Класичний приватний університет

Кафедра програмування та інформаційних технологій

Борю С.Ю.

МЕТОДИЧНІ ВКАЗІВКИ
ДО ВИКОНАННЯ КОНТРОЛЬНОЇ РОБОТИ З ДИСЦИПЛІНИ

Мова програмування JAVA

для студентів денного і заочного відділення спеціальності
7.080404 Програмне забезпечення автоматизованих систем
7.080200 - Системний аналіз
"

Запоріжжя, 2008

Борю С.Ю.

Методичні вказівки до виконання контрольної роботи з дисципліни "Мова програмування JAVA" для студентів денного і заочного відділення спеціальності "Програмне забезпечення автоматизованих систем".
(КПУ - Запоріжжя, 2008, 36 сторінок).

Методичні вказівки регламентують порядок виконання контрольних робот з дисципліни " Мова програмування JAVA " для студентів денного і заочного відділення спеціальності "Програмне забезпечення автоматизованих систем".

В методичних вказівках висвітлені мета, завдання, порядок виконання та вимоги щодо оформлення контрольної роботи.

Рецензент – доц. Ермолаев В.А.

Друкується за рішенням Вченої Ради КПУ.

Протокол № ___ від “____”_____ 2008 року.

1 МЕТА

Метою виконання контрольної роботи з дисципліни "Мова програмування JAVA" є поглиблене вивчення студентами принципів, що використовуються при створенні сучасних комп'ютернім програм і програмних засобів.

2 ЗАВДАННЯ НА ВИКОНАННЯ КОНТРОЛЬНОЙ РОБОТИ

Контрольна робота з дисципліни "Мова програмування JAVA" передбачає два завдання. Ці завдання передбачають розкриття основних теоретичних питань та вирішення стандартних практичних задач. Перелік завдань - питання, практичні задачі та їх варіанти, наведено нижче.

Варіант завдання для виконання вибирається як значене залишку від ділення номера залікової книжки на число 24. У випадку, якщо номер залікової книжки менше 24, то вказаний номер є номером варіанту.

Завдання 1. Докладно розкрийте зміст теми Вашого варіанту, найменування якої виберіть з таблиці 1.

Таблиця 1

Варіанти тем першого завдання курсової роботи

Варіант	Тема завдання
1	Пакеты и интерфейсы в языке программирования JAVA. Составить пример демонстрационной программы с полными пояснениями.
2	Способы работы с символьними строками в языке программирования JAVA. Составить пример демонстрационной программы с полными пояснениями.
3	Способы обработки исключительных ситуаций в языке программирования JAVA. Составить пример демонстрационной программы с полными пояснениями.
4	Способы работы с программными процесами в языке программирования JAVA. Приоритеты процессов. Составить пример демонстрационной программы с полными пояснениями.
5	Способы работы с программными процесами в языке программирования JAVA. Синхронизация процессов. Составить пример демонстрационной программы с полными

	пояснениями.
6	Способы работы с программными процесами в языке программирования JAVA. Взаимодействие процессов. Составить пример демонстрационной программы с полными пояснениями.
7	Способы работы с программными процесами в языке программирования JAVA. Передача сообщений между процессами. Составить пример демонстрационной программы с полными пояснениями.
8	Простые утилиты
9	Способы организации операций ввода – вывода в языке программирования JAVA. Работа с файлами и каталогами. Составить пример демонстрационной программы с полными пояснениями.
10	Способы организации операций ввода – вывода в языке программирования JAVA.. Файлове потоки. Составить пример демонстрационной программы с полными пояснениями.
11	Способы организации операций ввода – вывода в языке программирования JAVA. Буферизируемые потоки. Составить пример демонстрационной программы с полными пояснениями.
12	Способы организации операций ввода – вывода в языке программирования JAVA. Консольный ввод – вывод. Составить пример демонстрационной программы с полными пояснениями.
13	Сетевые средства в языке программирования JAVA. Понятие и назначение сокетов. Составить пример демонстрационной программы с полными пояснениями.
14	Сетевые средства в языке программирования JAVA. Class URL – назначение и способы использования. Составить пример демонстрационной программы с полными пояснениями.
15	Понятие аплетов в языке программирования JAVA. Назначение и способы использования. Составить пример демонстрационной программы с полными пояснениями.
16	Передача параметров в аплеты, понятие контекста атлета в языке программирования JAVA. Составить пример демонстрационной программы с полными пояснениями.
17	Инициализация аплетов, задание размеров графической информации в языке программирования JAVA. Составить пример демонстрационной программы с полными пояснениями.
18	Простые методы Class Graphics, методы class Color в языке программирования JAVA. Составить пример демонстрационной программы с полными пояснениями.
19	Способы применения и использования шрифтов в языке программирования JAVA. Составить пример демонстрационной программы с полными пояснениями.
20	Абстракции для работы с окнами в языке программирования JAVA. Составить пример демонстрационной программы с полными пояснениями

21	Способы работы с символьними строками в языке программирования JAVA, преобразование строки в массив символов.. Составить пример демонстрационной программы с полными пояснениями.
22	Способы работы с программными процесами в языке программирования JAVA. Синхронизация процессов. Составить пример демонстрационной программы с полными пояснениями.
23	Способы организации операций ввода – вывода в языке программирования JAVA. Буферизируемые потоки. Составить пример демонстрационной программы с полными пояснениями.
24	Способы работы с программными процесами в языке программирования JAVA. Передача сообщений между процессами. Составить пример демонстрационной программы с полными пояснениями.
25	Способы организации операций ввода – вывода в языке программирования JAVA. Консольный ввод – вывод. Составить пример демонстрационной программы с полными пояснениями.

Завдання 2. Складіть програму на мови JAVA для Вашого варіанту задачі, зміст якої наведено ниже. У роботі наведіть загальну схему вирішення задачі, алгоритми найбільш важливих блоків та повний текст програми. Текст програми повинен супроводжуватися доцільними коментарями.

Варіанти задач другого завдання курсової роботи

Розробити клас, що містить набір методів (конструктор і методи дозволяють виконувати дії з об'єктом) і властивостей для програмної моделі заданого об'єкту. Опис об'єкту і його основних властивостей наводиться нижче. Привести фрагмент програми (public static main), що використовує об'єкти розробленого класу. Пояснити Ваш вибір методів і описати призначення кожного розробленого методу. В обов'язковому порядку розробити код програми для вказаних в завданні методів.

1. Объект «комплексные числа». Операции определяются по обще принятым формулам. Предусмотреть возможность арифметических операции присваивания, сложения, умножения и перевода в текстовую строку текущих значений. Конструктор должен позволить создавать объекты без и с начальной инициализацией.

2. Объект «комплексные числа». Операции определяются по обще принятым формулам. Предусмотреть возможность операции присваивания, вычитания, умножения и перевода в текстовую строку текущих значений. Конструктор должен позволить создавать объекты без и с начальной инициализацией.
3. Объект «комплексные числа». Операции определяются по обще принятым формулам. Предусмотреть возможность операции присваивания, сложения, деления и перевода в текстовую строку текущих значений. Конструктор должен позволить создавать объекты без и с начальной инициализацией.
4. Объект «комплексные числа». Операции определяются по обще принятым формулам. Предусмотреть возможность операции присваивания, сложения, умножения и перевода в показательную форму с возможностью распечатки на консоль. Конструктор должен позволить создавать объекты без и с начальной инициализацией.
5. Объект «вектор на плоскости» заданный в системе декартовых координат. Начало вектора расположено в начале координат. Операции определяются согласно обще принятых формул линейной (векторной) алгебры. Предусмотреть возможность операции присваивания, сложения, скалярного умножения и распечатки координат текущих значений. Конструктор должен позволить создавать объекты без и с начальной инициализацией.
6. Объект «вектор на плоскости» заданный в системе декартовых координат. Начало вектора расположено в начале координат. Операции определяются согласно обще принятых формул линейной (векторной) алгебры. Предусмотреть возможность операции присваивания, вычитания, скалярного умножения и распечатки координат текущих значений. Конструктор должен позволить создавать объекты без и с начальной инициализацией.
7. Объект «вектор на плоскости» заданный в системе декартовых координат. Начало вектора расположено в начале координат. Операции определяются согласно обще принятых формул линейной (векторной) алгебры. Предусмотреть возможность операции присваивания, сравнения модулей, скалярного умножения и распечатки координат текущих значений. Конструктор должен позволить создавать объекты без и с начальной инициализацией.
8. Объект «вектор на плоскости» заданный в системе декартовых координат. Начало вектора расположено в начале координат. Операции определяются согласно обще принятых формул линейной (векторной) алгебры. Предусмотреть возможность операции присваивания, нахождения угла между векторами, скалярного умножения и распечатки координат текущих значений. Конструктор должен позволить создавать объекты без и с начальной инициализацией.

9. Объект «равнобедренный треугольник заданный длинами сторон». Предусмотреть возможность операции присваивания, определения площади и периметра, а так же логический метод, определяющий существует или такой треугольник. Конструктор должен позволить создавать объекты без и с начальной инициализацией.
10. Объект «равносторонний треугольник заданный длинами сторон». Предусмотреть возможность операции присваивания, определения площади и периметра, а так же логический метод, определяющий существует или такой треугольник. Конструктор должен позволить создавать объекты без и с начальной инициализацией.
11. Объект «прямоугольный треугольник заданный длинами сторон». Предусмотреть возможность операции присваивания, определения площади и периметра, а так же логический метод, определяющий существует или такой треугольник. Конструктор должен позволить создавать объекты без и с начальной инициализацией.
12. Объект «равнобедренный треугольник заданный длиной равнобедренной стороной и углом между ними». Предусмотреть возможность операции присваивания, определения площади и периметра, а так же логический метод, отвечающий на вопрос – остро или тупо угольным является заданный треугольник. Конструктор должен позволить создавать объекты без и с начальной инициализацией.
13. Объект «треугольник заданный длиной двух стороной и углом между ними». Предусмотреть возможность операции присваивания, определения площади и периметра, а так же логический метод, отвечающий на вопрос – остро или тупо угольным является заданный треугольник. Конструктор должен позволить создавать объекты без и с начальной инициализацией.
14. Объект «прямоугольник заданный длинами двух сторон». Предусмотреть возможность операции присваивания, определения площади и периметра, а так же логический метод, отвечающий на вопрос – является ли прямоугольник квадратом. Конструктор должен позволить создавать объекты без и с начальной инициализацией.
15. Объект «множество целых чисел заданной мощности». Предусмотреть возможность операции присваивания, объединения двух множеств, вывода на печать элементов множества, а так же метод отвечающий на вопрос – принадлежит ли указанное значение множеству. Конструктор должен позволить создавать объекты без и с начальной инициализацией. Мощность множества задается при создании объекта.
16. Объект «множество вещественных чисел заданной мощности». Предусмотреть возможность операции присваивания, объединения двух множеств, вывода на печать элементов множества, а так же метод отвечающий на вопрос – принадлежит ли указанное значение множеству. Конструктор должен позволить создавать объекты без и с начальной инициализацией. Мощность множества задается при создании объекта.

