализацию интерфейса iterator для указанной коллекции. Получив эту реализацию, можно обходить коллекцию в некотором порядке, определенном данным итератором, с помощью методов, описанных в интерфейсе iterator и реализованных в этом итераторе. Подобная техника использована в классе

Отличия от индексации

В процедурных языках программирования широко используется индексация, основанная на счётчике цикла для перебора всех элементов последовательности, например, массива. Хотя индексация может использоваться совместно с некоторыми объектно-ориентированными контейнерами, использование итераторов даёт свои преимущества:

- Индексация не подходит для некоторых структур данных, в частности, для структур данных с медленным произвольным доступом или вообще без поддержки такового (например, список или дерево).

- Итераторы предоставляют возможность последовательного перебора любых структур данных, поэтому делают код более читаемым, удобным для повторного использования и менее чувствительным к изменениям структур данных.

- Итератор может накладывать дополнительные ограничения доступа, как например, проверка отсутствия пропусков элементов или повторного перебора одного и того же элемента.

- Итератор позволяет модифицировать объекты контейнера без влияния на сам итератор. Например, после того, как итератор уже «прошёл» первый элемент, можно вставить дополнительные элементы в начало контейнера без каких-либо нежелательных последствий. При использовании индексации это весьма проблематично из-за смены номеров индексов.

Возможность модификации контейнера во время итерации его элементов стала необходимой в современном объектно-ориентированном программировании, где взаимосвязи между объектами и последствия выполнения операций могут быть не слишком очевидными. Использование итератора избавляет от этих видов проблем.

БЕЗОПАСТНОСТЬ ЖИЗНЕДЕЯТЕЛЬНОСТИ ПРИ РАБОТЕ НА КОМПЬЮТОРЕ

Разработка инженерно-технических и организационных мероприятий по обеспечению безопасности труда

Для сохранения здоровья человека необходимо принимать меры по устранению или смягчению воздействия опасных факторов, описанных выше. В случае проектирования информационной системы необходимо предусмотреть следующие меры.

Мероприятия по снижению влияния электромагнитных излучений и электростатических полей

Уровни электромагнитных излучений от ЭВМ регламентируются следующими стандартами: Шведский национальный комитет по защите от излучений MPR II и Шведская конференция профсоюзов – TCO-95 и 2003 за рубежом и в России Санитарные нормы и правила 222/24 1340-03.

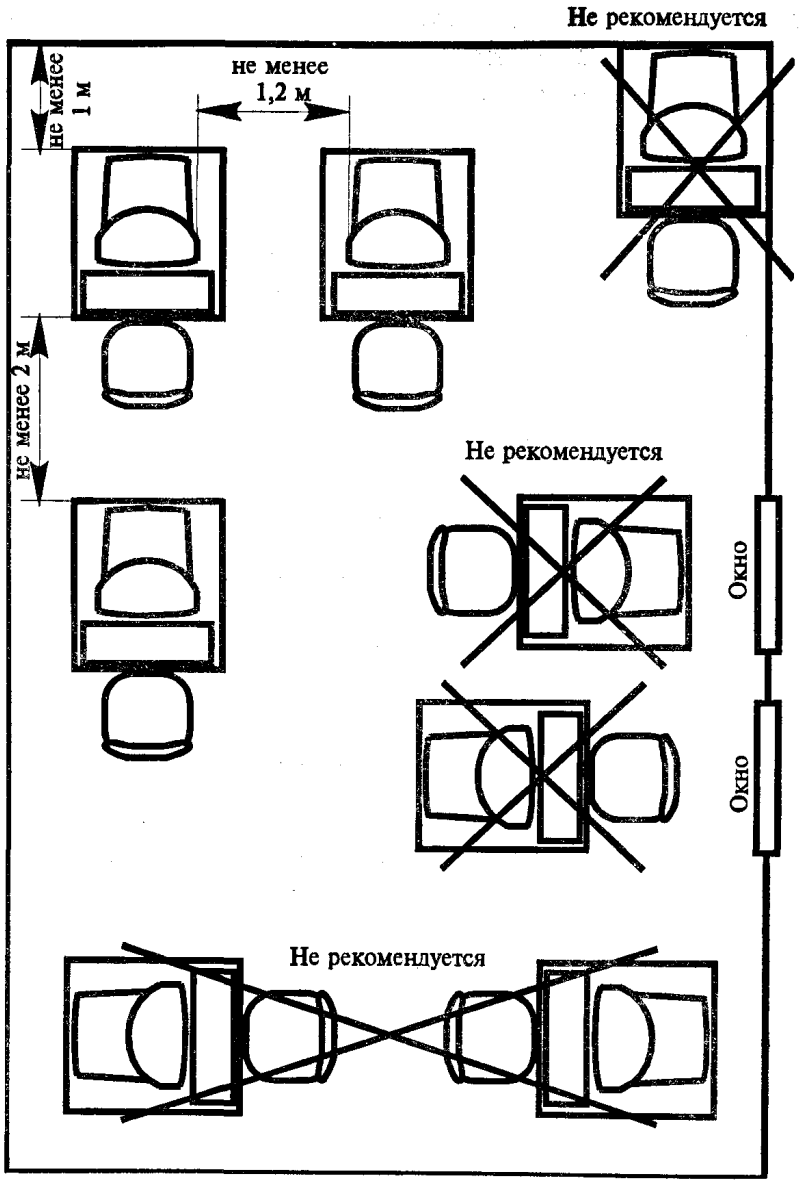
На задней стенке монитора, на табличке указывается, каким стандартам этот монитор удовлетворяет. Наиболее безопасные виды конструкций мониторов: Low Radiation, жидко-кристаллические мониторы, мониторы с установленной защитой по методу замкнутого металлического экрана (наиболее безопасны, но и самые дорогие).

Рекомендуется использовать специальные защитные экраны, поляроидные фильтры, которые устанавливаются на экран компьютеров типа IBM-PC/AT. Поляроидные фильтры резко снижают электромагнитное излучение, уменьшают электростатическое поле, устраняют блики на экране дисплея.

Целесообразно использовать приборы из набора «BIOACTIVATOR», уникального изобретения российских ученых, предлагаемых компанией «Vision». В частности, наклейки «АНТИРАДИАНТ», предназначенные для защиты от электромагнитных полей электробытовой техники. Механизм защиты основан не на устранении источника вредных излучений или его

экранировании, а на повышении общих неспецифических защитных свойств организма человека путем инверсии его собственных электромагнитных колебаний. Устройство крепится к мышке компьютера в месте соприкосновения поверхности прибора с ладонью.

Поскольку электромагнитное излучение исходит от всех частей монитора, (многие измерения показали, что уровень излучения по бокам и сзади монитора выше, чем спереди) наиболее безопасно установить компьютер в углу комнаты или в таком месте, где те, кто на нем работает, не казывались бы сбоку или сзади от машины.



Рекомендуемое расположение рабочих мест

Экран компьютера должен быть на расстоянии 70-120 см от глаз по рекомендации Центра электромагнитной безопасности. Применяйте специальный защитный экран. Также необходимо соблюдать оптимальный режим труда и отдыха.

Мероприятия по улучшению условий зрительной работы

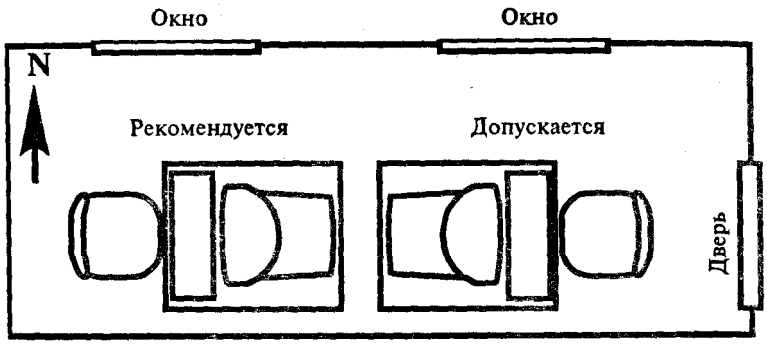
Снизить воздействие излучений и полей, уменьшить нагрузку на зрение человека можно так же путем подбора оптимального расстояния наблюдения информации. При работе с видеотерминалом это расстояние равно 600-700 мм.

Экран дисплея по высоте должен быть расположен так, чтобы угол между нормалью к центру экрана и горизонтальной линией взгляда составлял 20 градусов. В горизонтальной плоскости угол наблюдения экрана не должен превышать 60 градусов. Пульт дисплея следует располагать на столе или подставке так, чтобы высота клавиатуры пульта по отношению к полу составляла 650-720 мм. При размещении пульта на стандартном столе высотой 750 мм необходимо использовать кресло с регулируемой высотой сидения и подставку для ног.

Документ для ввода оператором данных рекомендуется располагать на расстоянии 450-500 мм от глаз оператора, преимущественно слева, при этом угол между экраном дисплея и документом в горизонтальной плоскости должен составлять 30-40 градусов. Угол наклона клавиатуры должен быть равен 15 градусов.

Экран дисплея, документы и клавиатура пульта располагают так, чтобы перепад яркостей поверхностей, зависящий от их расположения относительно источника света, не превышал 1:10 (рекомендуемое значение 1:3). Устройства документирования и другие, нечасто используемые технические средства, рекомендуется располагать справа от оператора в зоне максимальной досягаемости, а средства связи слева, чтобы освободить правую руку для записей.

Для снижения зрительной нагрузки так же необходимо правильно организовать режим работы персонала (пункт 3.2.3.) и освещенность рабочего места. Рабочие места с ВДТ и ПЭВМ по отношению к световым проемам должны располагаться так, чтобы естественный свет падал сбоку, преимущественно слева (рис. 4.2).



Рекомендуемое расположение рабочих мест

Важно принять меры по уменьшению отражений от монитора. Яркое и неровное освещение в комнате может вызвать неприятные отражения на экране. Возможные способы решения этой проблемы: выключение верхнего освещения, задерживание штор на окнах, которые пропускают слишком яркий свет, повороте монитора таким образом, чтобы не прямо перед ним, ни сзади не было ярких источников света. В цветовой композиции интерьера должны использоваться гармоничные сочетания (потолки белые, стена перед глазами работающего должна быть ярче остального фона). Применяется только общее освещение, возможно совмещенное (естественно-искусственное). Величина искусственной освещенности в горизонтальной плоскости не ниже 300 Лк.