17. Объект «множество символов заданной мощности». Предусмотреть возможность операции присваивания, объединения двух множеств, вывода на печать элементов множества, а так же метод отвечающий на вопрос – принадлежит ли указанное значение множеству. Конструктор должен позволить создавать объекты без и с начальной инициализацией. Мощность множества задается при создании объекта.
18. Объект «множество целых чисел удвоенной длины заданной мощности». Предусмотреть возможность операции присваивания, объединения двух множеств, вывода на печать элементов множества, а так же метод отвечающий на вопрос – принадлежит ли указанное значение множеству. Конструктор должен позволить создавать объекты без и с начальной инициализацией. Мощность множества задается при создании объекта.
19. Объект «множество вещественных чисел удвоенной точности заданной мощности». Предусмотреть возможность операции присваивания, объединения двух множеств, вывода на печать элементов множества, а так же метод отвечающий на вопрос – принадлежит ли указанное значение множеству. Конструктор должен позволить создавать объекты без и с начальной инициализацией. Мощность множества задается при создании объекта.
20. Объект «множество байт заданной мощности». Предусмотреть возможность операции присваивания, объединения двух множеств, вывода на печать элементов множества, а так же метод отвечающий на вопрос – принадлежит ли указанное значение множеству. Конструктор должен позволить создавать объекты без и с начальной инициализацией. Мощность множества задается при создании объекта.
21. Объект «множество целых чисел не заданной (переменной) мощности». Предусмотреть возможность операции добавить элемент к множеству, определение количество элементов в множестве, вывода на печать всех элементов множества, а так же метод удаляющий указанный элемент из множества, если этот элемент принадлежит множеству. Конструктор должен позволить создавать объекты без и с начальной инициализацией.
22. Объект «множество вещественных чисел не заданной (переменной) мощности». Предусмотреть возможность операции добавить элемент к множеству, определение количество элементов в множестве, вывода на печать всех элементов множества, а так же метод удаляющий указанный элемент из множества, если этот элемент принадлежит множеству. Конструктор должен позволить создавать объекты без и с начальной инициализацией.
23. Объект «множество символов не заданной (переменной) мощности». Предусмотреть возможность операции добавить элемент к множеству, определение количество элементов в множестве, вывода на печать всех элементов множества, а так же метод удаляющий указанный элемент из

множества, если этот элемент принадлежит множеству. Конструктор должен позволить создавать объекты без и с начальной инициализацией.

24. Объект «множество целых чисел удвоенной длины не заданной (переменной) мощности». Предусмотреть возможность операции добавить элемент к множеству, определение количество элементов в множестве, вывода на печать всех элементов множества, а так же метод удаляющий указанный элемент из множества, если этот элемент принадлежит множеству. Конструктор должен позволить создавать объекты без и с начальной инициализацией.
25. Объект «прямоугольник заданный длинами двух сторон». Предусмотреть возможность операции присваивания, определения площади и периметра, а так же логический метод, отвечающий на вопрос – содержится ли, указанный параметрами метода прямоугольник , внутри прямоугольника. Конструктор должен позволить создавать объекты без и с начальной инициализацией

У ДОДАТКУ С наводяться методична вказівка і приклад рішення одного з варіантів другого завдання.

3 ЛІТЕРАТУРА

Нижче наведений перелік навчальних посібників на наукових видань з питань програмування, що рекомендуються для використання при виконанні роботи

1. Нотон П.**JAVA:Справ.руководство**:Пер.с англ./Под ред.А.Тихонова.-М.:БИНОМ:Восточ.Кн.Компания,1996:Восточ.Кн.Компания.-447с..-(Club Computer)
2. Патрик Нотон, Герберт Шилдт **Полный справочник по Java**.-McGraw-Hill,1997, Издательство "Диалектика",1997
3. Дэвид Флэнэген **Java in a Nutshell**.- O'Reilly & Associates, Inc., 1997, Издательская группа BHV, Киев, 1998
4. Ренеган Э.Дж.(мл.)**1001 адрес WEB для программистов**:Новейший путеводитель программиста по ресурсам World Wide Web:Пер.с англ..-Минск:Попурри,1997.-512с.ил.
5. Сокольский М.В.**Все об Intranet и Internet**.-М.:Элиот,1998.-254с.ил.
6. Чен М.С. и др.**Программирование на JAVA:1001 совет:Наиболее полное руководство по Java и Visual J++**:Пер.с англ./Чен М.С.,Грифис С.В.,Изи Э.Ф..-Минск:Попурри,1997.-640с.ил.+ Прил.(1диск.)

7. Майкл Эферган **Java: справочник**.- QUE Corporation, 1997, Издательство "Питер Ком", 1998
8. Джо Вебер **Технология Java в подлиннике**.- QUE Corporation, 1996, "BHV-Санкт-Петербург",1997
9. Джейсон Мейнджер **Java: Основы программирования**.- McGraw-Hill,Inc.,1996, Издательская группа BHV, Киев,1997
10. И.Ю.Баженова **Язык программирования Java**.- АО "Диалог-МИФИ", 1997
11. Джон Родли **Создание Java-апплетов**.- The Coriolis Group,Inc.,1996, Издательство НИПФ "ДиаСофт Лтд.",1996
12. Майкл Томас, Пратик Пател, Алан Хадсон, Доналд Болл(мл.) **Секреты программирования для Internet на Java**.- Ventana Press, Ventana Communications Group, U.S.A.,1996, Издательство "Питер Пресс", 1997
13. Аарон И.Волш **Основы программирования на Java для World Wide Web**.- IDG Books Worldwide,Inc.,1996, Издательство "Диалектика",1996
14. Кен Арнольд, Джеймс Гослинг **Язык программирования Java**.- Addison-Wesley Longman,U.S.A.,1996, Издательство "Питер-Пресс", 1997
15. Нейл Бартлетт, Алекс Лесли, Стив Симкин **Программирование на Java. Путеводитель**.- The Coriolis Group,Inc.,1996, Издательство НИПФ "ДиаСофт Лтд.",1996
16. Крис Джамса **Библиотека программиста Java**.- Jamsa Press, 1996, ООО "Попурри", 1996

ДОДАТОК А

Класичний приватний університет

ІНСТІТУТ УПРАВЛІННЯ

КАФЕДРА ПРОГРАМУВАННЯ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ПОЯСНЮВАЛЬНА ЗАПИСКА

до контрольної роботи по дисципліні "Мова програмування JAVA"

варіант № "_____"

Підготовив студент групи . _____
підпис

Керівник
доцент, к.т.н. Борю С.Ю. _____
підпис

Запоріжжя
2008

ДОДАТОК Б

Постановка задач:

1 – перше завдання

2 – друге завдання

Вимога до програмної реалізації
(як правило, узгоджуються студентом з керівником роботи):

a. формати вхідних і вихідних даних;
b. мова програмування, середовище і інші додаткові вимоги до практичної реалізації програми;
c. опис методики тестування програми, тестові набори початкових даних;
d. форма надання реалізованої програми.

ДОДАТОК С

Объектно-ориентированное программирование в Java

Вся полувековая история программирования компьютеров, а может быть, и история всей науки — это попытка совладать со сложностью окружающего мира. Задачи, встающие перед программистами, становятся все более громоздкими, информация, которую надо обработать, растет как снежный ком. Еще недавно обычными единицами измерения информации были килобайты и мегабайты, а сейчас уже говорят только о гигабайтах и терабайтах. Как только программисты предлагают более-менее удовлетворительное решение предложенных задач, тут же возникают новые, еще более сложные задачи. Программисты придумывают новые методы, создают новые языки. За полвека появилось несколько сотен языков, предложено множество методов и стилей. Некоторые методы и стили становятся общепринятыми и образуют на некоторое время так называемую *парадигму программирования.*

Парадигмы программирования

Первые, даже самые простые программы, написанные в машинных кодах, составляли сотни строк совершенно непонятного текста. Для упрощения и ускорения программирования придумали языки высокого уровня: FORTRAN, Algol и сотни других, возложив рутинные операции по созданию машинного кода на компилятор. Те же программы, переписанные на языках высокого уровня, стали гораздо понятнее и короче. Но жизнь потребовала решения более сложных задач, и программы снова увеличились в размерах, стали необозримыми.

Возникла идея: оформить программу в виде нескольких, по возможности простых, процедур или функций, каждая из которых решает свой определенную задачу. Написать, откомпилировать и отладить небольшую процедуру можно легко и быстро. Затем остается только собрать все процедуры в нужном порядке в одну программу. Кроме того, один раз написанные процедуры можно затем использовать в других программах как строительные кирпичики. *Процедурное программирование* быстро стало парадигмой. Во все языки высокого уровня включили средства написания процедур и функций. Появилось множество библиотек процедур и функций на все случаи жизни.

Встал вопрос о том, как выявить структуру программы, разбить программу на процедуры, какую часть кода выделить в отдельную процедуру, как сделать алгоритм решения задачи простым и наглядным, как удобнее связать процедуры между собой. Опытные программисты предложили свои рекомендации, названные *структурным программированием.* Структурное программирование оказалось удобным и стало парадигмой. Появились языки программирования, например Pascal, на которых удобно писать структурные программы. Более того, на них очень трудно написать неструктурные программы.

Сложность стоящих перед программистами задач проявилась и тут: программу стали содержать сотни процедур, и опять оказались необозримыми. "Кирпичики" стали слишком маленькими. Потребовался новый стиль программирования,

В это же время обнаружилось, что удачная или неудачная структура исходных данных может сильно облегчить или усложнить их обработку. Одни исходные данные удобнее объединить в массив, для других больше подходит структура дерева или стека. Никлаус Вирт даже назвал свою книгу "Алгоритмы + структуры данных = программы".

Возникла идея объединить исходные данные и все процедуры их обработки в один модуль. Эта идея *модульного программирования* быстро завоевала умы и на некоторое время стала парадигмой. Программы составлялись из отдельных модулей, содержащих десяток-другой процедур и функций. Эффективность таких программ тем выше, чем меньше модули зависят друг от друга. Автономность модулей позволяет создавать и библиотеки модулей, чтобы потом использовать их в качестве строительных блоков для программы.

Для того чтобы обеспечить максимальную независимость модулей друг от друга, надо четко отделить процедуры, которые будут вызываться другими модулями,— *открытые* (public) процедуры, от вспомогательных, которые обрабатывают данные, заключенные в этот модуль, —

закрытых (private) процедур. Первые перечисляются в отдельной части модуля — *интерфейсе* (interface), вторые участвуют только в *реализации* (implementation) модуля. Данные, занесенные в модуль, тоже делятся на открытые, указанные в интерфейсе и доступные для других модулей, и закрытые, доступные только для процедур того же модуля. В разных языках программирования это деление производится по-разному. В языке Turbo Pascal модуль специально делится на интерфейс и реализацию в.языке С интерфейс выносится в отдельные "головные" (header) файлы. В даыке С++, кроме того, для описания интерфейса можно воспользоваться абстрактными классами. В языке Java есть специальная конструкция для описания интерфейсов, которая так и называется — interface, но можно написать и абстрактные классы.

Так возникла идея о скрытии, *инкапсуляции* (incapsulation) данных и методов их обработки. Подобные идеи периодически возникают в дизайне бытовой техники. То телевизоры испещряются кнопками и топорщатся ручками и движками на радость любознательному телезрителю, господствует "приборный" стиль, то вдруг все куда-то пропадает, а на панели остаются только кнопка включения и ручка громкости. Любознательный телезритель берется за отвертку.

Инкапсуляция, конечно, производится не для того, чтобы спрятать от другого модуля что-то любопытное. Здесь преследуются две основные цели. Первая — обеспечить безопасность использования модуля, вынести в интерфейс, сделать общедоступными только те методы обработки информации, которые не могут испортить или удалить исходные данные. Вторая цель — уменьшить сложность, скрыв от внешнего мира ненужные детали реализации.

Опять возник вопрос, каким образом разбить программу на модули? Тут кстати оказались методы решения старой задачи программирования — моделирования действий искусственных и природных объектов: роботов, станков с программным управлением, беспилотных самолетов, людей, животных, растений, систем обеспечения жизнедеятельности, систем управления технологическими процессами.

В самом деле, каждый объект — робот, автомобиль, человек — обладает определенными характеристиками. Ими могут служить: вес, рост, максимальная скорость, угол поворота, грузоподъемность, фамилия, возраст. Объект может производить какие-то действия: перемещаться в пространстве, поворачиваться, поднимать, копать, расти или уменьшаться, есть, пить, рождаться и умирать, изменяя свои первоначальные характеристики. Удобно смоделировать объект в виде модуля. Его характеристики будут данными, постоянными или переменными, а действия — процедурами.

Оказалось удобным сделать и обратное — разбить программу на модули так, чтобы она превратилась в совокупность взаимодействующих объектов. Так возникло *объектно-ориентированное программирование* (object-oriented programming), сокращенно ООП (ООР) — современная парадигма программирования.

В виде объектов можно представить совсем неожиданные понятия. Например, окно на экране дисплея — это объект, имеющий ширину width и высоту height , расположение на экране, описываемое обычно координатами (x, y) левого верхнего угла окна, а также шрифт, которым в окно выводится текст, скажем, Times New Roman, цвет фона color , несколько кнопок, линейки прокрутки и другие характеристики. Окно может перемещаться по экрану методом move() , увеличиваться или уменьшаться в размерах методом size() , сворачиваться в ярлык методом iconify() , как-то реагировать на действия мыши и нажатия клавиш. Это полноценный объект! Кнопки, полосы прокрутки и прочие элементы окна — это тоже объекты со своими размерами, шрифтами, перемещениями.

Разумеется, считать, что окно само "умеет" выполнять действия, а мы только даем ему поручения: "Свернись, развернись, передвинься", — это несколько неожиданный взгляд на вещи, но ведь сейчас можно подавать команды не только мышью и клавишами, но и голосом!

Идея объектно-ориентированного программирования оказалась очень плодотворной и стала активно развиваться. Выяснилось, что удобно ставить задачу сразу в виде совокупности действующих объектов — возник *объектно-ориентированный анализ,* ООА. Решили проектировать сложные системы в виде объектов — *появилось объектно-ориентированное проектирование,* ООП (OOD, object-oriented design).

Рассмотрим подробнее принципы объектно-ориентированного программирования.

Принципы объектно-ориентированного программирования

Объектно-ориентированное программирование развивается уже более двадцати лет. Имеется несколько школ, каждая из которых предлагает свой набор принципов работы с объектами и по-своему излагает эти принципы. Но есть несколько общепринятых понятий. Перечислим их.

Абстракция

Описывая поведение какого-либо объекта, например автомобиля, мы строим его модель. Модель, как правило, не может описать объект полностью, реальные объекты слишком сложны. Приходится отбирать только те характеристики объекта, которые важны для решения поставленной перед нами задачи. Для описания грузоперевозок важной характеристикой будет грузоподъемность автомобиля, а для описания автомобильных гонок она не существенна. Но для моделирования гонок обязательно надо описать метод набора скорости данным автомобилем, а для грузоперевозок это не столь важно.

Мы должны *абстрагироваться* от некоторых конкретных деталей объекта. Очень важно выбрать правильную степень абстракции. Слишком высокая степень даст только приблизительное описание объекта, не позволит правильно моделировать его поведение. Слишком низкая степень абстракции сделает модель очень сложной, перегруженной деталями, и потому непригодной.

Например, можно совершенно точно предсказать погоду на завтра в определенном месте, но расчеты по такой модели продлятся трое суток даже на самом мощном компьютере. Зачем нужна модель, опаздывающая на два дня? Ну а точность модели, используемой синоптиками, мы все знаем сами. Зато расчеты по этой модели занимают всего несколько часов.

Описание каждой модели производится в виде одного или нескольких *классов* (classes). Класс можно считать проектом, слепком, чертежом, по которому затем будут создаваться конкретные объекты. Класс содержит описание переменных и констант, характеризующих объект. Они называются *полями класса* (class fields). Процедуры, описывающие поведение объекта, называются *методами класса* (class methods). Внутри класса можно описать и *вложенные классы* (nested classes) и *вложенные интерфейсы.* Поля, методы и вложенные классы первого уровня являются *членами класса* (class members). Разные школы объектно-ориентированного программирования предлагают разные термины, мы используем терминологию, принятую в технологии Java.

Вот набросок описания автомобиля:

```
class Automobile{
int maxVelocity;  // Поле, содержащее наибольшую скорость автомобиля
int speed;        // Поле, содержащее текущую скорость автомобиля
int weight;       // Поле, содержащее вес автомобиля
                  // Прочие поля...
void moveTo(int x, int y){   // Метод, моделирующий перемещение
                             // автомобиля. Параметры x и y — не поля
int a = 1;   // Локальная переменная — не поле
             // Тело метода. Здесь описывается закон
             // перемещения автомобиля в точку (x, y)
}
// Прочие методы. . .
}
```

Знатокам Pascal

В Java нет вложенных процедур и функций, в теле метода нельзя описать другой метод.

После того как описание класса закончено, можно создавать конкретные объекты, *экземпляры* (instances) описанного класса. Создание экземпляров производится в три этапа, подобно описанию массивов. Сначала объявляются ссылки на объекты: записывается имя класса, и через пробел перечисляются экземпляры класса, точнее, ссылки на них.

```
Automobile Iada2110, fordScorpio, oka;
```

Затем операцией new определяются сами объекты, под них выделяется оперативная память, ссылка получает адрес этого участка в качестве своего значения.

```
lada2110 = new Automobile();
fordScorpio = new Automobile();
oka = new Automobile();
```

На третьем этапе происходит инициализация объектов, задаются начальные значения. Этот этап, как правило, совмещается со вторым, именно для этого в операции new повторяется имя класса со скобками Automobile () . Это так называемый *конструктор* (constructor) класса, но о нем поговорим попозже.

Поскольку имена полей, методов и вложенных классов у всех объектов одинаковы, они заданы в описании класса, их надо уточнять именем ссылки на объект:

```
lada2110.maxVelocity = 150;
fordScorpio.maxVelocity = 180;
oka.maxVelocity = 350;    // Почему бы и нет?
oka.moveTo(35, 120);
```

Напомним, что текстовая строка в кавычках понимается в Java как объект класса String . Поэтому можно написать

```
int strlen = "Это объект класса String".length();
```

Объект "строка" выполняет метод length() , один из методов своего класса string , подсчитывающий число символов в строке. В результате получаем значение strlen , равное 24. Подобная странная запись встречается в программах на Java на каждом шагу.

Во многих ситуациях строят несколько моделей с разной степенью детализации. Скажем, для конструирования пальто и шубы нужна менее точная модель контуров человеческого тела и его движений, а для конструирования фрака или вечернего платья — уже гораздо более точная. При этом более точная модель, с меньшей степенью абстракции, будет использовать уже имеющиеся методы менее точной модели.

Не кажется ли вам, что класс Automobile сильно перегружен? Действительно, в мире выпущены миллионы автомобилей разных марок и видов. Что между ними общего, кроме четырех колес? Да и колес может быть больше или меньше. Не лучше ли написать отдельные классы для легковых и грузовых автомобилей, для гоночных автомобилей и вездеходов? Как организовать все это множество классов? На этот вопрос объектно-ориентированное программирование отвечает так: надо организовать иерархию классов.

Иерархия

Иерархия объектов давно используете для их классификации. Особенно детально она проработана в биологии. Все знакомы с семействами, родами и видами. Мы можем сделать описание своих домашних животных (pets): кошек (cats), собак (dogs), коров (cows) и прочих следующим образом:

```
class Pet{     // Здесь описываем общие свойства всех домашних любимцев
Master person; // Хозяин животного
int weight, age, eatTimel];              // Вес, возраст, время кормления
int eat(int food, int drink, int time){ // Процесс кормления
                                         // Начальные действия...
if (time == eatTimefi]) person.getFood(food, drink);
               // Метод потребления пищи
}
void voice();  // Звуки, издаваемые животным
               // Прочее...
}
```

Затем создаем классы, описывающие более конкретные объекты, связывая их с общим классом:

```
class Cat extends Pet{  // Описываются свойства, присущие только кошкам:
int mouseCatched;       // число пойманных мышей
void toMouse();         // процесс ловли мышей
                        // Прочие свойства
}
class Dog extends Pet{  // Свойства собак:
void preserve();        // охранять
}
```

Заметьте, что мы не повторяем общие свойства, описанные в классе Pet . Они наследуются автоматически. Мы можем определить объект класса Dog и использовать в нем все свойства класса Pet так, как будто они описаны в классе Dog :

```
Dog tuzik = new Dog(), sharik = new Dog();
```

После этого определения можно будет написать

```
tuzik.age = 3;
int p = sharik.eat (30, 10, 12);
А классификацию продолжить так:
class Pointer extends Dog{ ... } // Свойства породы Пойнтер
class Setter extends Dog{ ... }  // Свойства сеттеров
```

Заметьте, что на каждом следующем уровне иерархии в класс добавляются новые свойства, но ни одно свойство не пропадает. Поэтому и употребляется слово extends — "расширяет" и говорят, что класс Dog — *расширение* (extension) класса Pet . С другой стороны, количество объектов при этом уменьшается: собак меньше, чем всех домашних животных. Поэтому часто говорят, что класс Dog — *подкласс* (subclass) класса Pet , а класс Pet — *суперкласс* (superclass) или надкласс класса Dog .

Часто используют генеалогическую терминологию: родительский класс, дочерний класс, класс-потомок, класс-предок, возникают племянники и внуки, вся беспокойная семейка вступает в отношения, достойные мексиканского сериала.

В этой терминологии говорят о *наследовании* (inheritance) классов, в нашем примере класс Dog наследует класс Pet .

Мы еще не определили счастливого владельца нашего домашнего зоопарка. Опишем его в классе Master. Делаем набросок:

```
class Master{ // Хозяин животного
String name;  // Фамилия, имя
              // Другие сведения
void getFood(int food, int drink); // Кормление
              // Прочее
}
```

Хозяин и его домашние животные постоянно соприкасаются в жизни. Их взаимодействие выражается глаголами "гулять", "кормить", "охранять", "чистить", "ласкаться", "проситься" и прочими. Для описания взаимодействия объектов применяется третий принцип объектно-ориентированного программирования — обязанность или ответственность.