Мероприятия по снижению статических физических нагрузок

Хороший результат снижения утомляемости при работе с ЭВМ дает правильная организация труда и отдыха. Согласно рекомендациям НИИ гигиены труда и профзаболеваний АМН РФ рабочий день за компьютером

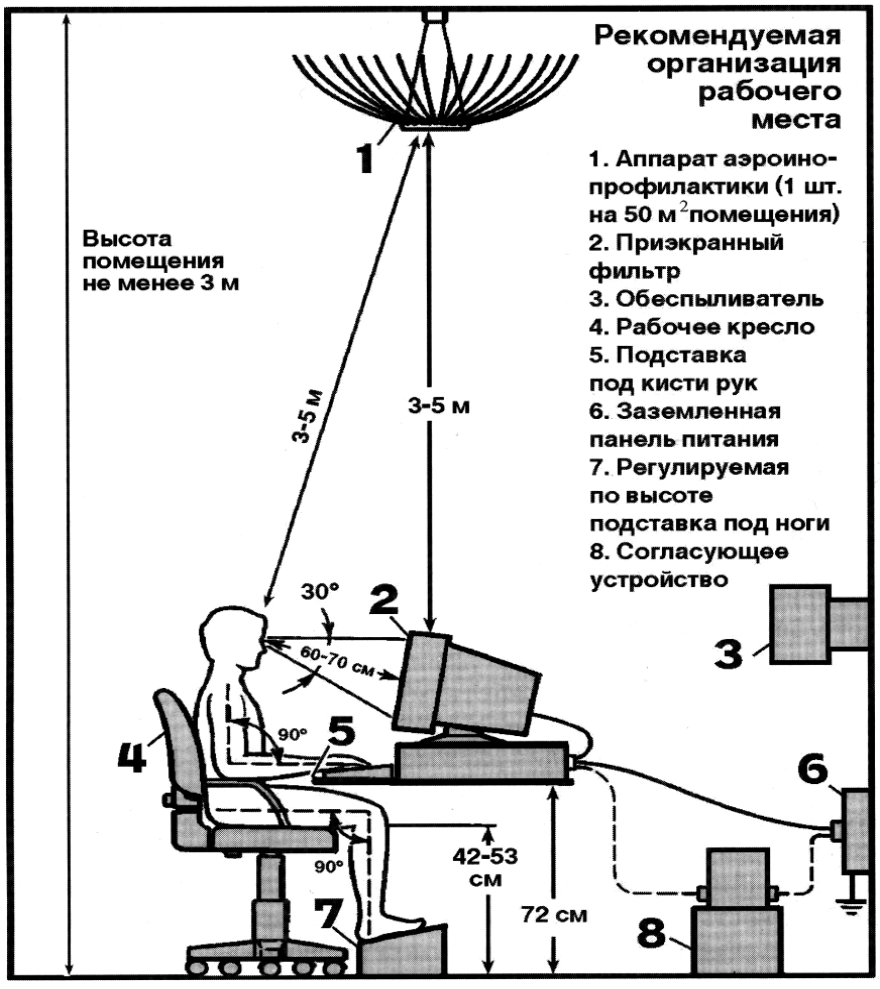
должен быть не более 6 часов, с дополнительными перерывами по 3 минуты через каждые полчаса, а через 2 часа работы по 15-20 минут.

Также для снижения статических нагрузок необходимо правильно организовать рабочее место. Высота рабочей поверхности стола должна регулироваться в пределах 680 – 800 мм; при отсутствии такой возможности высота рабочей поверхности стола должна составлять 725 мм. Модульными размерами рабочей поверхности стола для ВДТ и ПЭВМ, на основании которых должны рассчитываться конструктивные размеры, следует считать: ширину 800,100,1200 и 1400 мм, и глубину 800 и 1000 мм. при нерегулируемой его высоте, равной 720 мм. Рабочий стол должен иметь пространство для ног высотой не менее 600 мм, шириной – не менее 500 мм, глубиной на уровне колен – не менее 450 мм и на уровне вытянутых ног – не менее 650 мм. Рабочий стул (кресло) должен быть подъемно - поворотным и регулируемым по высоте и углам наклона сиденья и спинки, а также – расстоянию спинки от переднего края сиденья. Конструкция должна обеспечивать:

- ширину и глубину поверхности сиденья не менее 400 мм;
- поверхность с закругленным передним краем;
- регулировку высоты поверхности сиденья в пределах 400-550 мм и угол наклона вперед до 15 0 и назад до 50;
- высоту опорной поверхности спинки 300+-20 мм, ширину – не менее 380 мм и радиус кривизны горизонтальной плоскости – 400мм;
- угол наклона спинки в вертикальной плоскости в пределах 0-30 градусов;
- регулировку расстояния спинки от переднего края сиденья в пределах 260-400 мм;
- стационарные или съемные подлокотники длиной не менее 250 мм и шириной 50 -70мм;

- регулировку подлокотников по высоте над сиденьем в пределах 230+-30 мм и внутреннего расстояния между подлокотниками в пределах 350 -500мм.

Рабочее место с ВДТ и ПЭВМ должно быть оснащено легко перемещаемым пюпитром для документов. Клавиатуру следует располагать на поверхности стола на расстояние 100-300 мм от края, обращенного к пользователю, или специальной регулируемой по высоте рабочей поверхности, отделенной от основной столешницы.



Рекомендуемая организация рабочего места

Медики считают, что главное – это положение всего тела. При работе правильной считается поза, когда голеностопный, коленный, бедренный и

локтевой суставы согнуты под углом 90 градусов или чуть больше. Кисти прямые. Спина опирается на спинку кресла так, что сохраняется естественное положение шеи. Мышцы шеи и плеч расслаблены, взгляд направлен либо вперед, либо чуть вниз.

Положение тела должно соответствовать направлению взгляда:

- Нижний уровень экрана должен находиться на 20 см ниже уровня глаз
- Уровень верхней кромки экрана на высоте лба
- Кресло и клавиатуру устанавливают так, чтобы не надо было далеко тянуться.

Правильная поза минимизирует напряжение и снижает утомляемость. Однако, этого недостаточно. Врачи рекомендуют регулярно делать перерывы в монотонной работе для легкой зарядки или массажа, соблюдать режим работы.