Ответственность

В нашем примере рассматривается только взаимодействие в процессе кормления, описываемое методом eat() . В этом методе животное обращается к хозяину, умоляя его применить метод getFood() .

В англоязычной литературе подобное обращение описывается словом message. Это понятие неудачно переведено на русский язык ни к чему не обязывающим словом *"сообщение".* Лучше было бы использовать слово "послание", "поручение" или даже "распоряжение". Но термин "сообщение" устоялся и нам придется его применять. Почему же не используется словосочетание "вызов метода", ведь говорят: "Вызов процедуры"? Потому что между этими понятиями есть, по крайней мере, триЪтличия.

- Сообщение идет к конкретному объекту, знающему метод решения задачи, в примере этот объект — текущее значение переменной person . У каждого объекта свое текущее состояние, свои значения полей класса, и это может повлиять на выполнение метода.
- Способ выполнения поручения, содержащегося в сообщении, зависит от объекта, которому оно послано. Один хозяин поставит миску с "Chappi", другой бросит кость, третий выгонит собаку на улицу. Это интересное свойство называется *полиморфизмом* (polymorphism) и будет обсуждаться ниже.
- Обращение к методу произойдет только на этапе выполнения программы, компилятор ничего не знает про метод. Это называется *"поздним связыванием" в* противовес *"раннему связыванию",* при котором процедура присоединяется к программе на этапе компоновки.

Итак, объект sharik , выполняя свой метод eat () , посылает сообщение объекту, ссылка на который содержится в переменной person, с просьбой выдать ему определенное количество еды и питья. Сообщение записано в строке person.getFood(food, drink) .

Этим сообщением заключается *контракт* (contract) между объектами, суть которого в том, что объект sharik берет на себя *ответственность* (responsibility) задать правильные параметры в сообщении, а объект — текущее значение person — возлагает на себя *ответственность* применить метод кормления getFood() , каким бы он ни был.

Для того чтобы правильно реализовать принцип ответственности, применяется четвертый принцип объектно-ориентированного программирования — *модульность* (modularity).

Модульность

Этот принцип утверждает — каждый класс должен составлять отдельный модуль. Члены класса, к которым не планируется обращение извне, должны быть инкапсулированы.

В языке Java инкапсуляция достигается добавлением модификатора private к описанию члена класса. Например:

```
private int mouseCatched;
private String name;
private void preserve();
```

Эти члены классов становятся *закрытыми,* ими могут пользоваться только экземпляры того же самого класса, например, tuzik может дать поручение

```
sharik.preserve().
```

А если в классе Master мы напишем

```
private void getFood(int food, int drink);
```

то метод getFood() не будет найден, и несчастный sharik не сможет получить пищу. ,

В противоположность закрытости мы можем объявить некоторые члены класса *открытыми,* записав вместо слова private модификатор public , например:

```
public void getFood(int food, int drink);
```

К таким членам может обратиться любой объект любого класса.

Знатокам C++

В языке Java словами private, public и protected отмечается каждый член класса в отдельности.

Принцип модульности предписывает открывать члены класса только в случае необходимости. Вспомните надпись: "Нормальное положение шлагбаума — закрытое".

Если же надо обратиться к полю класса, то рекомендуется включить в класс специальные *методы доступа* (access methods), отдельно для чтения этого поля (get method) и для записи в это поле (set method). Имена методов доступа рекомендуется начинать со слов get и set , добавляя к этим словам имя поля. Для JavaBeans эти рекомендации возведены в ранг закона.

В нашем примере класса Master методы доступа к полю Name в самом простом виде могут выглядеть так:

```
public String getName(){
return name;
}
public void setName(String newName)
{
name = newName;
}
```

В реальных ситуациях доступ ограничивается разными проверками, особенно в *set-методах,* меняющих значения полей. Можно проверять тип вводимого значения, задавать диапазон значений, сравнивать со списком допустимых значений.

Кроме методов доступа рекомендуется создавать проверочные *is-методы,* возвращающие логическое значение true или false . Например, в класс Master можно включить метод, проверяющий, задано ли имя хозяина:

```
public boolean isEmpty(){
return name == null ? true : false;
}
```

и использовать этот метод для проверки при доступе к полю Name , например:

```
if (masterOl.isEmpty()) masterOl.setName("Иванов");
```

Итак, мы оставляем открытыми только методы, необходимые для взаимодействия объектов. При этом удобно спланировать классы так, чтобы зависимость между ними была наименьшей, как принято говорить в теории ООП, было наименьшее *зацепление* (low coupling) между классами. Тогда структура программы сильно упрощается. Кроме того, такие классы удобно использовать как строительные блоки для построения других программ.

Напротив, члены класса должны активно взаимодействовать друг с другом, как говорят, иметь тесную функциональную *связность* (high cohestion). Для этого в класс следует включать все методы, описывающие поведение моделируемого объекта, и только такие методы, ничего лишнего. Одно из правил достижения сильной функциональной связности, введенное Карлом Ли-берхером (Karl J. Lieberherr), получило название *закон Деметра.* Закон гласит: "в методе т() класса А следует

использовать только методы класса A, методы классов, к которым принадлежат аргументы метода т(), и методы классов, экземпляры которых создаются внутри метода m ().

Объекты, построенные по этим правилам, подобны кораблям, снабженным всем необходимым. Они уходят в автономное плавание, готовые выполнить любое поручение, на которое рассчитана их конструкция.

Будут ли закрытые члены класса доступны его наследникам? Если в классе Pet написано

```
private Master person;
```

то можно ли использовать sharik.person ? Разумеется, нет. Ведь в противном случае каждый, интересующийся закрытыми полями класса A , может расширить его классом B , и просмотреть закрытые поля класса A через экземпляры класса B .

Когда надо разрешить доступ наследникам класса, но нежелательно открывать его всему миру, тогда в Java используется *защищенный* (protected) доступ, отмечаемый модификатором protected , например, объект sharik может обратиться к полю person родительского класса pet , если в классе Pet это поле описано так:

```
protected Master person;
```

Следует сразу сказать, что на доступ к члену класса влияет еще и пакет, в котором находится класс, но об этом поговорим в следующей главе.

Из этого общего схематического описания принципов объектно-ориентированного программирования видно, что язык Java позволяет легко воплощать все эти принципы. Вы уже поняли, как записать класс, его поля и методы, как инкапсулировать члены класса, как сделать расширение класса и какими принципами следует при этом пользоваться. Разберем теперь подробнее правила записи классов и рассмотрим дополнительные их возможности.

Как описать класс и подкласс

Итак, описание класса начинается со слова class, после которого записывается имя класса. Соглашения "Code Conventions" рекомендуют начинать имя класса с заглавной буквы.

Перед словом class можно записать модификаторы класса (class modifiers). Это одно из слов public, abstract, final, strictfp . Перед именем вложенного класса можно поставить, кроме того, модификаторы protected, private, static . Модификаторы мы будем вводить по мере изучения языка.

Тело класса, в котором в любом порядке перечисляются поля, методы, вложенные классы и интерфейсы, заключается в фигурные скобки.

При описании поля указывается его тип, затем, через пробел, имя и, может быть, начальное значение после знака равенства, которое можно записать константным выражением. Все это уже описано в *главе 1.*

Описание поля может начинаться с одного или нескольких необязательных модификаторов public, protected, private, static, final, transient, volatile . Если надо поставить несколько модификаторов, то перечислять их JLS рекомендует в указанном порядке, поскольку некоторые компиляторы требуют определенного порядка записи модификаторов. С модификаторами мы будем знакомиться по мере необходимости.

При описании метода указывается тип возвращаемого им значения или слово void , затем, через пробел, имя метода, потом, в скобках, список параметров. После этого в фигурных скобках расписывается выполняемый метод.

Описание метода может начинаться с модификаторов public, protected, private, abstract, static, final, synchronized, native, strictfp . Мы будем вводить их по необходимости.

В списке параметров через запятую перечисляются тип и имя каждого параметра. Перед типом какого-либо параметра может стоять модификатор final . Такой параметр нельзя изменять внутри метода. Список параметров может отсутствовать, но скобки сохраняются.

Перед началом работы метода для каждого параметра выделяется ячейка оперативной памяти, в которую копируется значение параметра, заданное при обращении к методу. Такой способ называется передачей параметров *по значению.*

В листинге 1.1 показано, как можно оформить метод деления пополам для нахождения корня нелинейного уравнения.

Листинг 2.1. Нахождение корня нелинейного уравнения методом бисекцйи

```
class Bisection2{
```

```
private static double final EPS = le-8; // Константа
private double a = 0.0, b = 1.5, root;  // Закрытые поля
public double getRoot(){return root;}   // Метод доступа
private double f(double x)
{
return x*x*x — 3*x*x + 3;   // Или что-то другое
}
private void bisect(){      // Параметров нет —
                            // метод работает с полями экземпляра
double y = 0.0;             // Локальная переменная — не поле
do{
root = 0.5 *(a + b); y = f(root);
if (Math.abs(y) < EPS) break;
// Корень найден. Выходим из цикла
// Если на концах отрезка [a; root]
// функция имеет разные знаки:
if (f(a) * y < 0.0} b = root;
      // значит, корень здесь
      // Переносим точку b в точку root
      //В противном случае:
else a = root;
      // переносим точку a в точку root
      // Продолжаем, пока [a; Ь] не станет мал
} while(Math.abs(b-a) >= EPS);
}
public static void main(String[] args){
Bisection2 b2 = new Bisection2();
b2.bisect();
System.out.println("x = " +
b2.getRoot() +    // Обращаемся к корню через метод доступа
", f() = " +b2.f(b2.getRoot()));
}
}
```

В описании метода f() сохранен старый, процедурный стиль: метод получает аргумент, обрабатывает его и возвращает результат. Описание метода bisect о выполнено в духе ООП: метод активен, он сам обращается к полям экземпляра b2 и сам заносит результат в нужное поле. Метод bisect () — это внутренний механизм класса Bisection2, поэтому он закрыт (private).

Имя метода, число и типы параметров образуют *сигнатуру* (signature) метода. Компилятор различает методы не по их именам, а по сигнатурам. Это позволяет записывать разные методы с одинаковыми именами, различающиеся числом и/или типами параметров.

Замечание

Тип возвращаемого значения не входит в сигнатуру метода, значит, методы не могут различаться только типом результата их работы.

Например, в классе Automobile мы записали метод moveTo(int x, int y) , обозначив пункт назначения его географическими координатами. Можно определить еще метод moveTo (string destination) для указания географического названия пункта назначения и обращаться к нему так:

```
oka.moveTo("Москва") ;
```

Такое дублирование методов называется *перегрузкой* (overloading). Перегрузка методов очень удобна в использовании. Можно выводить данные любого типа на экран методом printin() не заботясь о том, данные какого именно типа мы выводим. На самом деле мы использовали разные методы в одним и тем же именем printin , даже не задумываясь об этом. Конечно, все эти методы надо тщательно спланировать и заранее описать в классе. Это и сделано в классе Printstream, где представлено около двадцати методов print() и println() .

Если же записать метод с тем же именем в подклассе, например:

```
class Truck extends Automobile{
void moveTo(int x, int y){
   // Какие-то действия
}
   // Что-то еще
```

```
}
```

то он перекроет метод суперкласса. Определив экземпляр класса Truck , например:

```
Truck gazel = new Truck();
```

и записав gazei.moveTo(25, 150) , мы обратимся к методу класса Truck . Произойдет *переопределение* (overriding) метода.

При переопределении права доступа к методу можно только расширить. Открытый метод public должен остаться открытым, защищенный protected может стать открытым.

Можно ли внутри подкласса обратиться к методу суперкласса? Да, можно, если уточнить имя метода, словом super , например, super.moveTo(30, 40) . Можно уточнить и имя метода, записанного в этом же классе, словом this , например, this.moveTo (50, 70) , но в данном случае это уже излишне. Таким же образом можно уточнять и совпадающие имена полей, а не только методов.

Данные уточнения подобны тому, как мы говорим про себя "я", а не "Иван Петрович", и говорим "отец", а не "Петр Сидорович".

Переопределение методов приводит к интересным результатам. В классе Pet мы описали метод voice() . Переопределим его в подклассах и используем в классе chorus , как показано в листинге 1.2.

Листинг 1.2. Пример полиморфного метода

```
abstract class Pet{
   abstract void voice();
}
class Dog extends Pet{
   int k = 10;
   void voice(){
      System.out.printin("Gav-gav!");
   }
}
class Cat extends Pet{
   void voice () {
      System.out.printin("Miaou!");
   }
}
class Cow extends Pet{
   void voice(){
      System.out.printin("Mu-u-u!");
   }
}
public class Chorus(
   public static void main(String[] args){
      Pet[] singer = new Pet[3];
      singer[0] = new Dog();
      singer[1] = new Cat();
      singer[2] = new Cow();
      for (int i = 0; i < singer.length; i++)
         singer[i].voice();
   }
}
```

На рис. 1.1 показан вывод этой программы. Животные поют своими голосами!

Все дело здесь в определении поля singer[]. Хотя массив ссылок singer [] имеет тип Pet , каждый его элемент ссылается на объект своего типа Dog, Cat, cow . При выполнении программы вызывается метод конкретного объекта, а не метод класса, которым определялось имя ссылки. Так в Java реализуется полиморфизм.

Знатокам C++ В языке Java все методы являются виртуальными функциями.

В описании класса Pet появилось новое слово abstract . Класс Pet и метод voice() являются абстрактными.

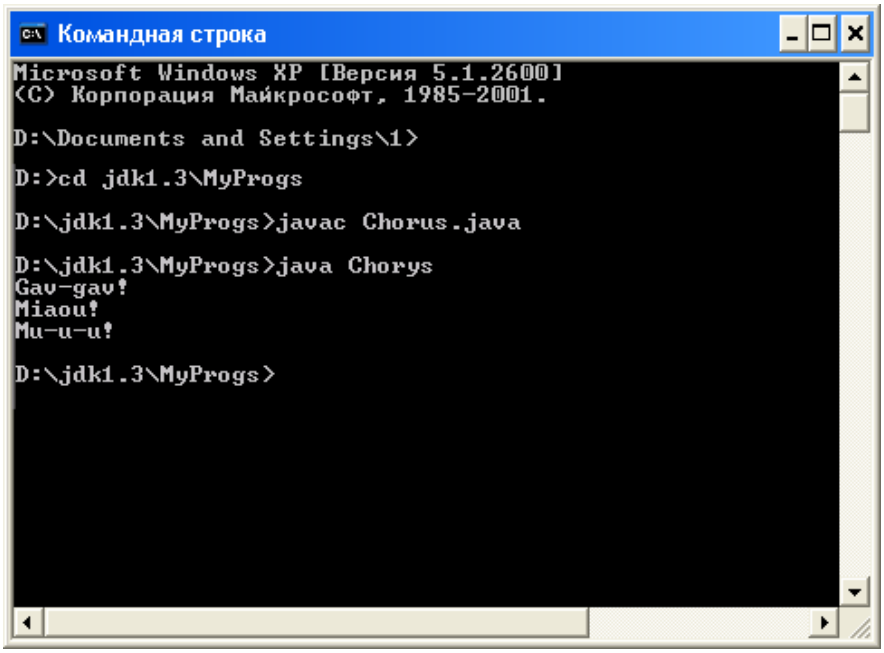

Рис. 1.1 Результат выполнения программы Chorus

Абстрактные методы и классы

При описании класса Pet мы не можем задать в методе voice () никакой полезный алгоритм, поскольку у всех животных совершенно разные голоса.

В таких случаях мы записываем только заголовок метода и ставим после закрывающей список параметров скобки точку с запятой. Этот метод будет *абстрактным* (abstract), что необходимо указать компилятору модификатором abstract .

Если класс содержит хоть один абстрактный метод, то создать его экземпляры, а тем более использовать их, не удастся. Такой класс становится *абстрактным,* что обязательно надо указать модификатором abstract .

Как же использовать абстрактные классы? Только порождая от них подклассы, в которых переопределены абстрактные методы.

Зачем же нужны абстрактные классы? Не лучше ли сразу написать нужные классы с полностью определенными методами, а не наследовать их от абстрактного класса? Для ответа снова обратимся к листингу 1.2.

Хотя элементы массива singer [] ссылаются на подклассы Dog, Cat, Cow , но все-таки это переменные типа Pet и ссылаться они могут только на поля и методы, описанные в суперклассе Pet . Дополнительные поля подкласса для них недоступны. Попробуйте обратиться, например, к полю k класса Dog , написав singer [0].k . Компилятор "скажет", что он не может реализовать такую ссылку. Поэтому метод, который реализуется в нескольких подклассах, приходится выносить в суперкласс, а если там его нельзя реализовать, то объявить абстрактным. Таким образом, абстрактные классы группируются на вершине иерархии классов.

Кстати, можно задать пустую реализацию метода, просто поставив пару фигурных скобок, ничего не написав между ними, например:

```
void voice(){}
```

Получится полноценный метод. Но это искусственное решение, запутывающее структуру класса.

Замкнуть же иерархию можно окончательными классами.

Окончательные члены и классы

Пометив метод модификатором final , можно запретить его переопределение в подклассах. Это удобно в целях безопасности. Вы можете быть уверены, что метод выполняет те действия, которые вы задали. Именно так определены математические функции sin(), cos() и прочие в классе Math . Мы уверены, что метод Math.cos (x) вычисляет именно косинус числа x . Разумеется, такой метод не может быть абстрактным.

Для полной безопасности, поля, обрабатываемые окончательными методами, следует сделать закрытыми (private).

Если же пометить модификатором final весь класс, то его вообще нельзя будет расширить. Так определен, например, класс Math :

```
public final class Math{ . . . }
```

Для переменных модификатор final имеет совершенно другой смысл. Если пометить модификатором final описание переменной, то ее значение (а оно должно быть обязательно задано или здесь же, или в блоке инициализации или в конструкторе) нельзя изменить ни в подклассах, ни в самом классе. Переменная превращается в константу. Именно так в языке Java определяются константы:

```
public final int MIN_VALUE = -1, MAX_VALUE = 9999;
```

По соглашению "Code Conventions" константы записываются прописными буквами, слова в них разделяются знаком подчеркивания.

На самой вершине иерархии классов Java стоит класс **Object** .

Класс Object

Если при описании класса мы не указываем никакое расширение, т. е. не пишем слово extends и имя класса за ним, как при описании класса Pet, то Java считает этот класс расширением класса object , и компилятор дописывает это за нас:

```
class Pet extends Object{ . . . }
```

Можно записать это расширение и явно.

Сам же класс object не является ничьим наследником, от него начинается иерархия любых классов Java. В частности, все массивы — прямые наследники класса object .

Поскольку такой класс может содержать только общие свойства всех классов, в него включено лишь несколько самых общих методов, например, метод equals() , сравнивающий данный объект на равенство с объектом, заданным в аргументе, и возвращающий логическое значение. Его можно использовать так:

```
Object obj1 = new Dog(), obj 2 = new Cat();
if (obj1.equals(obj2)) ...
```

Оцените объектно-ориентированный дух этой записи: объект obj1 активен, он сам сравнивает себя с другим объектом. Можно, конечно, записать и obj2.equals (obj1) , сделав активным объект obj2 , с тем же результатом.

Ссылки можно сравнивать на равенство и неравенство:

```
obj1 == obj2; obj1 != obj 2;
```

В этом случае сопоставляются адреса объектов, мы можем узнать, не указывают ли обе ссылки на один и тот же объект.

Метод equals() же сравнивает содержимое объектов в их текущем состоянии, фактически он реализован в классе object как тождество: объект равен только самому себе. Поэтому его часто переопределяют в подклассах, более того, правильно спроектированные, "хорошо воспитанные", классы должны переопределить методы класса object , если их не устраивает стандартная реализация.

Второй метод класса object , который следует переопределять в подклассах, — метод tostring() . Это метод без параметров, который пытается содержимое объекта преобразовать в строку символов и возвращает объект класса string .

К этому методу исполняющая система Java обращается каждый раз, когда требуется представить объект в виде строки, например, в методе printing .

Конструкторы класса

Вы уже обратили внимание на то, что в операции new, определяющей экземпляры класса, повторяется имя класса со скобками. Это похоже на обращение к методу, но что за "метод", имя которого полностью совпадает с именем класса?

Такой "метод" называется *конструктором класса* (class constructor). Его своет образие заключается не только в имени. Перечислим особенности конструктора.

- Конструктор имеется в любом классе. Даже если вы его не написали, компилятор Java сам создаст *конструктор по умолчанию* (default constructor), который, впрочем, пуст, он не делает ничего, кроме вызова конструктора суперкласса.
- Конструктор выполняется автоматически при создании экземпляра класса, после распределения памяти и обнуления полей, но до начала использования создаваемого объекта.
- Конструктор не возвращает никакого значения. Поэтому в его описании не пишется даже слово void , но можно задать один из трех модификаторов public , protected или private .
- Конструктор не является методом, он даже не считается членом класса. Поэтому его нельзя наследовать или переопределить в подклассе.
- Тело конструктора может начинаться:
- с вызова одного из конструкторов суперкласса, для этого записывается слово super() с параметрами в скобках, если они нужны;
- с вызова другого конструктора того же класса, для этого записывается слово this() с параметрами в скобках, если они нужны.

Если же super() в начале конструктора не указан, то вначале выполняется конструктор суперкласса без аргументов, затем происходит инициализация полей значениями, указанными при их объявлении, а уж потом то, что записано в конструкторе.

Во всем остальном конструктор можно считать обычным методом, в нем разрешается записывать любые операторы, даже оператор return , но только пустой, без всякого возвращаемого значения.

В классе может быть несколько конструкторов. Поскольку у них одно и то же имя, совпадающее с именем класса, то они должны отличаться типом и/или количеством параметров.

В наших примерах мы ни разу не рассматривали конструкторы классов, поэтому при создании экземпляров наших классов вызывался конструктор класса object .

Операция new

Пора подробнее описать операцию с одним операндом, обозначаемую словом new . Она применяется для выделения памяти массивам и объектам.