Работа ЭВМ и вспомогательных устройств, связана с выделением тепла. При высокой температуре воздуха у людей, работающих в помещениях, возникает перегрев организма, что приводит к повышенному потовыделению и снижению работоспособности. Работа оператора по энергозатратам организма относится к 1 категории работ, т.е. работ легкой категории, которая выполняется сидя и затраты энергии не превышают 150 Ккал/час. Описание параметров «Оптимальных норм микроклимата для помещений с ВДТ и ПЭВМ» приводится в таблице.

Оптимальные нормы микроклимата для помещений с ВДТ и ПЭВМ

Период года	**Категория работ**	**Температура воздуха, С не более**	**Относит. влажность воздуха, %**	**Скорость движения воздуха, м/с**
Холодный	легкая - 1а	22 -24	40 - 60	0,1
	легкая - 1б	21 - 23	40 - 60	0,1
Теплый	легкая - 1а	23 - 25	40 - 60	0,1

Оптимальные нормы микроклимата для помещений с ВДТ и ПЭВМ

Период года	**Категория работ**	**Температура воздуха, С не более**	**Относит. влажность воздуха, %**	**Скорость движения воздуха, м/с**
	легкая - 1б	22 - 24	40 - 60	0,2

Для создания оптимальных метеоусловий в помещении рекомендую применять сочетание естественной вентиляции с кондиционированием воздуха.

Шум возникает вследствие работы вентилятора, находящегося в корпусе компьютера и виброакустических шумов на верхнем пороге слышимости, производимых строчным трансформатором дисплея. Так же присутствуют шумы, издаваемые накопителями на жестком и мягком магнитных дисках. Все эти шумы в целом оказывают достаточно сильное влияние на психику и общее состояние человека, вызывая чувства неуверенности, стесненности, тревоги, плохого самочувствия, что проводит к снижению производительности труда, возникновению ошибок.

Чтобы уменьшить уровень шумов в помещении, рекомендую использовать звукоизолирующие преграды, стенки и потолки отделывают специальными пористыми плитами, хорошо поглощающими звук.

Требования по электрической и пожарной безопасности

Энергоснабжение компьютера осуществляется через сеть бытового электропитания с номинальным напряжением 220 В. и частотой 50 Гц.

Для обеспечения электробезопасности персонала применяют защитное заземление, которое подключается к ЭВМ и вспомогательным устройствам через вилку электропитания. Так как бытовая электрическая сеть является сетью с напряжением до 1000 В., то защитное заземление применяется в трехфазных сетях переменного тока с изолированной нейтралью.

Сопротивление заземляющего устройства r_3 электроустановки напряжением до 1000 В. с изолированной нейтралью должно быть не более 4 Ом по ГОСТ 12.1.030-81. Защитное заземление используют для того, чтобы не возникало разности потенциалов между компьютером и периферийными устройствами, раздельно подключенными к электросети, а также между двумя соседними персональными компьютерами. Это особенно важно в случае работы в помещении достаточно большого количества пользователей.

Необходимо принять меры к предотвращению доступа пользователей к частям компьютера, находящихся под опасным напряжением, защитным корпусом.

Необходим контроль за состоянием изоляции. Работу по ремонту компьютеров следует производить только лицам, имеющим соответствующую подготовку и прошедшим инструктаж по технике безопасности.

Для профилактики пожара необходимо следить за состоянием электрических средств, своевременно производить их ремонт, правильно эксплуатировать аппаратуру.

Перед началом работы с компьютером персонал должен быть ознакомлен с правилами по технике безопасности, что должно быть зафиксировано в соответствующем журнале. Проведение инструктажа возлагается на ответственного сотрудника кафедры информатики.

Мероприятия по пожарной защите включают применение огнестойких конструкций и материалов в отделке помещения, использование средств оповещения и пожаротушения. В помещениях должны быть установлены средства связи для быстрого вызова городской пожарной части. Рабочее помещение должно быть оснащено углекислотными огнетушителями из расчета 1 на 100 $м^2$. Огнетушители должны располагаться на видном месте и в любое время суток к ним обеспечивается беспрепятственный доступ.

Учитывая выше изложенное, представляется целесообразным оснастить рабочие места компьютерами, подключенными к сети электропитания, имеющей защитное заземление.

Планировка рабочих мест должна быть продумана таким образом, чтобы обеспечить легкий доступ пользователей к своим рабочим местам и предотвратить возможность опрокидывания персоналом мониторов и периферийного оборудования при эвакуации, а также исключить возможность травматизма и несчастных случаев при эксплуатации. Работы по ремонту компьютеров должны производиться в отдельном помещении.

Мероприятия по повышению устойчивости функционирования проектируемой системы

Для повышения надежности и обеспечения устойчивости функционирования данного программного обеспечения целесообразно применять элементы с повышенными ресурсами, своевременно выполнять сервисное обслуживание, использовать сетевые фильтры (пилоты) для сглаживания скачков напряжения в сети и источники бесперебойного питания.