В первом случае в качестве операнда указывается тип элементов массива и количество его элементов в квадратных скобках, например:

```
double a[] = new double[100];
```

Во втором случае операндом служит конструктор класса. Если конструктора в классе нет, то вызывается конструктор по умолчанию.

Числовые поля класса получают нулевые значения, логические поля — значение false , ссылки — значение null .

Результатом операции new будет ссылка на созданный объект. Эта ссылка может быть присвоена переменной типа ссылка на данный тип:

```
Dog k9 = new Dog () ;
```

но может использоваться и непосредственно

```
new Dog().voice();
```

Здесь после создания безымянного объекта сразу выполняется его метод voice() . Такая странная запись встречается в программах, написанных на Java, на каждом шагу.

Статические члены класса

Разные экземпляры одного класса имеют совершенно независимые друг от друга поля-, принимающие разные значения. Изменение поля в одном экземпляре никак не влияет на то же поле в другом экземпляре. В каждом экземпляре для таких полей выделяется своя ячейка памяти. Поэтому такие поля называются переменными *экземпляра класса* (instance variables) или переменными *объекта.*

Иногда надо определить поле, общее для всего класса, изменение которого в одном экземпляре повлечет изменение того же поля во всех экземплярах. Например, мы хотим в классе Automobile отмечать порядковый заводской номер автомобиля. Такие поля называются *переменными класса* (class variables). Для переменных класса выделяется только одна ячейка

памяти, общая для всех экземпляров. Переменные класса образуются в Java модификатором static . В листинге 1.3 мы записываем этот модификатор при определении переменной number .

Листинг 1.3. Статическая переменная

```
class Automobile {
   private static int number;
   Automobile(){
      number++;
      System.out.println("From Automobile constructor:"+
                         " number = "+number);
   }
}
public class AutomobiieTest{
   public static void main(String[] args){
   Automobile lada2105    = new Automobile(),
              fordScorpio = new Automobile(),
              oka         = new Automobile!);
   }
}
```

Получаем результат, показанный на рис. 1.2.

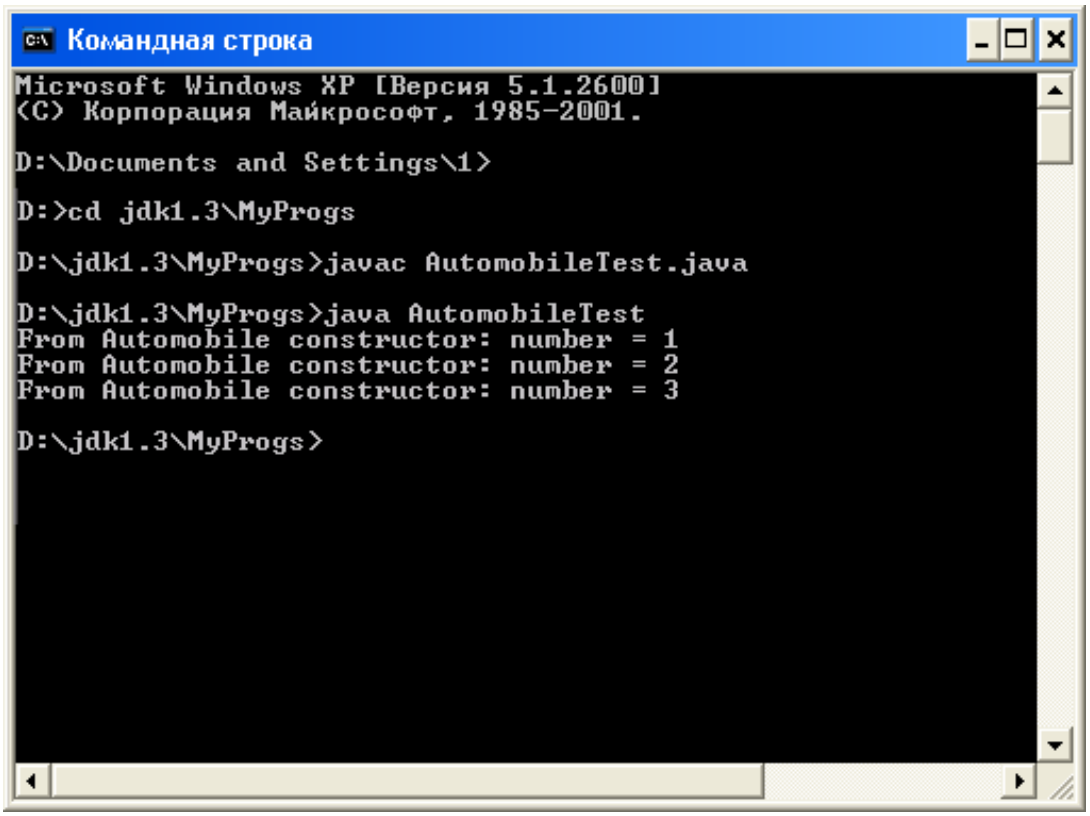

Рис. 1.2. Изменение статической переменной

Интересно, что к статическим переменным можно обращаться с именем класса, Automobile.number , а не только с именем экземпляра, lada2105.number , причем это можно делать, даже если не создан ни один экземпляр класса.

Для работы с такими *статическими переменными* обычно создаются *статические методы,* помеченные модификатором static . Для методов слово static имеет совсем другой смысл. Исполняющая система Java всегда создает в памяти только одну копию машинного кода метода, разделяемую всеми экземплярами, независимо от того, статический это метод или нет.

Основная особенность статических методов — они выполняются сразу во всех экземплярах класса. Более того, они могут выполняться, даже если не создан ни один экземпляр класса. Достаточно уточнить имя метода именем класса (а не именем объекта), чтобы метод мог работать. Именно так мы пользовались методами класса Math , не создавая его экземпляры, а просто

записывая Math.abs(x), Math.sqrt(x). Точно так же мы использовали метод System, out. println() . Да и методом main() мы пользуемся, вообще не создавая никаких объектов.

Поэтому статические методы называются *методами класса* (class methods), в отличие от нестатических методов, называемых *методами экземпляра* (instance methods).

Отсюда вытекают другие особенности статических методов:

- в статическом методе нельзя использовать ссылки this и super ;
- в статическом методе нельзя прямо, не создавая экземпляров, ссылаться на нестатические поля и методы;
- статические методы не могут быть абстрактными;
- статические методы переопределяются в подклассах только как статические.

Именно поэтому в листинге 1.5 мы пометили метод f() модификатором static . Но в листинге 2.1 мы работали с экземпляром b2 класса Bisection2 , и нам не потребовалось объявлять метод f() статическим.

Статические переменные инициализируются еще до начала работы конструктора, но при инициализации можно использовать только константные выражения. Если же инициализация требует сложных вычислений, например, циклов для задания значений элементам статических массивов или обращений к методам, то эти вычисления заключают в блок, помеченный словом static , который тоже будет выполнен до запуска конструктора:

```
static int[] a = new a[10];
static {
for(int k = 0; k < a.length; k++)
a[k] = k * k;
}
```

Операторы, заключенные в такой блок, выполняются только один раз, при первой загрузке класса, а не при создании каждого экземпляра.

Блоки статической инициализации, и блоки инициализации экземпляра записываются вне всяких методов и выполняются до начала выполнения не то что метода, но даже конструктора.

Класс Complex

Комплексные числа широко используются не только в математике. Они часто применяются в графических преобразованиях, в построении фракталов, не говоря уже о физике и технических дисциплинах.

Листинг 1.4 длинный, но просмотрите его внимательно, при обучении языку программирования очень полезно чтение программ на этом языке.

Листинг 1.4. Класс Complex

```
class Complex {
   private static final double EPS = le-12; // Точность вычислений
   private double re, im;                   // Действительная и мнимая
часть
                                            // Четыре конструктора
   Complex(double re, double im) {
      this, re = re; this.im = im;
   }
   Complex(double re){this(re, 0.0); }
   Complex(){this(0.0, 0.0); }
   Complex(Complex z){this(z.re, z.im) ; }
      // Методы доступа
   public double getRe(){return re;}
   public double getlmf){return im;}
   public Complex getZ(){return new Complex(re, im);}
   public void setRe(double re){this.re = re;}
   public void setlm(double im){this.im = im;}
   public void setZ(Complex z){re = z.re; im = z.im;}
      // Модуль и аргумент комплексного числа
   public double mod(){return Math.sqrt(re * re + im * im);}
   public double arg()(return Math.atan2(re, im);}
      // Проверка: действительное число?
```

```
   public boolean isReal(){return Math.abs(im) < EPS;}
   public void pr(){    // Вывод на экран
      System.out.println(re + (im < 0.0 ? "" : '"+") + im + "i");
   }
      // Переопределение методов класса Object
   public boolean equals(Complex z){
      return Math.abs(re -'z.re) < EPS &&
             Math.abs(im - z.im) < EPS;
   }
   public String toString(){
      return "Complex: " + re + " " + im;
   }
      // Методы, реализующие операции +=, -=, *=, /=
   public void add(Complex z){re += z.re; im += z.im;}
   public void sub(Complex z){re -= z.re; im —= z.im;}
   public void mul(Complex z){
      double t = re * z.re — im * z. im;
            im = re * z.im + im * z.re;
            re = t;
   }
   public void div(Complex z){
      double m = z.mod();
      double t = re * z.re — im * z.im;
      im = (im * z.re — re * z.im) / m;
      re = t / m;
   }
      // Методы, реализующие операции +, -, *, /
   public Complex plus(Complex z){
      return new Complex(re + z.re, im + z im);
   }
   public Complex minus(Complex z){
      return new Complex(re - z.re, im - z.im);
   }
   public Complex asterisk(Complex z){
      return new Complex(
         re * z.re - im * z.im, re * z.im + im * z re);
   }
public Complex slash(Complex z){
   double m = z.mod();
   return new Complex(
      (re * z.re - im * z.im) / m, (im * z.re - re * z.im) / m);
   }
}
   // Проверим работу класса Complex
public class ComplexTest{
   public static void main(Stringf] args){
      Complex zl = new Complex(),
              z2 = new Complex(1.5),
              z3 = new Complex(3.6, -2.2),
              z4 = new Complex(z3);
      System.out.printlnf);       // Оставляем пустую строку
      System.out.print("zl = "); zl.pr();
      System.out.print("z2 = "); z2.pr();
      System.out.print("z3 = "); z3.pr();
      System.out.print ("z4 = "}; z4.pr();
      System.out.println(z4);     // Работает метод toString()
      z2.add(z3);
      System.out.print("z2 + z3 = "}; z2.pr();
      z2.div(z3);
      System.out.print("z2 / z3 = "); z2.pr();
      z2 = z2.plus(z2);
      System.out.print("z2 + z2 = "); z2.pr();
```

```
        z3 = z2.slash(z1);
        System.out.print("z2 / z1 = "); z3.pr();
    }
}
```

На рис. 1.3 показан вывод этой программы.

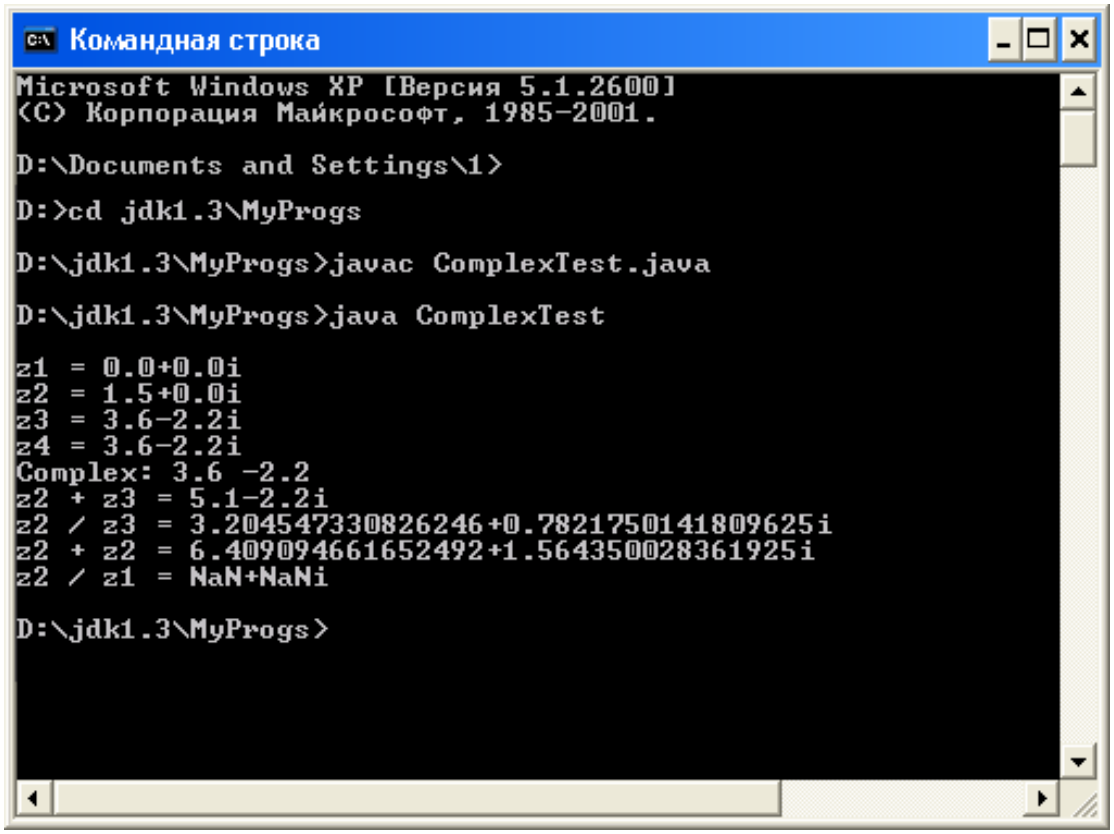

Рис. 1.3. Вывод программы ComplexTest

Метод main()

Всякая программа, оформленная как *приложение* (application), должна содержать метод с именем main . Он может быть один на все приложение или содержаться в некоторых классах этого приложения, а может находиться и в каждом классе.

Метод main() записывается как обычный метод, может содержать любые описания и действия, но он обязательно должен быть открытым (public), статическим (static), не иметь возвращаемого значения (void). Его аргументом обязательно должен быть массив строк (string[]). По традиции этот массив называют args , хотя имя может быть любым.

Эти особенности возникают из-за того, что метод main() вызывается автоматически исполняющей системой Java в самом начале выполнения приложения. При вызове интерпретатора java указывается класс, где записан метод main() , с которого надо начать выполнение. Поскольку классов с методом main() может быть несколько, можно построить приложение с дополнительными точками входа, начиная выполнение приложения в разных ситуациях из различных классов.

Часто метод main() заносят в каждый класс с целью отладки. В этом случае в метод main() включают тесты для проверки работы всех методов класса.

При вызове интерпретатора java можно передать в метод main() несколько параметров, которые интерпретатор заносит в массив строк. Эти параметры перечисляются в строке вызова java через пробел сразу после имени класса. Если же параметр содержит пробелы, надо заключить его в кавычки. Кавычки не будут включены в параметр, это только ограничители.

Все это легко понять на примере листинга 1.5, в котором записана программа, просто выводящая параметры, передаваемые в метод main() при запуске.

Листинг 1.5. Передача параметров в метод main()

```
class Echo {
   public static void main(String[] args){
      for (int i = 0; i < args.length; i++)
         System.out.println("args[" + i +"]="+ args[i]);
   }
}
```

На рис. 1.4 показаны результаты работы этой программы с разными вариантами задания параметров.

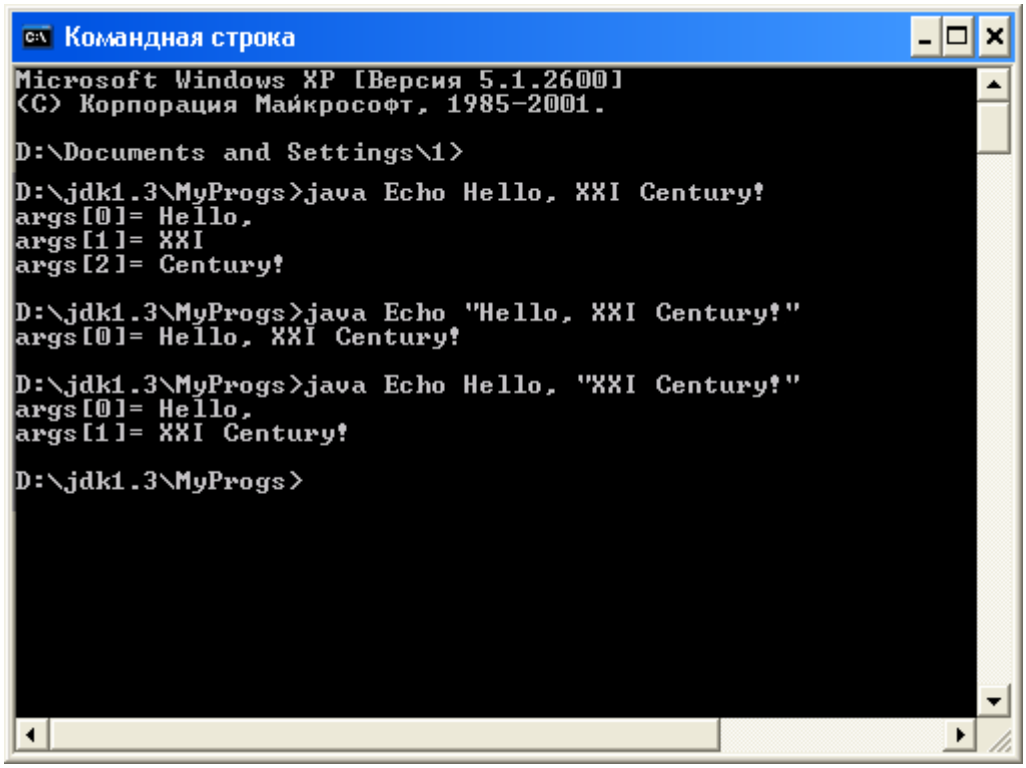

Рис. 1.4. Вывод параметров командной строки

Как видите, имя класса не входит в число параметров. Оно и так известно в методе main() .

Знатокам C/C++

Поскольку в Java имя файла всегда совпадает с именем класса, содержащего метод main() , оно не заносится в args[0] . Вместо argc используется args.length. Доступ к переменным среды разрешен не всегда и осуществляется другим способом. Некоторые значения можно просмотреть так: System.getProperties().list(System.out);.

Где видны переменные

В языке Java нестатические переменные можно объявлять в любом месте кода между операторами. Статические переменные могут быть только полями класса, а значит, не могут объявляться внутри методов и блоков. Какова же *область видимости* (scope) переменных? Из каких методов мы можем обратиться к той или иной переменной? В каких операторах использовать? Рассмотрим на примере листинга 1.6 разные случаи объявления переменных.

Листинг 1.6. Видимость и инициализация переменных

```
class ManyVariables{
  static int x = 9, y;  // Статические переменные — поля класса
                        // Они известны во всех методах и блоках класса
                        // Переменная y получает значение 0
  static{   // Блок инициализации статических переменных
            // Выполняется один раз при первой загрузке класса после
            // инициализаций в объявлениях переменных
x = 99;     // Оператор выполняется вне всякого метода!
```

```
}
int a = 1, p;  // Нестатические переменные — поля экземпляра
               // Известны во всех методах и блоках класса, в которых они
               //не перекрыты другими переменными с тем же именем
               // Переменная p получает значение 0
{          // Блок инициализации экземпляра
           // Выполняется при создании, каждого экземпляра после
           // инициализаций при объявлениях переменных
p = 999;   // Оператор выполняется вне всякого метода!
}
static void f(int b){   // Параметр метода b — локальная
                        // переменная, известна только внутри метода
int a = 2;    // Это вторая переменная с тем же именем "a"
              // Она известна только внутри метода f() и
              // здесь перекрывает первую "a"
int c;        // Локальная переменная, известна только в методе f()
              //Не получает никакого начального значения
              //и должна быть определена перед применением
{ int c = 555; // Ошибка! Попытка повторного объявления
int x = 333;   // Локальная переменная, известна только в этом блоке
}
// Здесь переменная x уже неизвестна
for (int d = 0; d < 10; d++){
  // Переменная цикла d известна только в цикле
  int a = 4;   // Ошибка!
  int e = 5;   // Локальная переменная, известна только в цикле for
  e++;         // Инициализируется при каждом выполнении цикла
  System.out.println("e = " + e) ; // Выводится всегда "e = 6"
}
// Здесь переменные d и e неизвестны
}
public static void main(String!] args){
  int a = 9999; // Локальная переменная, известна
                // только внутри метода main()
  f (a);
  }
}
```

Обратите внимание на то, что переменным класса и экземпляра неявно присваиваются нулевые значения. Символы неявно получают значение '\u0000' , логические переменные — значение false , ссылки получают неявно значение null .

Локальные же переменные неявно не инициализируются. Им должны либо явно присваиваться значения, либо они обязаны определяться до первого использования. К счастью, компилятор замечает неопределенные локальные переменные и сообщает о них.

Внимание

Поля класса при объявлении обнуляются, локальные переменные автоматически не инициализируются.

В листинге 1.6 появилась еще одна новая конструкция: *блок инициализации экземпляра* (instance initialization). Это просто блок операторов в фигурных скобках, но записывается он вне всякого метода, прямо в теле класса. Этот блок выполняется при создании каждого экземпляра, после инициализации при объявлении переменных, но до выполнения конструктора. Он играет такую же роль, как и static-блок для статических переменных. Зачем же он нужен, ведь все его содержимое можно написать в начале конструктора? В тех случаях, когда конструктор написать нельзя, а именно, в безымянных внутренних классах.

Вложенные классы

В этой главе уже несколько раз упоминалось, что в теле класса можно сделать описание другого, *вложенного* (nested) класса. А во вложенном классе можно снова описать вложенный, *внутренний* (inner) класс и т. д. Эта матрешка кажется вполне естественной, но вы уже поднаторели в написании классов, и у вас возникает масса вопросов.

- Можем ли мы из вложенного класса обратиться к членам внешнего класса? Можем, для того это все и задумывалось.
- А можем ли мы в таком случае определить экземпляр вложенного класса, не определяя экземпляры внешнего класса? Нет, не можем, сначала надо определить хоть один экземпляр внешнего класса, матрешка ведь!
- А если экземпляров внешнего класса несколько, как узнать, с каким экземпляром внешнего класса работает данный экземпляр вложенного класса? Имя экземпляра вложенного класса уточняется именем связанного с ним экземпляра внешнего класса. Более того, при создании вложенного экземпляра операция new тоже уточняется именем внешнего экземпляра.
- А...?

Хватит вопросов, давайте разберем все по порядку.

Все вложенные классы можно разделить на вложенные *классы-члены* класса (member classes), описанные вне методов, и вложенные *локальные классы* (local classes), описанные внутри методов и/или блоков. Локальные классы, как и все локальные переменные, не являются членами класса.

Классы-члены могут быть объявлены статическим модификатором static. Поведение статических классов-членов ничем не отличается от поведения обычных классов, отличается только обращение к таким классам. Поэтому они называются *вложенными классами верхнего уровня* (nestee tep-level classes), хотя статические классы-члены можно вкладывать друг в друга. В них можно объявлять статические члены. Используются они обычно для того, чтобы сгруппировать вспомогательные классы вместе с основным классом.

Все нестатические вложенные классы называются *внутренними* (inner). В них нельзя объявлять статические члены.

Локальные классы, как и все локальные переменные, известны только в блоке, в котором они определены. Они могут быть *безымянными* (anonymous classes).

В листинге 1.7 рассмотрены все эти случаи.

Листинг 1.7. Вложенные классы

```
class Nested{
static private int pr;   // Переменная pr объявлена статической
                                 // чтобы к ней был доступ из статических
классов A и AB
String s = "Member of Nested";
// Вкладываем статический класс.
static class .A{      // Полное имя этого класса — Nested.A
  private int a=pr;
  String s = "Member of A";
    // Во вложенньм класс A вкладываем еще один статический класс
  static class AB{    // Полное имя класса — Nested.A.AB
    private int ab=pr;
    String s = "Member of AB";
  }
}
//В класс Nested вкладываем нестатический класс
class B{     // Полное имя этого класса — Nested.B
  private int b=pr;
  String s = "Member of B";
  // В класс B вкладываем еще один класс
  class BC{  // Полное имя класса — Nested.B.BC
    private int bc=pr;
    String s = "Member of BC";
  }
  void f(final int i){    // Без слова final переменные i и j
    final int j = 99;     // нельзя использовать в локальном классе D
    class D{              // Локальный класс D известен только внутри f()
      private int d=pr;
      String s = "Member of D";
      void pr(){
```

```
                // Обратите внимание на то, как различаются
                // переменные с одним и тем же именем "s"
            System.out.println(s + (i+j));   // "s" эквивалентно "this.s"
            System.out.println(B.this.s);
            System.out.println(Nested.this.s);
//          System.out.println(AB.this.s);    // Нет доступа
//          System.out.println(A.this.s);     // Нет доступа
          }
        }
        D d = new D();    // Объект определяется тут же, в методе f()
        d.pr();           // Объект известен только в методе f()
      }
    }
    void m(){
      new Object(){    // Создается объект безымянного класса,
                       // указывается конструктор его суперкласса
        private int e = pr;
        void g(){
          System.out.println("From g()) ;
        }
      }.g();   // Тут же выполняется метод только что созданного объекта
      }
    }
    public class NestedClasses{
      public static void main(String[] args){
        Nested nest = new Nested();   // Последовательно раскрываются
                                      // три матрешки
        Nested.A theA = nest.new A(); // Полное имя класса и уточненная
                                      // операция new. Но конструктор только
    вложенного класса
        Nested.A.AB theAB = theA.new AB(); // Те же правила. Операция
                                           // new уточняется только одним
    именем
        Nested.B theB = nest.new B(); // Еще одна матрешка
        Nested.B.BC theBC = theB.new BC();
        theB.f(999);  // Методы вызываются обычным образом
        nest.m();
      }
    }
```

Дадим пояснения.

- Как видите, доступ к полям внешнего класса Nested возможен отовсюду, даже к закрытому полю pr . Именно для этого в Java и введены вложенные классы. Остальные конструкции введены вынужденно, для того чтобы увязать концы с концами.
- Язык Java позволяет использовать одни и те же имена в разных областях видимости — пришлось уточнять константу this именем класса: Nested.this, B.this .
- В безымянном классе не может быть конструктора, ведь имя конструктора должно совпадать с именем класса, — пришлось использовать имя суперкласса, в примере это класс object . Вместо конструктора в безымянном классе используется блок инициализации экземпляра.
- Нельзя создать экземпляр вложенного класса, не создав предварительно экземпляр внешнего класса, — пришлось подстраховать это правило уточнением операции new именем экземпляра внешнего класса— nest.new , theA.new, theB.new .
- При определении экземпляра указывается полное имя вложенного класса, но в операции new записывается просто конструктор класса.

Введение вложенных классов сильно усложнило синтаксис и поставило много задач разработчикам языка.

- Можно ли наследовать вложенные классы? Можно.

- Как из подкласса обратиться к методу суперкласса? Константа super уточняется именем соответствующего суперкласса, подобно константе this .
- А могут ли вложенные классы быть расширениями других классов? Могут.

Механизм вложенных классов станет понятнее, если посмотреть, какие файлы с байт-кодами создал компилятор:

- Nested$l$D.class — локальный класс о, вложенный в класс Nested ;
- NestedSl.class — безымянный класс;
- Nested$A$AB.class — класс Nested.A.AB ;
- Nested$A.class — класс Nested.A ;
- Nested$B$BC.class — класс Nested.в.вс ;
- NestedSB.class — класс Nested.в ;
- Nested.class — внешний класс Nested ;
- NestedClasses.class - класс с методом main () .

Компилятор разложил матрешки и, как всегда, создал отдельные файлы для каждого класса. При этом, поскольку в идентификаторах недопустимы точки, компилятор заменил их знаками доллара. Для безымянного класса компилятор придумал имя. Локальный класс компилятор пометил номером.

Оказывается, вложенные классы существуют только на уровне исходного кода. Виртуальная машина Java ничего не знает о вложенных классах. Она работает с обычными внешними классами. Для взаимодействия объектов вложенных классов компилятор вставляет в них специальные закрытые поля. Поэтому в локальных классах можно использовать только константы объемлющего метода, т. е. переменные, помеченные словом final . Виртуальная машина просто не догадается передавать изменяющиеся значения переменных в локальный класс. Таким образом не имеет смысла помечать вложенные классы private , все равно они выходят на самый внешний уровень.

Отношения "быть частью" и "являться"

Теперь у нас появились две различные иерархии классов. Одну иерархию образует наследование классов, другую — вложенность классов.

Определив, какие классы будут написаны в вашей программе, и сколько их будет, подумайте, как спроектировать взаимодействие классов? Вырастить пышное генеалогическое дерево классов-наследников или расписать матрешку вложенных классов?

Теория ООП советует прежде всего выяснить, в каком отношении находятся ваши классы P и Q — в отношении "класс Q является экземпляром класса P" ("a class Q is a class p") или в отношении "класс Q — часть класса P" ("a class Q has a class P").

Например: "Собака является животным" или "Собака — часть животного"? Ясно, что верно первое отношение "is-a", поэтому мы и определили класс Dog как расширение класса Pet.

Отношение "is-a" — это отношение "обобщение-детализация", отношение большей или меньшей абстракции, и ему соответствует наследование классов.

Отношение "has-a" — это отношение "целое-часть", ему соответствует вложение.

Полный пример реализации класса

```
/*
Класс Complex
Комплексные числа широко используются не только в математике.
Они часто применяются в графических преобразованиях,
в построении фракталов, не говоря уже о физике и технических
дисциплинах.
*/
class Complex {
   private static final double EPS = 1e-12; // Точность
вычислений
```

```
   private double re, im;                    // Действительная и
мнимая часть

   // Четыре конструктора  ******************************
   Complex(double re, double im) {
      this.re = re; this.im = im;
   };

   Complex(double re) {
      this(re, 0.0);
   };

   Complex() {
      this(0.0, 0.0);
   };

   Complex(Complex z) {
      this(z.re, z.im);
   };

   // Методы доступа  ***********************************
   public double getRe() { return re;}
   public double getIm() { return im;}
   public Complex getZ() { return new Complex(re, im); }
   public void setRe(double re) { this.re = re; }
   public void setIm(double im) { this.im = im; }
   public void setZ (Complex z) { re = z.re; im = z.im; }

   // Модуль и аргумент комплексного числа  **************
   public double mod() { return Math.sqrt(re * re + im * im); }
   public double arg() { return Math.atan2(re, im); }

   // Проверка: действительное число?  ***********************
   public boolean isReal() { return Math.abs(im) < EPS; }

   // Вывод на экран
*****************************************
   public void pr() {
      System.out.println(re + ( im < 0.0 ? "" : "+" ) + im +
"i");
   }

   // Переопределение методов класса Object
**********************
   public boolean equals(Complex z) {
      return Math.abs(re - z.re) < EPS &&
             Math.abs(im - z.im) < EPS;
   }
   public String toString() {
      return "Complex: " + re + " " + im;
   }
```

```
   // Методы, реализующие операции +=, -=, *=, /=
   public void add(Complex z) {re += z.re; im += z.im; }
   public void sub(Complex z) {re -= z.re; im -= z.im; }
   public void mul(Complex z) {
      double t = re * z.re - im * z.im;
            im = re * z.im + im * z.re;
            re = t;
   }
   public void div(Complex z) {
      double m = z.mod();
      double t = re * z.re - im * z.im;
      im = (im * z.re - re * z.im) / m;
      re = t / m;
   }

   // Методы, реализующие операции +, -, *, /
   public Complex plus(Complex z) {
      return new Complex(re + z.re, im + z.im);
   }
   public Complex minus(Complex z) {
      return new Complex(re - z.re, im - z.im);
   }
   public Complex asterisk(Complex z) {
      return new Complex(
          re * z.re - im * z.im,
          re * z.im + im * z.re );
   }
   public Complex slash(Complex z) {
      double m = z.mod();
      return new Complex(
       (re * z.re - im * z.im) / m,
       (im * z.re - re * z.im) / m );
   }
}


   // Проверим работу класса Complex
public class ComplexTest{
      public static void main(String[] args) {
         Complex z1 = new Complex(),
                 z2 = new Complex(1.5),
                 z3 = new Complex(3.6, -2.2),
                 z4 = new Complex(z3);
         System.out.println();       // Оставляем пустую строку
         System.out.print("z1 = "); z1.pr();
         System.out.print("z2 = "); z2.pr();
         System.out.print("z3 = "); z3.pr();
         System.out.print("z4 = "); z4.pr();
         System.out.println(z4);     // Работает метод
toString()
```

```
            z2.add(z3);
              System.out.print("z2 + z3 = "); z2.pr();
            z2.div(z3);
              System.out.print("z2 / z3 = "); z2.pr();
            z2 = z2.plus(z2);
              System.out.print("z2 + z2 = "); z2.pr();
            z3 = z2.slash(z1);
              System.out.print("z2 / z1 = "); z3.pr();
        }
}
```