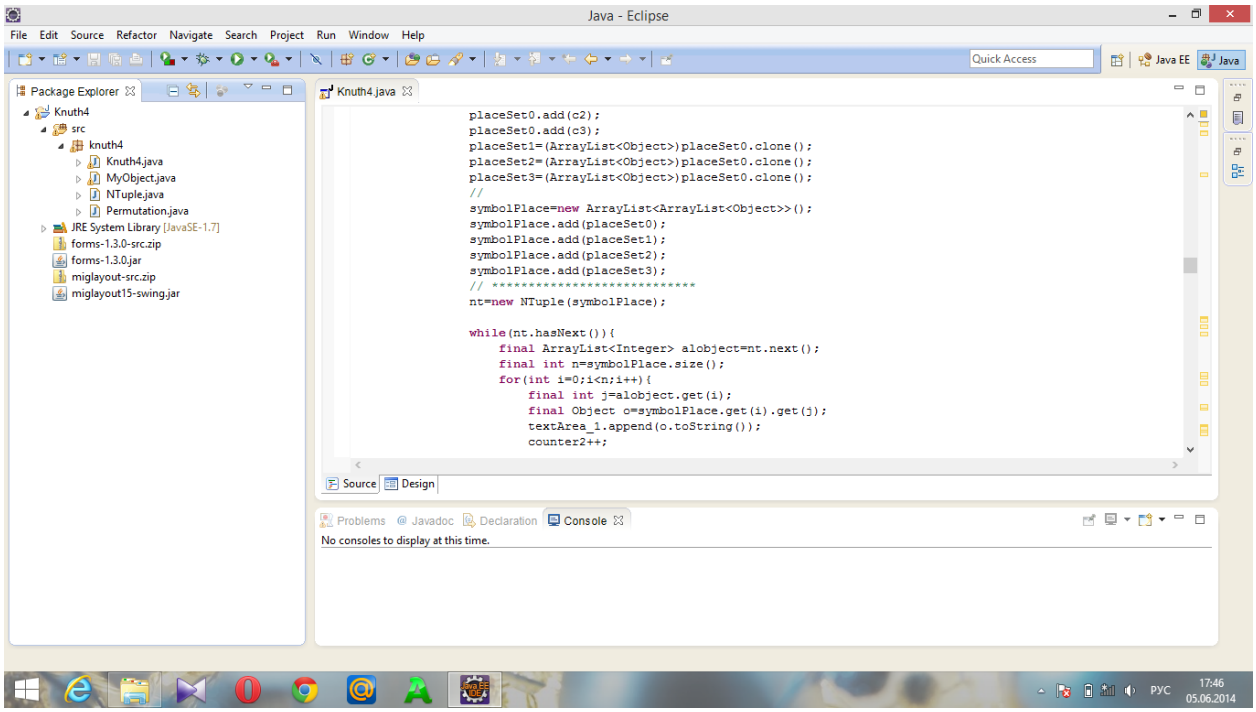
ЗАКЛЮЧЕНИЕ

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ:

1. Постановление Президента Республики Узбекистан принятое 21 марта 2012 года «О мерах по дальнейшему внедрению и развитию современных информационно-коммуникационных технологий.
2. Долинский М.С. Решение сложных и олимпиадных задач по программированию. – СПБ.:Питер,2006.
3. Кнут Д. Искусство программирования. Тот3: Поиск и сартировка.- М.:Издательский дом «Вильямс»,2000.
4. Кнут Д. Искусство программирования. Том1: Основные алгоритм.- М.:Издательский дом «Вильямс»,2000.
5. Кнут Д. Искусство программирования. Тот3: Поиск и сартировка.- М.:Издательский дом «Вильямс»,2000.
6. Элементы теории функции и функционального анализа. А. Н. Колмогоров, С. В. Фомин. Главная редакция физико-математической летературы нэд-ва «наука», М,1975г.
7. M. B. Wells, Elements of Combinatorial Computing, Pergaman Press, Oxford, 1971.
8. Безопастность жизнедеятельности. Под ред. С.В.Белова, М., 2002.
9. http://en.wikipedia.org/wiki/Stanford_University
10. http://en.wikipedia.org/wiki/Donald_Knuth
11. http://www-cs-faculty.stanford.edu/ ~ knuth/knuth/sgb/html for information about The Stanford GraphBase.
12. http://williamspublishing.com

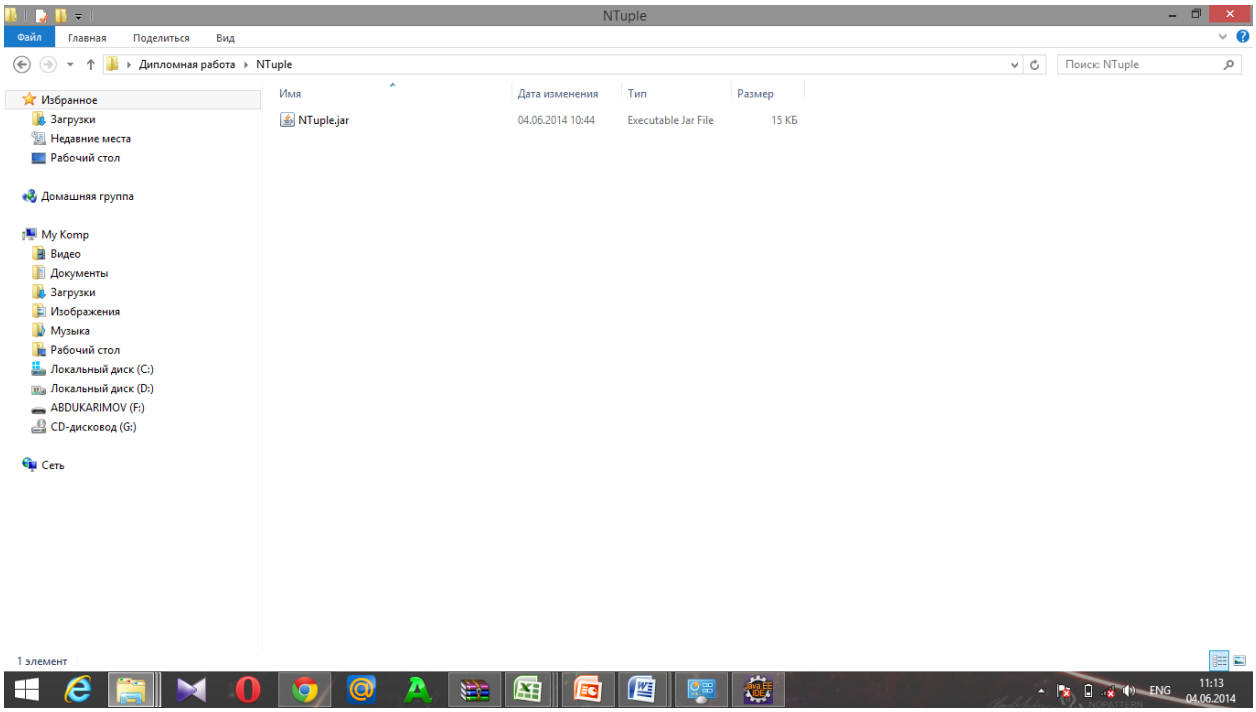
Процесс работы программы

Рабочая среда на Eclipce:

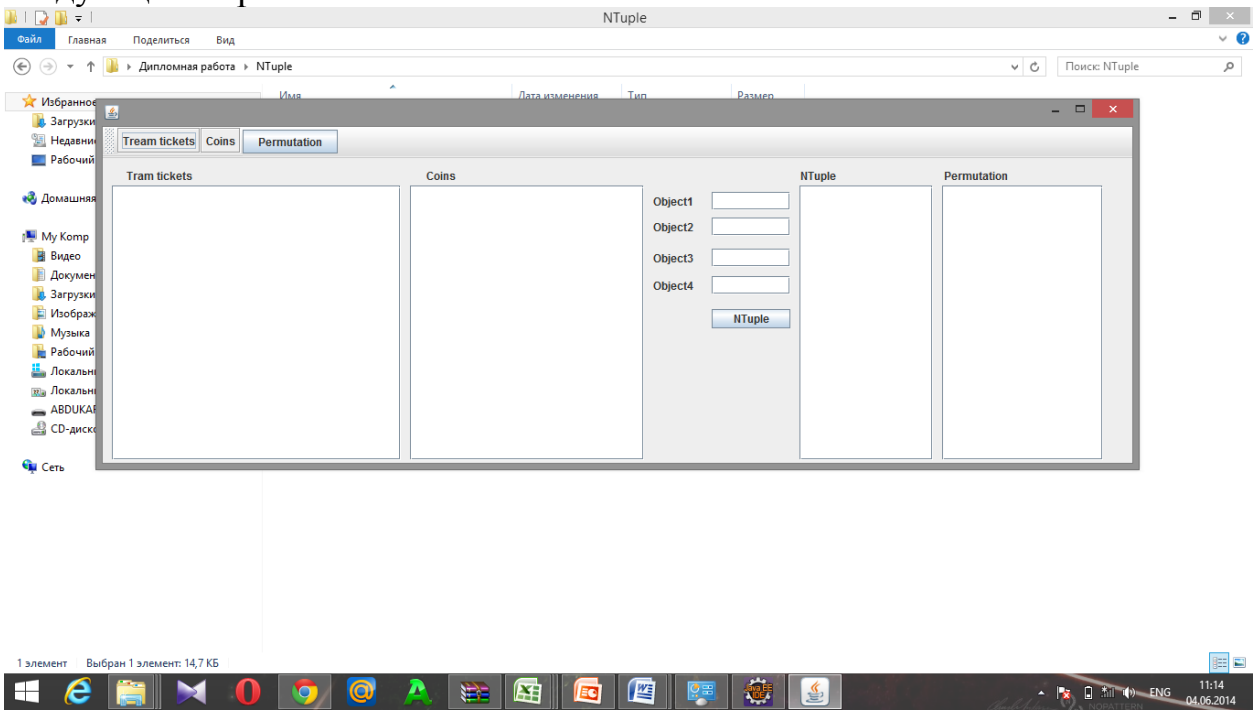


Принцип и ход использования программы:

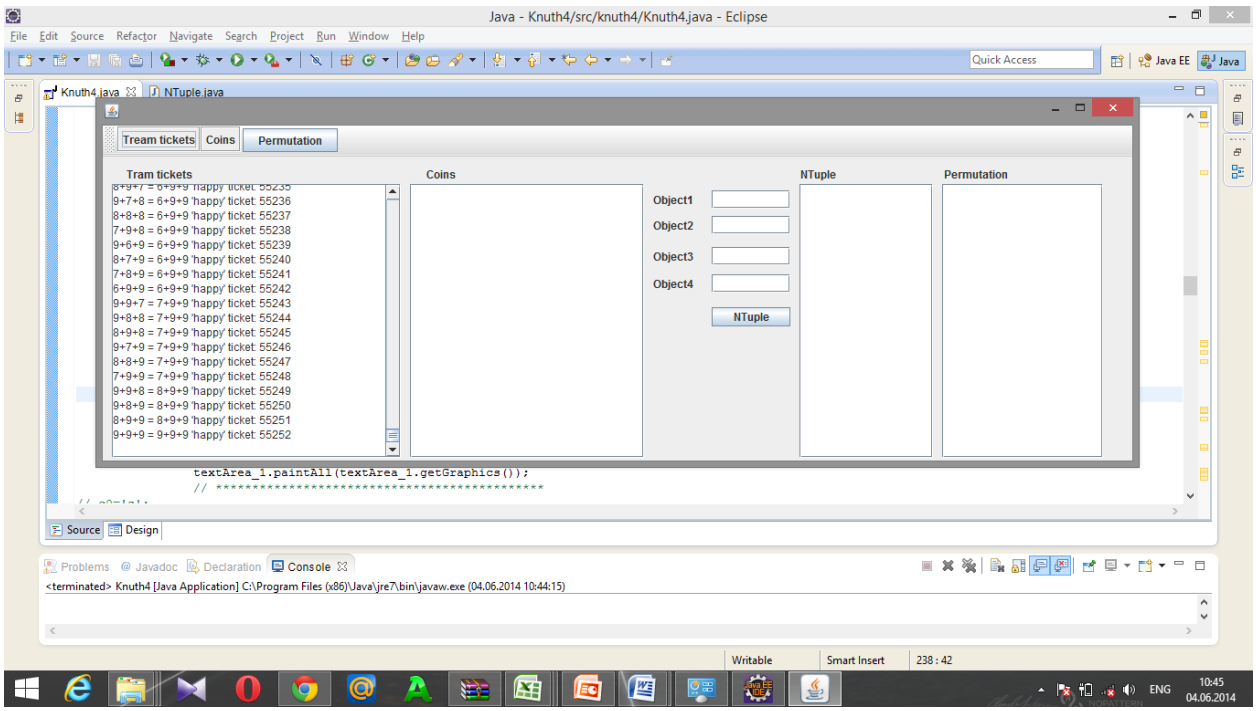
В первую очередь, для того, чтобы работать с программным прадуктом необходимо запустить jar файл называемое NTuple.jar.



После двайного клика запустится программа, интерфейс когой выглядит следующем образом:

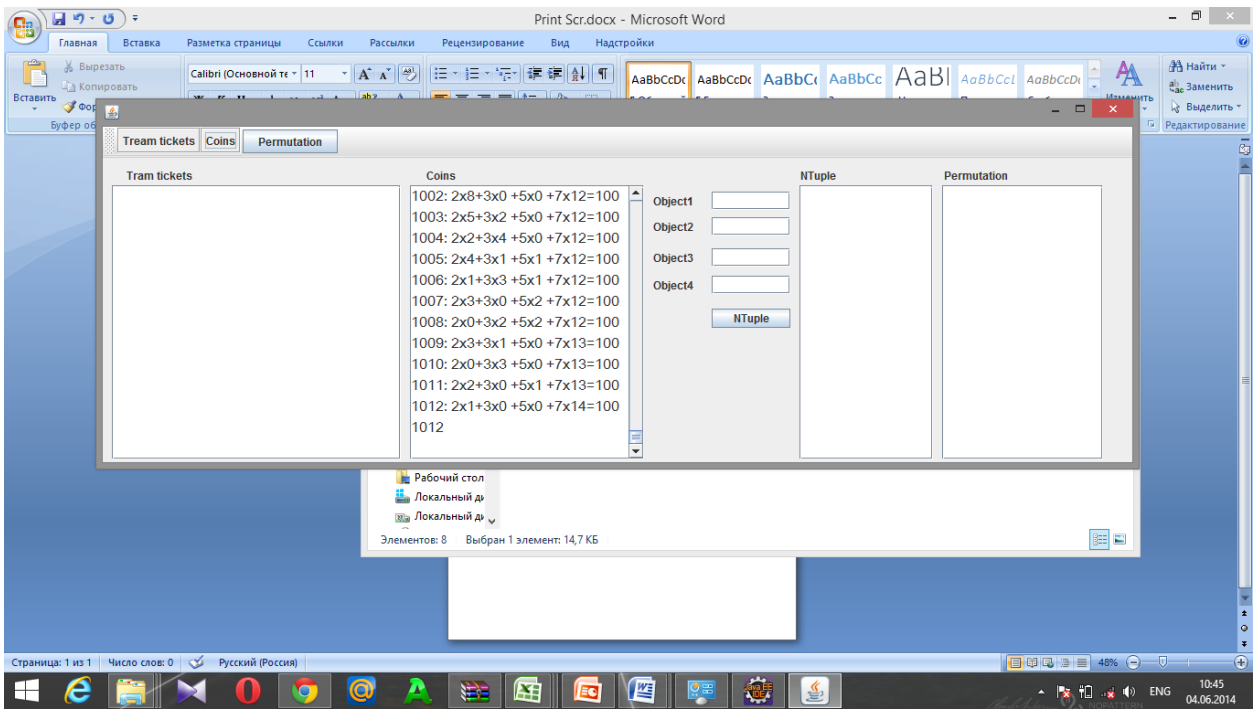


После нажатия на кнопку «Tream tickets» и у нас в первом текстовом поле появляются записи(«Счастливые билеты»).

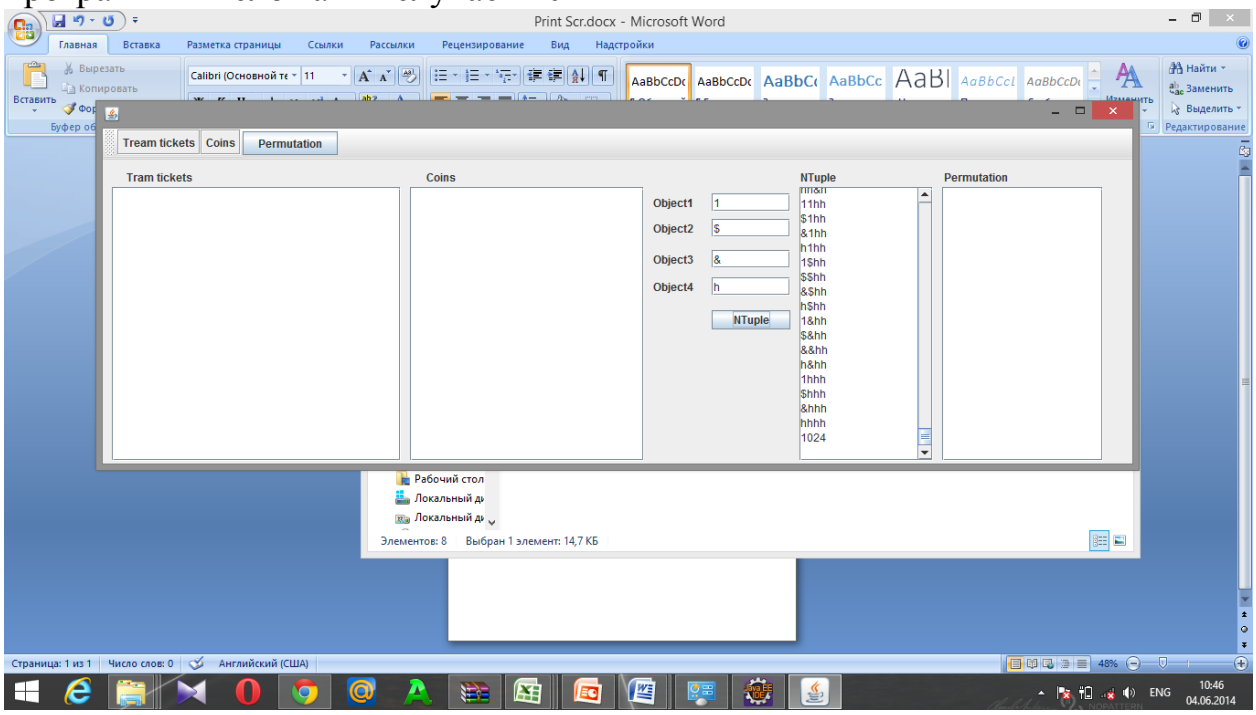


Задача о «Трамвайные билеты» при нажатии на кнопку «Tream tickets» генерируется числа от 0000000 до 999999 при этом выделяются все счастливые билеты удовлетворяющие условие M1+M3+M5=M2+M4+M6 (где M - пермутации).
После выполнения программы можно убедиться, что таких билетов 55252 шт.

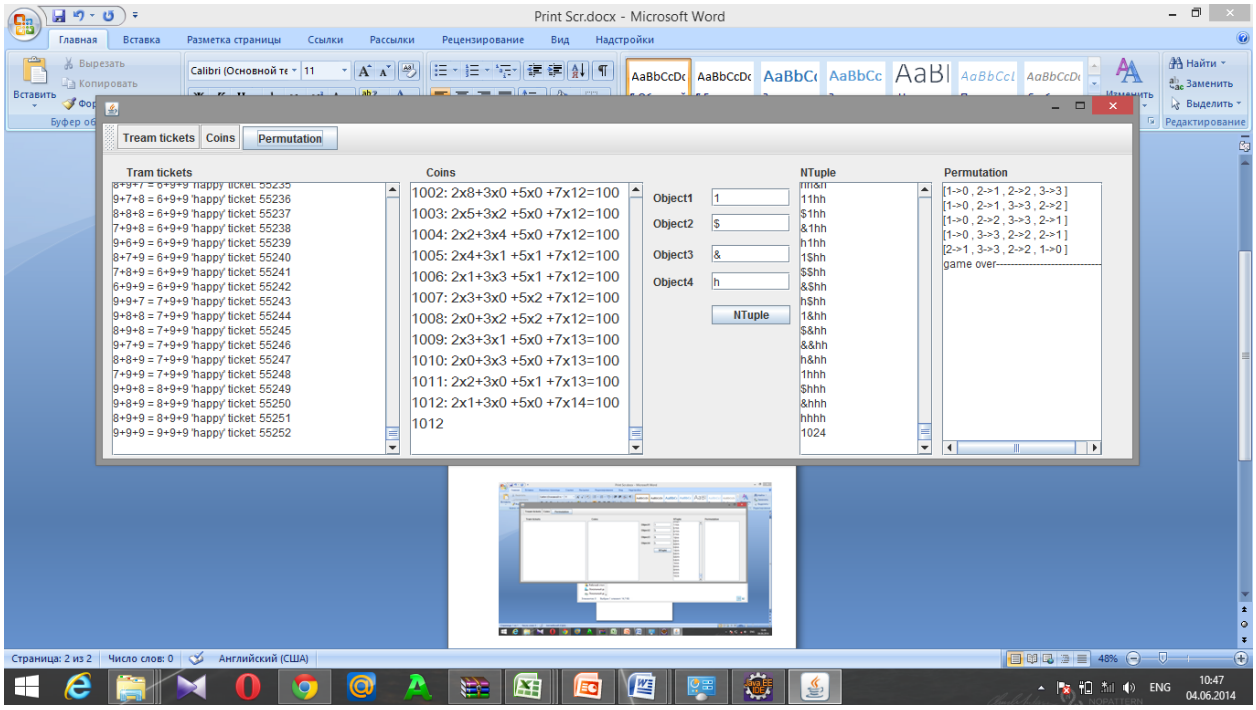
Итак следующий шаг, в ходе объяснения своей дипломной работы(Монеты) нам необходимо нажать на кнопку «Coins» и мы получим результат:



Задача «Монеты» находим все четвёрки всевозможных случаев X1,X2,X3,X4 удовлетворяющие условию A1*X1 + A2*X2 + A3*X3 + A4*X4 = M. Т.е. нам известно значения A1,A2,A3,A4 и M. Как видно из результата программы число таких случаев 1012



Для объяснения дипломной работы в качестве генерации всех кортежей мы раскрываем суть через двух примеров «Трамвайные билеты» и «Монеты» и генерируя в частном случае(4 объекта) объекты любого рода мы получим 1024 комбинацию всевозможные случаев. В данном случае мы в качестве первого объекта берём 1, в качестве второго знак $ потом & и h.



В итоге мы получим все результаты удовлетворяющие условию по отдельным конкретным задачам.

Код программы:

На языке программирования Java: