

В. Монахов

**Язык
программирования
Java и среда
NetBeans**

2-е издание

Санкт-Петербург

«БХВ-Петербург»

2009

УДК 681.3.068
ББК 32.973.26-018.1
М77

Монахов В. В.

М77 Язык программирования Java и среда NetBeans. — 2-е изд., перераб. и доп. — СПб.: БХВ-Петербург, 2009. — 720 с.: ил. + CD-ROM

ISBN 978-5-9775-0424-9

Книга написана на базе курса лекций, читаемых автором на кафедре вычислительной физики Санкт-Петербургского государственного университета. Второе издание ориентировано на использование среды NetBeans 6.5 и дополнено новыми материалами. Изложены основные синтаксические конструкции Java, принципы объектно-ориентированного программирования, особенности проведения численных расчетов. Приводятся сведения о разработке основных категорий программного обеспечения Java (ME, SE и EE). Рассказывается о создании сетевых приложений и приложений для мобильных устройств. Разбираются методики написания многопоточных приложений Java для систем с многоядерными процессорами. Материал сопровождается большим количеством примеров с подробным анализом их исходных кодов. На компакт-диске находятся дистрибутивы JDK 6u11 и NetBeans 6.5 для Windows и Linux, а также исходные тексты примеров.

Для начинающих Java-программистов и студентов

УДК 681.3.068
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	<i>Екатерина Кондукова</i>
Зам. главного редактора	<i>Игорь Шишигин</i>
Зав. редакцией	<i>Григорий Добин</i>
Редактор	<i>Леонид Кочин</i>
Компьютерная верстка	<i>Ольги Сергиенко</i>
Корректор	<i>Зинаида Дмитриева</i>
Оформление обложки	<i>Елены Беляевой</i>
Зав. производством	<i>Николай Тверских</i>

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 27.02.09.

Формат 70×100^{1/16}. Печать офсетная. Усл. печ. л. 58,05.

Тираж 2000 экз. Заказ №

"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

Санитарно-эпидемиологическое заключение на продукцию № 77.99.60.953.Д.003650.04.08 от 14.04.2008 г. выдано Федеральной службой по надзору в сфере защиты прав потребителей и благополучия человека.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 978-5-9775-0424-9

© Монахов В. В., 2009
© Оформление, издательство "БХВ-Петербург", 2009

Оглавление

Введение	11
Глава 1. Общие представления о языке Java.....	15
1.1. Java и другие языки программирования. Системное и прикладное программирование.....	15
1.2. Виртуальная Java-машина, байт-код, JIT-компиляция. Категории программ, написанных на языке Java.....	20
1.3. Алфавит языка Java. Десятичные и шестнадцатеричные цифры и целые числа. Резервированные слова	26
Алфавит языка Java	27
Десятичные и шестнадцатеричные цифры и целые числа.....	27
Резервированные слова и литералы языка Java.....	29
1.4. Управляющие последовательности. Символы Unicode. Специальные символы	30
Управляющие последовательности.....	30
Управляющие последовательности — символы формирования текста	30
Управляющие последовательности — символы Unicode	30
Простые специальные символы	31
Составные специальные символы.....	32
1.5. Идентификаторы. Переменные и типы. Примитивные и ссылочные типы.....	33
1.6. Процедурное и объектно-ориентированное программирование. Инкапсуляция	37
1.7. Работа со ссылочными переменными. Сборка мусора.....	41
1.8. Проекты. Пакеты. Уровни видимости классов. Импорт классов	46
1.9. Базовые пакеты и классы Java	50
1.10. Технологии Java, .NET, ASP, PHP	53
1.11. Среды разработки NetBeans, Eclipse, JDeveloper, JBuilder, IntelliJ IDEA	58
NetBeans 6.5	59
Eclipse 3.4.1 (Ganymede)	61
JDeveloper 11	63
JBuilder 2008	66
IntelliJ IDEA 7.0.4	68
Общий итог сравнения сред разработки.....	68
Краткие итоги по главе.....	69
Типичные ошибки.....	72
Задания.....	72

Глава 2. Среда NetBeans 6.5	73
2.1. Установка JDK 6	73
2.2. Установка среды NetBeans 6.5	77
2.3. Создание в NetBeans простейшего приложения Java	85
2.4. Компиляция файлов проекта и запуск приложения. Jar-файлы	91
2.5. Структура проекта NetBeans	94
2.6. Создание в NetBeans приложения Java с графическим интерфейсом	98
2.7. Редактор экранных форм	103
2.8. Свойства компонентов	109
2.9. Внешний вид приложения — технология Look and Feel	115
2.10. Ведение проектов	118
2.11. Редактирование меню экранной формы	120
2.12. Создание нового класса	129
2.13. Документирование исходного кода в Java	130
2.14. Основные компоненты пакетов <i>swing</i> и <i>awt</i>	139
2.15. Менеджеры размещения (<i>Layout</i>) и якоря (<i>Anchor</i>)	144
2.16. Создание приложения Desktop Application	145
Краткие итоги по главе	147
Типичные ошибки	148
Задания	148
Глава 3. Примитивные типы данных и операторы для работы с ними	151
3.1. Булевый (логический) тип	151
3.2. Целые типы, переменные, константы	152
3.3. Основные операторы для работы с целочисленными величинами	156
3.4. Вещественные типы и класс <i>Math</i>	157
3.5. Правила явного и автоматического преобразования типа при работе с числовыми величинами	161
3.6. Оболочечные классы. Упаковка (<i>boxing</i>) и распаковка (<i>unboxing</i>)	163
3.7. Приоритет операторов	165
Краткие итоги по главе	166
Типичные ошибки	168
Задания	168
Глава 4. Работа с числами в языке Java	171
4.1. Двоичное представление целых чисел	171
Позиционные и непозиционные системы счисления	171
Двоичное представление положительных целых чисел	173
Двоичное представление отрицательных целых чисел. Дополнительный код	175
Сложение положительного и отрицательного чисел	175
Сложение отрицательных чисел	176
Проблемы целочисленной машинной арифметики.	
Целочисленное переполнение	176
Сложение положительных чисел	176
Умножение положительных чисел	177

Шестнадцатеричное представление целых чисел и перевод из одной системы счисления в другую	179
Преобразование чисел из системы с меньшим основанием в систему с большим основанием	180
Преобразование чисел из системы с большим основанием в систему с меньшим основанием	181
Преобразование чисел в системах счисления с кратными основаниями	181
4.2. Побитовые маски и сдвиги	182
4.3. Двоичное представление вещественных чисел	185
Двоичные дроби	185
Мантисса и порядок числа	185
Научная нотация записи вещественных чисел	187
Стандарт IEEE 754 представления чисел в формате с плавающей точкой	188
Краткие итоги по главе	193
Типичные ошибки	194
Задания	194
Глава 5. Управляющие конструкции	197
5.1. Составной оператор	197
5.2. Условный оператор <i>if</i>	198
5.3. Оператор выбора <i>switch</i>	202
5.4. Условное выражение <i>...?... :</i>	203
5.5. Операторы инкремента <i>++</i> и декремента <i>--</i>	204
5.6. Оператор цикла <i>for</i>	204
5.7. Ошибки при использовании вещественного счетчика цикла	206
5.8. Эффективная организация циклов при вычислениях в формате с плавающей точкой	209
5.9. Особенности целочисленных вычислений — организация циклов, приоритет операторов и арифметическое переполнение	214
5.10. Оператор цикла <i>while</i> — цикл с предусловием	219
5.11. Оператор цикла <i>do...while</i> — цикл с постусловием	221
5.12. Операторы прерывания <i>continue</i> , <i>break</i> , <i>return</i> , <i>System.exit</i>	221
Краткие итоги по главе	225
Типичные ошибки	226
Задания	226
Глава 6. Начальные сведения об объектном программировании	227
6.1. Наследование и полиморфизм	227
6.2. Функции. Модификаторы. Передача примитивных типов в функции	236
6.3. Локальные и глобальные переменные. Модификаторы доступа и правила видимости. Ссылка <i>this</i>	239
6.4. Передача ссылочных типов в функции. Проблема изменения ссылки внутри подпрограммы	242
6.5. Наследование. Суперклассы и подклассы. Переопределение методов	249
6.6. Наследование и правила видимости. Зарезервированное слово <i>super</i>	255

6.7. Статическое и динамическое связывание методов. Полиморфизм	257
6.8. Базовый класс <i>Object</i>	259
6.9. Конструкторы. Резервированные слова <i>super</i> и <i>this</i> . Блоки инициализации.....	261
6.10. Удаление неиспользуемых объектов и метод <i>finalize</i> . Проблема деструкторов для сложно устроенных объектов.....	264
6.11. Перегрузка методов	265
6.12. Правила совместимости ссылочных типов как основа использования полиморфного кода. Приведение и проверка типов	268
Краткие итоги по главе.....	271
Типичные ошибки.....	273
Задания.....	274

Глава 7. UML-диаграммы. Прямое и обратное проектирование.

Рефакторинг.....	275
7.1. UML-диаграммы	275
7.2. Структура окон проектов с UML-диаграммами.....	284
7.3. Панель инструментов проектов с UML-диаграммами.....	287
7.4. Прямое проектирование (Forward engineering) — построение кода классов по UML-диаграммам	291
7.5. Обратное проектирование (Reverse engineering) — построение UML-диаграмм по разработанным классам	298
7.6. Рефакторинг	304
Первый случай — переименование элементов программы	306
Второй случай — перемещение класса	308
Третий случай — копирование класса для его дальнейшей модификации	308
Четвертый случай — безопасное удаление элемента программы.....	309
Пятый случай — изменение списка параметров метода	309
Шестой случай — инкапсуляция полей данных	310
Краткие итоги по главе.....	311
Типичные ошибки.....	311
Задания.....	312

Глава 8. Важнейшие объектные типы..... 313

8.1. Массивы.....	313
8.2. Коллекции, списки, итераторы	319
8.3. Перебор в цикле элементов коллекций. Оператор цикла <i>for-each</i>	323
8.4. Работа со строками в Java. Строки как объекты. Классы <i>String</i> , <i>StringBuffer</i> и <i>StringBuilder</i>	324
8.5. Типы-перечисления (enum).....	338
8.6. Работа с датами и временем. Класс <i>GregorianCalendar</i>	340
8.7. Работа с графикой и графическим пользовательским интерфейсом.....	343
Код визуализации и код бизнес-логики приложения.....	343
Графические примитивы.....	344
Пример метода, работающего с графикой	347
8.8. Исключительные ситуации	350
Обработка исключительных ситуаций.....	350

Иерархия исключительных ситуаций	353
Объявление типа исключительной ситуации и оператор <i>throw</i>	356
Объявление метода, который может возбуждать исключительную ситуацию. Зарезервированное слово <i>throws</i>	358
8.9. Работа с файлами и папками.....	361
Работа с файлами и папками с помощью объектов типа <i>File</i>	361
Выбор файлов и папок с помощью файлового диалога	365
Работа с потоками ввода/вывода.....	369
Краткие итоги по главе.....	377
Типичные ошибки.....	379
Задания.....	379

Глава 9. Интерфейсы и композиция 381

9.1. Проблемы множественного наследования классов. Интерфейсы	381
9.2. Отличия интерфейсов от классов. Проблемы наследования интерфейсов.....	384
9.3. Пример работы с интерфейсами.....	386
9.4. Композиция — еще одна альтернатива множественному наследованию.....	389
Краткие итоги по главе.....	391
Типичные ошибки.....	391
Задания.....	391

Глава 10. Многопоточное программирование и многоядерные системы 393

10.1. Потоки выполнения (threads) и синхронизация	393
10.2. Преимущества и проблемы при работе с потоками выполнения	394
10.3. Синхронизация по ресурсам и событиям.....	395
Синхронизация по ресурсам	395
Синхронизация по событиям	397
10.4. Класс <i>Thread</i> и интерфейсы <i>Runnable</i> и <i>Callable</i> . Создание и запуск потока выполнения.....	398
10.5. Поля и методы, заданные в классе <i>Thread</i>	400
10.6. Работа многопоточных приложений в многопроцессорных и многоядерных системах.....	402
Причины перехода к многопроцессорным и многоядерным системам.....	402
Пример многопоточной программы	404
Работа многопоточного приложения на многоядерном компьютере	412
10.7. Синхронизация на основе интерфейсов <i>Lock</i> и <i>Condition</i>	414
Интерфейс <i>Lock</i>	414
Интерфейс <i>Condition</i>	418
Краткие итоги по главе.....	420
Типичные ошибки.....	421
Задания.....	422

Глава 11. Введение в сетевое программирование..... 423

11.1. Краткая справка по языку HTML	423
Система WWW и язык HTML	423

Теги и их атрибуты.....	425
Математические и специальные символы, греческие буквы	431
11.2. Апплеты	433
Структура апплета	433
Примеры апплетов. Аннотация <i>@Override</i>	436
Создание проекта с апплетами	440
Редактор HTML-документов, стилей CSS и кода JavaScript.....	443
11.3. Сервлеты.....	448
Создание сервлета и конфигурирование сервера приложений.....	450
Локализация сервлетов — работа с языковыми форматами.....	457
11.4. Технология JSP — Java Server Pages.....	462
11.5. Технология JSTL — JSP Standard Tag Library.....	467
Краткие итоги по главе.....	469
Типичные ошибки.....	470
Задания.....	470

Глава 12. Вложенные классы..... 471

12.1. Виды вложенных классов.....	471
12.2. Статические (static) вложенные классы и интерфейсы.....	472
12.3. Внутренние (inner) классы	474
12.4. Локальные (local) классы	477
12.5. Анонимные (anonymous) классы и обработчики событий	477
12.6. Анонимные (anonymous) классы и слушатели событий (listeners)	478
Краткие итоги по главе.....	481
Типичные ошибки.....	483
Задания.....	483

Глава 13. Приложение с графическим интерфейсом —

***DesktopApplication*..... 485**

13.1. Структура простой заготовки <i>DesktopApplication</i>	485
13.2. Исходный код класса <i>DesktopApplication</i>	487
13.3. Исходный код класса <i>DesktopApplicationView</i>	489
13.4. Конструктор главной формы приложения. Ресурсы приложения и концепция управляющих синглетонов.....	493
13.5. Аннотация <i>@Action</i> и показ справки.....	498
13.6. Концепция управляющего объекта-модели. Модель <i>TaskMonitor</i> , класс <i>Task</i> и обработчик <i>propertyChange</i>	500
13.7. Создание задачи типа <i>Task</i> . Назначение иконок и добавление изображений.....	507
13.8. Локализация приложения.....	512
13.9. Измерение времени и досрочное прекращение выполнения задания	518
13.10. Работа с межпрограммным буфером обмена — Clipboard	519
Копирование строк и изображений из буфера обмена в программу.....	519
Копирование строк из программы в буфер обмена. Класс <i>StringSelection</i>	521
Копирование изображений из программы в буфер обмена. Создание класса <i>ImageSelection</i>	523
13.11. Многопоточная система вычислений и индикации	527

Краткие итоги по главе.....	531
Типичные ошибки.....	532
Задания.....	532

Глава 14. Программирование мобильных телефонов..... 533

14.1. Спецификация Java Micro Edition. Конфигурации и профили. Мидлеты	533
14.2. Создание приложений для мобильных устройств.....	534
14.3. Дизайнер пользовательского интерфейса мидлета	540
14.4. Исходный код мидлета <i>Hello, World!</i>	546
14.5. Заготовка мидлета "Крестики-нолики"	555
14.6. Код бизнес-логики мидлета	564
14.7. Создание собственного компонента, являющегося наследником <i>TableItem</i> , и мидлета на его основе. Создание библиотеки	571
14.8. Запуск мидлета "Крестики-нолики" на реальном телефоне.....	579
14.9. Пример мидлета с переключением экрана и списком выбора.....	581
14.10. Примеры приложений для мобильных телефонов, поставляемые со средой NetBeans	582
Краткие итоги по главе.....	586
Типичные ошибки.....	587
Задания.....	587

Глава 15. Компонентное программирование..... 589

15.1. Компонентная архитектура JavaBeans	589
15.2. Создание компонента в NetBeans 6.5	590
15.3. Пример создания компонента в NetBeans 6.5 — панель с заголовком	594
15.4. Добавление в компонент новых свойств	598
15.5. Добавление в компонент новых событий	600
Краткие итоги по главе.....	603
Типичные ошибки.....	605
Задания.....	605

Глава 16. Платформо-зависимые методы и модуль C/C++ Pack..... 607

16.1. Использование динамических библиотек. Объявление платформо-зависимых (native) методов.....	607
16.2. Интерфейс JNI (Java Native Interface) взаимодействия Java с C/C++. Соответствие типов Java и C++	608
16.3. Пакет C/C++ Pack. Подключение компиляторов C++, C и библиотек.....	614
16.4. Создание приложений C++ и C	624
16.5. Примеры приложений с native-методами	628
Первый этап. Создание исходного кода Java для native-методов.....	628
Второй этап. Генерация h-файла заголовка C++, содержащего декларацию методов	629
Третий этап. Написание на C++ реализации методов, заголовки которых заданы в сгенерированном файле <code>jnipackage_JniMultiply.h</code>	630
Четвертый этап. Компиляция кода C++ в динамическую библиотеку (DLL)	631

Пятый этап. Запуск приложения вместе с DLL в NetBeans	632
Шестой этап. Распространение приложения вместе с DLL	632
Краткие итоги по главе.....	637
Типичные ошибки.....	637
Задания.....	638
Глава 17. Отладка, тестирование и профилирование программ	639
17.1. Отладка приложений в среде NetBeans.....	639
17.2. Команды режима пошагового выполнения	643
17.3. Пример отладки неправильно работающего приложения.....	650
17.4. Принцип "презумпции виновности" и тестирование классов	654
17.5. Профилирование приложений	668
17.6. Профилирование отдельных методов	677
17.7. Настройка профилировщика "на лету". Точки профилирования	684
17.8. Нахождение проблем с использованием памяти — телеметрия.....	692
17.9. Профилирование использования памяти классами.....	695
Краткие итоги по главе.....	701
Типичные ошибки.....	702
Задания.....	702
Заключение	703
Описание прилагаемого компакт-диска.....	705
Литература.....	711
Предметный указатель	713

Введение

Предлагаемый учебный курс может рассматриваться как краткое введение в современное программирование на Java и обзор возможностей NetBeans — одной из самых популярных и быстро развивающихся сред разработки программного обеспечения для языков Java, C, C++, Fortran, PHP, JavaScript, Ruby, Groovy и многих других.

О среде NetBeans в русскоязычной литературе рассказывается только в данной книге автора, а в иностранной — имеется около десятка таких книг, причем лишь в двух-трех из них излагается курс программирования на Java с использованием NetBeans. Поэтому читателю, желающему научиться программированию на языке Java, данное учебное пособие предоставляет уникальную возможность сделать это с помощью наиболее современных средств, заметно облегчающих работу программиста и делающих ее гораздо более эффективной по сравнению с методиками, описанными в других книгах.

Первое издание книги, вышедшее в марте 2008 года, было полностью распродано к декабрю того же года. Оно оказалось очень популярным как среди начинающих программистов, так и среди профессионалов, переходящих с C или C++ на Java. Возникла необходимость во втором издании. В нем изложена работа с NetBeans версии 6.5 (первое издание основывалось на версии 6.0), а также исправлены ошибки и опечатки, допущенные в первом издании. Кроме того, добавлен раздел, в котором NetBeans сравнивается с другими средами разработки Java-программ, заметно расширено описание работы со строками и дописаны разделы про современные средства Java для работы с датой и временем, а также про применение в NetBeans тестовых пакетов и библиотеки JUnit 4.

Несмотря на то, что языку Java уже посвящены тысячи книг и учебных курсов, все книги, написанные до выхода JDK 5 (прежнее обозначение JDK 1.5), не соответствуют современным представлениям об этом языке.

Данное учебное пособие содержит информацию об основных синтаксических конструкциях современной версии языка Java (JDK 6), типичных ошибках их использования, рекомендации по написанию на языке Java эффективно работающих программ.

Одной из привлекательных особенностей языка Java с самого начала была бесплатность распространения базовых средств разработки SDK (Software Development Kit) и исполняющей среды Java (виртуальной Java-машины). Однако компилятор, входящий в состав SDK, работал в режиме командной строки, т. е. отставал идеологически по крайней мере лет на 20 от современных профессиональных компиляторов. В 2006 году корпорация Sun Microsystems пошла на беспрецедентный шаг — сделала бесплатными свои профессио-

нальные средства разработки программного обеспечения. В настоящее время все они основаны на бесплатно распространяемой среде разработки NetBeans (платформе NetBeans), подключаемых к ней модулях (packs) и наборе библиотечных компонентов (Beans — зерен). Интересна игра слов: язык Java получил наименование по названию кофе, который любили программисты, так что Java Beans может быть расшифровано как "зерна кофе Java".

Среда NetBeans дала возможность разрабатывать программное обеспечение на языке Java с помощью самых современных методик. Она имеет развитые средства визуального проектирования пользовательского интерфейса, не уступающие Delphi, развитый редактор исходного кода программы с подсветкой ошибок непосредственно во время редактирования, а также множество других возможностей, о которых рассказывается в данной книге.

В NetBeans версии 6.5 не только существенно усовершенствован редактор исходного кода и отладчик, но и изменен сам подход к развитию среды. Если предыдущие версии ориентировались только на язык Java и в небольшой степени на C/C++, то в NetBeans 6.5 произошел переход к поддержке широкого спектра языков программирования. И если раньше корпорация Sun Microsystems, разработчик языка Java и значительной части среды NetBeans, основное внимание уделяла языку Java, то в последнее время акцент сменился — среда NetBeans и технология Java Scripting (исполнение виртуальной Java-машиной программ, написанных на отличных от Java языках программирования) оказались удобными инструментами для поддержки огромного количества языков программирования и сопутствующих им технологий. При этом связующим звеном с NetBeans и Java-машиной остается язык Java.

В состав дистрибутива NetBeans 6.5 (имеется на прилагаемом к книге компакт-диске) входят все основные модули, необходимые для программирования на Java. Кроме того, в него входит модуль "C/C++ Pack", позволяющий программировать в среде NetBeans не только на Java, но и на языках C и C++, благодаря подключению к NetBeans внешних компиляторов C/C++. Это дает возможность разрабатывать полный спектр Java-программ в одной и той же среде. Раньше для создания платформо-зависимого (native) кода, используемого native-методами классов Java, требовались сторонние среды разработки программ, написанных на языках C/C++.

Среда разработки NetBeans может быть установлена с прилагаемого к книге компакт-диска или свободно загружена с сайта Sun. Она распространяется на условиях лицензии, дающей право использовать разработанное с помощью среды NetBeans программное обеспечение не только в некоммерческих, но и в коммерческих целях. Таким образом, теперь как начинающим разработчикам, так и крупным фирмам нет необходимости тратить деньги на покупку профессиональных средств разработки — можно работать в среде NetBeans.

Почему же это не бесплатный сыр? Фирма Sun получает деньги не у частных лиц или маленьких компаний, а у крупных корпоративных пользователей за техническую поддержку своих бесплатно распространяемых приложений. И лидерство в данной области косвенно обеспечивает гораздо большие прибыли, чем продажа этих продуктов.

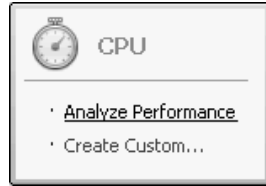
Java очень быстро развивается, особенно в последние годы. Из-за этого многие прежние представления и методы программирования на Java оказываются устаревшими. Среда

NetBeans развивается еще быстрее — версия 6.5, появившаяся в 2008 году, заметно превосходит по возможностям версию 6.0, вышедшую в 2007 году. Данная книга — первый шаг к изучению современных технологий на основе Java и NetBeans.

Изложение ведется на основе языка Java для JDK версии 6 и среды NetBeans версии 6.5. Примеры, приведенные в данной книге, проверены в среде NetBeans версии 6.5 с JDK 6.

Поскольку технология Java мультиплатформенна, то все изложенное справедливо как для MS Windows, так и для Solaris, Linux, Mac OS и других операционных систем.

Автор выражает благодарность рецензентам фирмы Sun Microsystems Алексею Акопяну и Алексею Уткину за полезные замечания по первоначальному варианту данного учебного пособия, Игорю Богдановичу Керницкому за содействие в подготовке примеров с native-кодом, а также жене Светлане за помощь в оформлении книги.



ГЛАВА 1

Общие представления о языке Java

1.1. Java и другие языки программирования. Системное и прикладное программирование

Язык программирования Java был создан в рамках проекта корпорации Sun Microsystems по разработке компьютерных программно-аппаратных комплексов нового поколения. Первая версия языка была официально опубликована в 1995 году. С тех пор язык Java стал стандартом "де-факто", вытеснив за десять лет языки C и C++ из многих областей программирования. В 1995 году они были абсолютными лидерами, но к 2006 году число программирующих на Java стало заметно превышать число программистов, использующих C и C++, и сейчас составило более четырех с половиной миллионов человек. А число устройств, запрограммированных с помощью Java, превысило полтора миллиарда.

Как связаны между собой языки C, C++, JavaScript и Java? Что между ними общего и в чем они отличаются? В каких случаях следует, а в каких не следует их применять? Для того чтобы ответить на эти вопросы, нужно сначала остановиться на особенностях программного обеспечения предыдущих поколений и на современных тенденциях в развитии программного обеспечения.

Первоначально компьютеры программировались в машинных кодах. Затем появились языки ассемблера, которые заменили команды процессоров мнемоническими сокращениями, гораздо более удобными для человека, чем последовательности нулей и единиц. Их принято считать языками программирования *низкого уровня* (т. е. близкими к аппаратному уровню), т. к. они ориентированы на особенности конкретных процессоров. Программы, написанные на языках ассемблера, нельзя было переносить на компьютеры с другим типом процессора из-за несовместимости наборов команд (т. е. несовместимости на уровне исходного кода — source code).

Программы в *машинных кодах* (т. е. в виде последовательности нулей и единиц), соответствующих командам процессора и необходимым для них данным, нет надобности как-то преобразовывать. Их можно просто скопировать в нужное место памяти компьютера и передать управление первой команде программы (задать точку входа в программу).

Программы, написанные на каком-либо языке программирования, сначала нужно перевести из одной формы (текстовой) в другую (двоичную, т. е. в машинные коды). Процесс

такого перевода называется *трансляцией* (от англ. *translation* — "перевод", "перемещение"). Не обязательно переводить программу из текстовой формы в двоичные коды, возможен процесс трансляции с одного языка программирования на другой. Или из кодов одного типа процессора в коды другого типа.

Имеются два основных вида трансляции — *компиляция* и *интерпретация*.

При *компиляции* первоначальный набор инструкций однократно переводится в исполняемую форму (машинные коды), а в последующем при работе программы используются только эти коды.

При *интерпретации* во время вызова необходимых инструкций каждый раз сначала происходит перевод инструкций из одной формы (текстовой или двоичной) в другую — в исполняемые коды процессора используемого компьютера. И только потом эти коды исполняются. Естественно, что интерпретируемые коды исполняются медленнее, чем скомпилированные, т. к. перевод инструкций из одной формы в другую обычно занимает в несколько раз больше времени, чем выполнение полученных инструкций. Но интерпретация обеспечивает большую гибкость по сравнению с компиляцией, и в ряде случаев без нее не обойтись.

В 1956 году появился язык Fortran — первый язык программирования *высокого уровня* (т. е. не ориентированный на конкретную аппаратную реализацию компьютера). Он обеспечил переносимость программ на уровне исходных кодов, но довольно дорогой ценой. Во-первых, быстродействие программ, написанных на этом языке, было в несколько раз меньше, чем для ассемблерных. Во-вторых, такие программы занимали примерно в два раза больше места в памяти компьютера, чем ассемблерные. И, наконец, пришлось отказаться от поддержки особенностей периферийных устройств — ограничить общение с "внешним миром" простейшими возможностями, которые в программе одинаково реализовывались для ввода/вывода с помощью перфокарточного считывателя, клавиатуры, принтера, текстового дисплея и т. д. Тем не менее, языки программирования высокого уровня постепенно вытеснили ассемблер, поскольку обеспечивали не только переносимость программ, но и гораздо большую их надежность, а также несоизмеримо более высокую скорость разработки сложного программного обеспечения. А Fortran до сих пор остается важнейшим языком программирования для высокопроизводительных численных научных расчетов.

Увеличение аппаратных возможностей компьютеров (рост объема оперативной и дисковой памяти, значительное повышение быстродействия), а также появление разнообразных периферийных устройств привело к необходимости пересмотра того, как должны работать программы. Массовый выпуск компьютеров потребовал унификации доступа из программ к различным устройствам, обеспечивая игнорирование особенностей их аппаратной реализации. Это можно осуществить, если обращение к устройству идет не напрямую, а через прилагающуюся программу — драйвер устройства (по-английски *driver* означает "водитель").

Появились операционные системы — наборы драйверов и специальных программ, распределяющих ресурсы компьютера между разными программами. Соответственно, программное обеспечение стало разделяться на *системное* и *прикладное*. Системное программное обеспечение — непосредственно обращающееся к аппаратуре, прикладное — решающее какие-либо прикладные задачи и использующее аппаратные возможности

компьютера не напрямую, а через вызовы программ операционной системы. Прикладные программы стали приложениями операционной системы (сокращенно *приложениями* — applications). Этот термин означает, что программа может работать только под управлением операционной системы. Если на том же компьютере установить другой тип операционной системы, то программа-приложение первой операционной системы, возможно, не будет работать.

Требования к прикладным и системным программам принципиально различаются. От системного программного обеспечения требуется максимальное быстродействие и минимальное количество занимаемых ресурсов, а также возможность доступа к любым необходимым аппаратным ресурсам. От прикладного — максимальная функциональность в конкретной предметной области. При этом быстродействие и занимаемые ресурсы не имеют значения до тех пор, пока не влияют на функциональность. Например, нет никакой разницы, реагирует ли программа на нажатие клавиши за одну десятую или за одну миллионную долю секунды. Правда, на первоначальном этапе создания прикладного программного обеспечения даже прикладные по назначению программы являлись системными по реализации, т. к. были вынуждены напрямую обращаться к аппаратуре.

Язык C был создан в 1972 году в одной из исследовательских групп Bell Laboratories при разработке операционной системы UNIX. Сначала была предпринята попытка написать операционную систему на ассемблере, но после появления в группе новых компьютеров возникла необходимость в новом платформи-независимом языке программирования высокого уровня, с помощью которого можно было бы писать операционные системы. Таким образом, язык C предназначался для создания системного программного обеспечения, и таким он остается до сих пор. Его идеология и синтаксические конструкции ориентированы на максимальную близость к аппаратному уровню реализации операций — в той степени, в какой он может быть обеспечен на аппаратно-независимом уровне. При этом главным требованием была максимальная скорость работы и минимальное количество занимаемых ресурсов, а также возможность доступа ко всем аппаратным средствам. Язык C — это язык *процедурного программирования*, т. к. его базовыми конструкциями являются *подпрограммы*. В общем случае подпрограммы принято называть подпрограммами-процедурами (откуда и идет название "процедурное программирование") и подпрограммами-функциями. Но в C имеются только подпрограммы-функции. Обычно их называют просто *функциями*.

Язык C произвел настоящую революцию в разработке программного обеспечения, получил широкое распространение и стал промышленным стандартом. Он до сих пор применяется для написания операционных систем и программирования микроконтроллеров. Но мало кто в полной мере осознает причины его популярности. В чем они заключались? В том, что он смог обеспечить необходимую функциональность программного обеспечения в условиях низкой производительности компьютеров, крайней ограниченности их ресурсов и неразвитости периферийных устройств! При этом повторилась та же история, что и с языком Fortran, но теперь уже для языка системного программирования. Переход на язык программирования высокого уровня, но с минимальными потерями по производительности и ресурсам, дал большие преимущества.

Большое влияние на развитие теории программирования оказал язык Pascal, разработанный в 1974 году швейцарским профессором Никлаусом Виртом. В этом проекте сочетались два оригинальных замысла. Первый — собственно язык Pascal, предназначенный

для обучения идеям *структурного программирования*. Второй — идея *виртуальной машины*. Никлаус Вирт предложил обеспечить переносимость программ, написанных на Pascal, за счет компиляции их в набор команд некоей абстрактной Р-машины (Р — сокращение от Pascal), а не в исполняемый код конкретной аппаратной платформы. А на каждой аппаратной платформе должна была работать программа, интерпретирующая эти коды. Говорят, что такая программа эмулирует (т. е. имитирует) систему команд несуществующего процессора. А саму программу называют виртуальной машиной.

В связи с ограниченностью ресурсов компьютеров и отсутствием в Pascal средств системного программирования этот язык не смог составить конкуренцию языку С, т. к. практически все промышленное программирование вплоть до середины последнего десятилетия XX века по реализации было системным. Идеи Р-машины были в дальнейшем использованы и значительно усовершенствованы в Java.

Развитие теории и практики программирования привело к становлению в 1967—1972 годах нового направления — *объектного программирования*, основанного на концепциях работы с *классами* и *объектами*. Оно обеспечило принципиально новые возможности по сравнению с процедурным. Были предприняты попытки расширения различных языков путем введения в них конструкций объектного программирования.

В 1982 году Бьерном Страуструпом путем такого расширения языка С был создан язык, который он назвал "С с классами". В 1983 году после очередных усовершенствований им был создан первый компилятор языка С++. Два плюса означают: "язык С с очень большим количеством добавлений", и одновременно — "новое, увеличенное на один по сравнению с прежним, значение" (игра понятий: в С существует оператор "+").

Язык С++ является надмножеством над языком С — на нем можно писать программы как на "чистом С", без использования каких-либо конструкций объектного программирования. В связи с этим, а также и с дополнительными преимуществами объектного программирования, язык С++ быстро приобрел популярность и стал промышленным стандартом (сначала "де-факто", а потом и "де-юре"). Так что в настоящее время С++ признан базовым языком системного программирования. Длительное время он использовался и для написания прикладных программ. Но, как мы уже знаем, требования к прикладным программам совпадают с требованиями к системным только в том случае, когда быстродействие компьютера можно рассматривать как низкое, а ресурсы компьютера — малыми. Кроме этого, у языков С и С++ имеются еще два принципиальных недостатка: а) низкая надежность на уровне как исходного, так и исполняемого кода; б) отсутствие переносимости на уровне исполняемого кода. С появлением компьютерных сетей эти недостатки стали очень существенным ограничивающим фактором, поскольку вопросы безопасности при работе в локальных, а особенно в глобальных сетях приобретают первостепенную значимость.

В 1995 году появились сразу два языка программирования, имеющие в настоящее время огромное значение: Java, разработанный в корпорации Sun, и JavaScript, созданный в небольшой фирме Netscape Communication, получившей к тому времени известность благодаря браузеру Netscape Navigator.

Язык Java создавался как универсальный, предназначенный для прикладного программирования в неоднородных компьютерных сетях как со стороны клиентского компьюте-

ра, так и со стороны сервера. В том числе — для использования на *тонких аппаратных клиентах* (устройствах малой вычислительной мощности с крайне ограниченными ресурсами). При этом скомпилированные программы Java работают только под управлением виртуальной Java-машины, поэтому они называются *приложениями Java*. Синтаксис операторов Java практически полностью совпадает с синтаксисом языка C, но в отличие от C++, Java это не расширение C, а совершенно независимый язык, со своими собственными синтаксическими правилами. Он гораздо сильнее типизирован по сравнению с C и C++, т. е. вносит больше ограничений на действия с переменными и величинами разных типов. Например, в C/C++ нет разницы между целочисленными числовыми, булевыми и символьными величинами, а также адресами в памяти. Так, можно умножить символ на булево значение `true` (истина), из которого вычтено целое число, и разделить результат на адрес! В Java введен вполне разумный запрет на почти все действия такого рода.

JavaScript создавался как узкоспециализированный прикладной язык программирования HTML-страниц, расширяющий возможности HTML, и он в полной мере отвечает этим потребностям до сих пор. Про язык HTML, являющийся основой для создания WWW-документов, более подробно будет рассказано в *главе 11*.

Следует подчеркнуть, что язык JavaScript *не имеет никакого отношения* к Java. Слово "script" означает сценарий. Включение слова "Java" в название JavaScript — рекламный трюк фирмы Netscape Communication. Язык JavaScript также C-образен, но в отличие от Java, интерпретируемый. Основное его назначение — программное управление элементами WWW-документов. Языки HTML и XML позволяют задавать статический, неизменный внешний вид документов, с их помощью невозможно запрограммировать реакцию на действия пользователя. JavaScript позволяет ввести элементы программирования в поведение документа. Программы, написанные на JavaScript, встраиваются в документы в виде исходных кодов (сценариев) и имеют небольшой размер. Для упрощения работы с динамически формируемыми документами JavaScript имеет свободную типизацию — переменные меняют тип по результату присваивания. Поэтому программы JavaScript гораздо менее надежны, чем C/C++, не говоря уже про Java.

Java, JavaScript и C++ относятся к объектно-ориентированным языкам программирования, все они имеют C-образный синтаксис операторов. Но как объектные модели, так и базовые конструкции этих языков (за исключением синтаксиса операторов) принципиально различны. Ни один из них не является версией или упрощением другого — это совсем разные языки, предназначенные для различных целей.

Итак:

- ◆ Java — универсальный язык прикладного программирования;
- ◆ JavaScript — узкоспециализированный язык программирования HTML-документов;
- ◆ C++ — универсальный язык системного программирования.

В 2000 году в корпорации Microsoft была разработана платформа .NET (читается "дот-нет", DotNet — в переводе с английского "точка NET"). Она стала альтернативой платформе Java и во многом повторила ее идеи. Основное различие заключалось в том, что для этой платформы можно было использовать произвольное количество языков программирования, а не один. Причем классы .NET оказываются совместимыми как в целях

наследования, так и по исполняемому коду независимо от языка, на котором они написаны. Важнейшим языком .NET стал Java-образный язык C# (читается "Си шарп"). Фактически, C# унаследовал от Java большинство особенностей — динамическую объектную модель, сборку "мусора", основные синтаксические конструкции, хотя он вполне самостоятельный язык программирования, имеющий много привлекательных черт. В частности, компонентные модели Java и C# принципиально отличаются.

Java стал первым универсальным C-образным языком прикладного программирования, что обеспечило легкость перехода на этот язык большого числа программистов, знакомых с C и C++. А наличие средств строгой проверки типов, ориентация на работу с компьютерными сетями, переносимость на уровне исполняемого кода и поддержка платформо-независимого графического интерфейса, а также запрет прямого обращения к аппаратуре обеспечили выполнение большинства требований, предъявлявшихся к языку прикладного программирования. Чем больше становятся быстродействие и объем памяти компьютеров, тем больше потребность в разделении прикладного и системного программного обеспечения. Соответственно, для прикладных программ исчезает необходимость напрямую обращаться к памяти и другим аппаратным устройствам компьютера. Поэтому среди прикладных программ с каждым годом растет доля программного обеспечения, написанного на Java и языках .NET. Но как по числу программистов, так и по числу устройств, использующих соответствующие платформы, Java в настоящее время лидирует с большим отрывом.

1.2. Виртуальная Java-машина, байт-код, JIT-компиляция. Категории программ, написанных на языке Java

Первоначально слово "программа" означало последовательность инструкций процессора для решения какой-либо задачи. Эти инструкции представляли собой машинные коды, и разницы между исходным и исполняемым кодом программы не было. Разница появилась, когда программы стали писать на языках программирования. При этом программой стали называть как текст, содержащийся в файле с исходным кодом, так и исполняемый файл.

Для устранения неоднозначности термина "программа" исполняемый код программы принято называть приложением (application). Термин "приложение" — сокращение от фразы "приложение операционной системы". Он означает, что исполняемый код программы может работать только под управлением соответствующей операционной системы. Работа под управлением операционной системы позволяет избежать зависимости программы от устройства конкретного варианта аппаратуры на компьютере, где она должна выполняться. Например, как автору программы, так и пользователю совершенно безразлична аппаратная реализация устройства, с которого считывается информация: будет ли это жесткий диск с одной, двумя или шестнадцатью считывающими головками, или это будет CD-привод, DVD-привод, или еще какой-либо другой тип носителя. Но переносимость обычных приложений ограничивается одним типом операционных систем. Например, приложение MS Windows не будет работать под Linux, и наоборот. Су-

существует возможность исключить зависимость приложения от операционной системы, введя выполнение его команд не средствами операционной системы, а *исполняющей средой* — программой, контролирующей работу приложения и выполняющей его вызовы. Программы, написанные на языке Java, выполняются исполняющей средой — виртуальной Java-машиной, и поэтому обладают переносимостью на любую операционную систему, где имеется соответствующая Java-машина. Благодаря этому они являются не приложениями какой-либо операционной системы, а *приложениями Java*.

Программы, написанные на языке Java, представляют собой наборы классов (о классах будет рассказано несколько позже) и сохраняются в текстовых файлах с расширением .java. При компиляции текст программы переводится (транслируется) в двоичные файлы с расширением .class. Такие файлы содержат байт-код, представляющий собой совокупность инструкций для абстрактного Java-процессора в виде байтовых последовательностей команд этого процессора и данных к ним. Для того чтобы выполнить байт-код на каком-либо компьютере, его нужно перевести в инструкции для соответствующего процессора. Именно этим и занимается Java-машина. Первоначально байт-код всегда интерпретировался следующим образом: каждый раз, когда встречалась какая-либо инструкция Java-процессора, она переводилась в последовательность инструкций процессора компьютера, что, естественно, значительно замедляло работу приложений Java.

В настоящее время используется более сложная схема, называемая JIT-компиляцией (Just-In-Time) — это компиляция "по ходу дела", "на лету". Когда какая-либо инструкция (или набор инструкций) Java-процессора выполняется в первый раз, происходит компиляция соответствующего ей байт-кода с сохранением скомпилированного кода в специальном буфере. При последующем вызове той же инструкции вместо ее интерпретации скомпилированный код извлекается из буфера. Поэтому интерпретация происходит только при первом вызове инструкции.

Сложные оптимизирующие JIT-компиляторы действуют еще изощренней. Поскольку обычно компиляция инструкции идет гораздо дольше по сравнению с интерпретацией, время ее выполнения в первый раз при наличии JIT-компиляции может заметно отличаться в худшую сторону по сравнению с "чистой" интерпретацией. Поэтому бывает выгоднее сначала запустить процесс интерпретации, а параллельно ему в фоновом режиме компилировать инструкцию. Только после окончания процесса компиляции при последующих вызовах инструкции будет исполняться ее скомпилированный код (а до этого все ее вызовы будут интерпретироваться). Разработанная Sun виртуальная машина HotSpot осуществляет JIT-компиляцию только тех участков байт-кода, которые критичны ко времени выполнения программы. При этом по ходу работы программы происходит оптимизация скомпилированного кода.

Благодаря компиляции программ Java в платформу-независимый байт-код обеспечивается переносимость этих программ не только на уровне исходного кода, но и на уровне скомпилированных приложений. Конечно, при этом на компьютере, где выполняется приложение, должна быть установлена программа виртуальной Java-машины (Java Virtual Machine — JVM), скомпилированная в коды соответствующего процессора (native code — платформу-зависимый код). На одном и том же компьютере может быть несколько Java-машин разных версий или от разных производителей. Спецификация Java-машины и требования к компилятору языка Java открытые, поэтому различные фирмы (а не только Sun) разрабатывают компиляторы Java и Java-машины.

Обычно приложение операционной системы запускается с помощью средств самой операционной системы. Приложение Java, напротив, исполняется с помощью виртуальной Java-машины, которая сама является приложением операционной системы. Таким образом, сначала стартует Java-машина. Она получает в качестве параметра имя файла с скомпилированным кодом класса. В этом классе ищется и запускается на выполнение подпрограмма с именем `main`.

Приложения Java обладают не только хорошей переносимостью, но и высокой скоростью работы. Однако даже при наличии JIT-компиляции они все-таки выполняются медленнее, чем программы, написанные на C или C++. Это связано с тем, что JIT-компиляция создает не настолько оптимальный код, как многопроходный компилятор C/C++ (он имеет возможность тратить много времени и ресурсов на поиск конструкций программы, которые можно оптимизировать). А JIT-компиляция происходит "на лету", в условиях жесткой ограниченности времени и ресурсов. Для решения этой проблемы были разработаны компиляторы программ Java в код конкретных программно-аппаратных платформ (`native code` — платформо-зависимый код). Например, свободно распространяемый фондом GNU компилятор GCJ (<http://gcc.gnu.org/java/>). Правда, заметные успехи Sun в усовершенствовании Java-машины позволили практически достичь, а в ряде случаев даже обогнать по быстродействию программы, написанные на других языках, и отказаться от статической компиляции в платформо-зависимый код. В частности, приложения Java, активно занимающиеся выделением/высвобождением памяти, работают быстрее своих аналогов, написанных на C/C++, благодаря специальному механизму программных слотов памяти (`slot` — паз, отверстие для вставки чего-либо).

Виртуальная Java-машина не только исполняет байт-код (интерпретирует его, занимается JIT-компиляцией и исполняет JIT-компилированный код), но и выполняет ряд других функций (например, взаимодействует с операционной системой, обеспечивая доступ к файлам или поддержку графики), а также реализует автоматическое высвобождение памяти, занятой ненужными объектами, — так называемую сборку мусора (`garbage collection`).

Программы Java можно разделить на несколько основных категорий.

- ◆ Приложение (`application`) — аналог обычной прикладной программы.
- ◆ Апплет (`applet`) — специализированная программа с ограниченными возможностями, работающая в окне WWW-документа под управлением браузера.
- ◆ Мидлет (`midlet`) — специализированная программа с ограниченными возможностями, работающая на мобильном устройстве. В настоящее время мидлеты — один из наиболее популярных сегментов рынка программного обеспечения, пишущегося на языке Java.
- ◆ Сервлет (`servlet`) — специализированная программа с ограниченными возможностями, работающая в WWW на стороне сервера. Используется преимущественно в рамках технологии JSP (Java Server Pages — серверных страниц Java) для программирования WWW-документов со стороны сервера и обработки отсылаемой на сервер информации.
- ◆ Серверное приложение (`Enterprise application`) — предназначено для многократного выполнения на стороне сервера.

- ◆ Библиотека (Java Class Library — библиотека классов, либо NetBeans Module — модуль платформы NetBeans) — предназначена для многократного использования программами Java.

Между приложениями и апплетами Java существует принципиальное различие: приложение запускается непосредственно с компьютера пользователя и имеет доступ ко всем ресурсам компьютера наравне с любыми другими программами. Апплет же загружается из WWW с постороннего сервера, причем из-за самой идеологии WWW-сайт, с которого загружен апплет, в общем случае не может быть признан надежным. А сам апплет может передавать данные на произвольный сервер в WWW. Поэтому, для того чтобы избежать риска утечки конфиденциальной информации с компьютера пользователя или совершения враждебных действий, у апплетов убраны многие возможности, имеющиеся у приложений.

Сервлеты — это приложения Java, запускаемые на стороне сервера. Они обладают возможностью доступа к файловой системе и другим ресурсам сервера через набор управляющих конструкций, предопределенных в рамках технологии JSP и пакета javax.servlet. Кроме того, они имеют доступ к содержимому HTTP-запросов со стороны клиентского компьютера, т. е. того компьютера, на котором работает пользователь. HTTP (HyperText Transport Protocol) — это протокол передачи WWW-документов через Интернет.

Технология JSP заключается в наличии дополнительных конструкций (тегов) в HTML-или XML-документах, которые позволяют серверу автоматически генерировать и компилировать сервлеты. В результате удается очень просто и удобно обрабатывать данные или элементы документа, а также внедрять в нужные места документа результаты обработки. Автоматическая генерация сервлетов происходит всего один раз — перед первой обработкой документа с тегами JSP на стороне сервера, поэтому код JSP выполняется достаточно быстро, но конечно требует, чтобы была установлена соответствующая Java-машина (например, входящая в состав сервера GlassFish — программного обеспечения, обеспечивающего поддержку большого количества необходимых серверных возможностей для работы в WWW). Отметим, что программный сервер GlassFish, основанный на кодах Sun Application Server, распространяется бесплатно и входит в состав NetBeans 6.5.

Более подробная информация о сетевом программировании и серверных приложениях Java будет изложена в *главе 11*.

Первоначально корпорация Sun Microsystems позиционировала Java как язык, обеспечивающий развитые графические возможности WWW-документов благодаря включению в них апплетов. Однако в настоящее время основные области применения Java — прикладное программирование на основе приложений, страниц JSP и сервлетов, а также других видов серверных программ. При этом апплеты играют незначительную роль.

Виртуальную Java-машину часто называют исполняющей средой (Java Runtime Environment — JRE).

Существуют два основных способа установки Java-машины на клиентский компьютер:

- ◆ JRE из поставки Java Development Kit (JDK) — комплекта разработки программного обеспечения на языке Java. Как синоним может употребляться термин SDK — Software Development Kit (комплект разработки программного обеспечения);
- ◆ специализированный вариант JRE в составе интернет-браузера, называющийся Java plugin.

Комплект последних версий JDK можно свободно загружать с сайта Sun <http://java.sun.com/>.

При использовании апплетов требуется, чтобы в состав браузера входил специализированный комплект JRE. Как правило, он поставляется вместе с браузером и может при необходимости обновляться. Для MS Internet Explorer такой комплект и его обновления могут быть свободно загружены с сайта <http://www.java.com/>.

Имеется возможность установки Java-машины от различных производителей (не обязательно устанавливать комплект JDK от Sun). На одном и том же компьютере можно установить сразу несколько различных Java-машин (в том числе комплекты JDK разных версий). Правда, опыт показывает, что при этом некоторые программы, написанные на Java, теряют работоспособность (частично или полностью).

Комплекты JDK классифицируются исходя из версии Java (языка программирования и, соответственно, Java-машины) и типа создаваемых приложений. Так, ко времени написания данного текста выходили версии JDK 1.0, 1.1, 1.2, 1.3, 1.4, 1.5, 1.6 и готовится к выходу 1.7. В каждой версии есть ряд подверсий, не сопровождающихся изменением языка программирования, а связанных в основном с исправлением ошибок или внесением небольших изменений в библиотеки (например, 1.4.1_01 или 1.5.0_04).

Версии Java 1.0 и 1.1 принято называть Java 1. Все версии Java, начиная с 1.2, называют Java 2. Однако более надежно классифицировать по номеру JDK, т. к. язык Java для версии JDK 1.5 заметно отличается по возможностям от языка Java для более ранних версий JDK — в него добавлено большое количество новых синтаксических конструкций, а также изменен ряд правил. Поэтому код, правильный в Java для версии JDK 1.5, может оказаться неправильным в Java для версии JDK 1.4 (не говоря уже про Java для версии JDK 1.3 или 1.2). Кроме того, начиная с JDK 1.5 компания Sun перестала использовать в названиях комплектов программного обеспечения термин Java 2 и происходящие от него сокращения вида j2, а в нумерации комплектов JDK отказалась от префикса "1". Поэтому сейчас принято говорить о версиях JDK 5 или JDK 6, а не о JDK 1.5 или JDK 1.6. Но соответствующие папки при установке среды на компьютере все-таки называются jdk1.5 и jdk1.6.

Комплекты разработки JDK одной версии отличаются по типу создаваемых с их помощью приложений. По применяемым средствам разработки и условиям выполнения приложений Java различают три платформы:

- ◆ Микроиздание — Java Platform, Micro Edition (Java ME) <http://java.sun.com/javame/> для программирования "тонких аппаратных клиентов" (т. е. устройств, обладающих малыми ресурсами — налагодных компьютеров, сотовых телефонов, микроконтроллеров, смарт-карт). Старое название J2ME. Для разработки приложений применяется не JDK, а специальный набор инструментов (Sun Java Wireless Toolkit for CLDC, Sun Java Toolkit for CDC, NetBeans 6.5, инструментарий от других производителей). Приложения Java работают под управлением одной из двух имеющихся для этой платформы вариантов Java-машины: максимально облегченной килобайтной виртуальной машины KVM или несколько более мощной виртуальной машины CVM.
- ◆ Стандартное издание — Java Platform, Standard Edition (Java SE) <http://java.sun.com/javase/> для программирования "толстых клиентов" (т. е. устройств, об-

ладающих достаточно большими ресурсами — обычных компьютеров и серверов). Старое название J2SE. Основой средств разработки служит JDK. Приложения Java работают под управлением Java-машины JVM (Java Virtual Machine).

- ◆ Корпоративное издание — Java Platform, Enterprise Edition (Java EE) <http://java.sun.com/javaee/> для написания серверного программного обеспечения. Старое название J2EE. Является надстройкой над платформой Java SE, поэтому все приложения Java EE работают с использованием классов Java SE и под управлением JVM. В SDK данной платформы входит мощный программный сервер Sun Java System Application Server (сервер приложений Java), а также большое число других исполняющих сред, основанных на технологиях Java.

При распространении какого-либо продукта, написанного на Java для платформы Java SE, возможна установка только программного обеспечения Java-машины (JRE — Java Runtime Environment). Например, в случае Java 6 update 7 (обновление 7) комплект JDK назывался `jdk1.6.0_07`, и ему будет соответствовать комплект `jre1.6.0_07`. При этом создавалась папка с именем `jre1.6.0_07` с вложенными папками `bin` и `lib`. Начиная с Java 6 update 10 среда `jre` устанавливается в папку `jre6` без указания номера обновления — эта и более поздние версии допускают обновление JRE без скачивания и переустановки всего дистрибутива.

Фактически, Java 6 update 10 — новый этап в развитии Java, и его отличие от предыдущей версии Java 6 update 7 (обновления 8 и 9 отсутствовали) гораздо больше, чем различия между Java 5 и Java 6. Очередной большой этап — предстоящий переход к Java 7.

Состав `jre6` достаточно прост. В папке `bin` содержатся файлы и папки, необходимые для работы Java-машины и дополнительных инструментов для работы с ней в специальных режимах. В папке `lib` находятся вспомогательные файлы и библиотеки, в основном связанные с параметрами настроек системы.

Также возможна установка целиком JDK. Например, при установке Java 1.6 update 11 создается папка `jdk1.6.0_11` с вложенными папками:

- ◆ `bin` — содержатся файлы инструментов разработки;
- ◆ `demo` — файлы примеров с исходными кодами;
- ◆ `include` — заголовки файлов C для доступа к ряду библиотек Java и отладчику виртуальной Java-машины на платформо-зависимом уровне — на основе интерфейсов JNI (Java Native Interface) и JVMDI (Java Virtual Machine Debugging Interface) соответственно;
- ◆ `jre` — файлы, необходимые для работы с виртуальной Java-машиной;
- ◆ `lib` — ряд библиотек и сопроводительных файлов для работы инструментов из папки `bin`;
- ◆ `sample` — примеры с исходными кодами.

В папке `jdk1.6.0_11` содержится также архив `src.zip` с исходными кодами стандартных классов Java.

Следует отметить, что комплекты JRE, входящие в состав JDK и поставляемые отдельно от JDK, заметно различаются. Это связано с тем, что в отдельно поставляемом комплек-

те не нужны средства, предназначенные для отладки приложений и специальных отладочных режимов работы с Java-машиной (управления загрузкой, профилирования, проверки стека и т. д.).

К сожалению, в JDK отсутствует даже самая простейшая документация с описанием назначения имеющихся в нем инструментов, там даны лишь ссылки на сайт компании Sun, где можно найти эту информацию. В табл. 1.1 перечислено назначение основных инструментов.

Таблица 1.1. Средства разработки приложений

Утилита	Назначение
javac	Компилятор в режиме командной строки для программ, написанных на языке Java
java	Утилита для запуска в режиме командной строки откомпилированных программ-приложений
appletviewer	Утилита для запуска на исполнение и отладку апплетов без браузера. При этом не гарантируется работоспособность отлаженного апплета в браузере
jdb	Отладчик программ, написанных на языке Java
javadoc	Генератор документации по классам на основе комментариев, начинающихся с /**
jar	Создание и управление Java-архивами jar
javah	Генератор заголовочных файлов C/C++ для подключения к программам Java внешних библиотек C/C++ на основе интерфейса JNI
javap	Дизассемблер классов
extcheck	Утилита для обнаружения конфликтов между файлами архивов jar
native2ascii	Утилита для конвертации в режиме командной строки параметра, передаваемого в виде текста на национальном алфавите, в последовательность символов Unicode

Кроме того, имеются средства поддержки работы в WWW и корпоративных сетях (интранет) с интерфейсом RMI — интерфейсом удаленного вызова методов (Remote Methods Invocation). Это программы *rmic*, *rmiregistry*, *rmid*. Также присутствуют средства поддержки информационной безопасности *keytool*, *jarsigner*, *policytool* и ряд других категорий утилит.

Подчеркнем, что набор утилит JDK рассчитан на морально устаревший режим командной строки, и что в большинстве случаев гораздо удобнее и правильнее пользоваться современной профессиональной средой разработки NetBeans. Для ее работы из JDK необходим только комплект JRE.

1.3. Алфавит языка Java.

Десятичные и шестнадцатеричные цифры и целые числа. Резервированные слова

Подробную информацию об алфавите языка его Java и основных синтаксических конструкциях можно найти на сайте <http://java.sun.com/docs/books/jls/index.html> (на английском языке).

Алфавит языка Java

Алфавит языка Java состоит из букв, десятичных цифр, разделителей и специальных символов. Буквами считаются латинские буквы (кодируются в стандарте ASCII), буквы национальных алфавитов (кодируются в стандарте Unicode, кодировка UTF-16), а также соответствующие им символы, кодируемые управляющими последовательностями (о них будет рассказано чуть позже).

Буквы и цифры можно использовать в качестве идентификаторов (т. е. имен) переменных, методов и других элементов языка программирования. Правда, при наличии в идентификаторах букв национальных алфавитов в ряде случаев могут возникнуть проблемы — эти символы будут показываться в виде вопросительных знаков.

Непосредственно как буквы рассматривается только часть символов национальных алфавитов, остальные — специальные символы — используются в качестве операторов и разделителей языка Java и не могут входить в состав идентификаторов.

Латинские буквы ASCII:

- ◆ ABCD...XYZ — заглавные (прописные);
- ◆ abcd...xyz — строчные.

Дополнительные "буквы" ASCII:

- ◆ _ — знак подчеркивания;
- ◆ \$ — знак доллара.

Национальные буквы на примере русского алфавита:

- ◆ АБВГ...ЭЮЯ — заглавные (прописные);
- ◆ абвг...эюя — строчные.

Десятичные цифры:

0 1 2 3 4 5 6 7 8 9

Разделители (их девять, задаются как символы ASCII):

() { } [] ; , .

Десятичные и шестнадцатеричные цифры и целые числа

Целые числовые константы в исходном коде Java (так называемые литерные константы) могут быть десятичными или шестнадцатеричными. Они записываются либо символами ASCII, либо символами Unicode следующим образом.

Десятичные константы записывают как обычно (например, `-137`).

Константы-значения, записанные в виде последовательности символов (литер), называют *литерными* (иногда — *литералами*). В Java литерные константы бывают не только числовыми, но и булевыми (значения `true` и `false`), символьными (например, `'a'`, `'b'`, `'1'`, `'2'`), строковыми (например, `"Это - строка"`). Также имеется особая литерная константа `null`.

Шестнадцатеричная литерная константа целого типа начинается с символов $0x$ или $0X$ (цифра 0, после которой следует латинская буква x), затем идет само число в шестнадцатеричной нотации. Например:

$0x10$ соответствует $10_{16} = 16$;

$0x2F$ соответствует $2F_{16} = 47$ и т. д.

О шестнадцатеричной нотации будет рассказано чуть позже.

Ранее иногда применялись восьмеричные числа, в языках C/C++, а также и в Java их записывали в виде числа, начинающегося с цифры 0 (т. е. 010 означало $10_8 = 8$).

В настоящее время в программировании восьмеричные числа практически не применяются, а неадекватное использование ведущего нуля может приводить к логическим ошибкам в программе.

Целая литерная константа в обычной записи имеет тип `int`. Если после константы добавить букву `L` (или `l`, что хуже видно в тексте, хотя в среде разработки выделяется цветом), то она будет иметь тип `long`, обладающий более широким диапазоном значений, чем тип `int`.

Поясним теперь, что такое шестнадцатеричная нотация записи чисел и зачем она нужна.

Информация представляется в компьютере в двоичном виде — как последовательность битов. Бит — это минимальная порция информации, он может быть представлен в виде ячейки, в которой хранится ноль или единица. Но бит — слишком мелкая единица, поэтому в компьютерах информация хранится, кодируется и передается байтами — порциями по 8 битов.

В данной книге под "ячейкой памяти" будет пониматься непрерывная область памяти (с последовательно идущими адресами), выделенная программой для хранения данных. На рисунках мы будем изображать ячейку прямоугольником, внутри которого находятся хранящиеся в ней данные. Если у ячейки имеется имя, оно будет выведено рядом с этим прямоугольником.

Мы привыкли работать с числами, записанными в так называемой десятичной системе счисления. В этой системе имеются 10 цифр (от 0 до 9), а в числе имеются *десятичные разряды*. Каждый разряд слева имеет вес 10 по сравнению с предыдущим, т. е. для получения значения числа, соответствующего цифре в каком-то разряде, стоящую в нем цифру надо умножить на 10 в соответствующей степени (т. е. $52 = 5 * 10 + 2$; $137 = 1 * 10^2 + 3 * 10^1 + 7$, и т. п.).

В программировании десятичной системой счисления пользоваться не всегда удобно, т. к. в компьютерах информация организована в виде битов, байтов и более крупных порций. Человеку неудобно оперировать данными в виде длинных последовательностей нулей и единиц. В настоящее время в программировании стандартной является шестнадцатеричная система записи чисел. Например, с ее помощью естественным образом кодируется цвет, устанавливаются значения отдельных битов числа, осуществляется шифрование и дешифрование информации, и т. д. В этой системе счисления все очень похоже на десятичную, но только не 10, а 16 цифр, и вес разряда не 10, а 16. Первые десять цифр записывают обычными десятичными цифрами, а недостающие цифры (большие 9) — заглавными латинскими буквами: A, B, C, D, E, F:

0 1 2 3 4 5 6 7 8 9 A B C D E F

То есть: A = 10, B = 11, C = 12, D = 13, E = 14, F = 15.

Заметим, что в шестнадцатеричной системе счисления числа от 0 до 9 записываются одинаково, а превышающие 9 отличаются. Для чисел от 10 до 15 в шестнадцатеричной системе счисления используют буквы от A до F, после чего задействуют следующий шестнадцатеричный разряд. Десятичное число 16 в шестнадцатеричной системе счисления записывается как 10. Для того чтобы не путать числа, записанные в разных системах счисления, около них справа пишут индекс с указанием *основания* системы счисления. Для десятичной системы счисления это 10, для шестнадцатеричной — 16. Для десятичной системы основание обычно не указывают (если это не приводит к путанице).

Точно так же в технической литературе часто не указывают основание для чисел, записанных в шестнадцатеричной системе счисления, если в записи числа встречаются не только обычные цифры от 0 до 9, но и буквенные цифры от A до F. Обычно употребляют заглавные буквы, но допустимы и строчные.

Примеры

```

0x10 = 1016 = 16;
0x100 = 10016 = 16·16 = 256;
0x1000 = 100016 = (16)3 = 4096;
0x20 = 2016 = 2·16 = 32;
0x21 = 2116 = 2·16 + 1 = 33;
0xF = F16 = 15;
0x1F = 1F16 = 1·16 + 15 = 31;
0x2F = 2F16 = 2·16 + 15 = 47;
0xFF = FF16 = 15·16 + 15 = 255;

```

Более подробно представление чисел в компьютере будет рассмотрено в *главе 4*.

Зарезервированные слова и литералы языка Java

В табл. 1.2 приведены ключевые слова (keywords), зарезервированные для синтаксических конструкций языка. Их назначение нельзя переопределять внутри программы, они не могут быть идентификаторами (именами переменных, подпрограмм и т. п.).

Таблица 1.2. Зарезервированные слова языка Java

abstract	assert	boolean	break	byte
case	catch	char	class	const
continue	default	do	double	else
enum	extends	final	finally	float
for	goto	if	implements	import
instanceof	int	interface	long	native
new	package	private	protected	public
return	short	static	strictfp	super
switch	synchronized	this	throw	throws
transient	try	void	volatile	while

Кроме ключевых слов имеются предопределенные литералы `false`, `null` и `true`. Фактическое отличие предопределенных литералов от ключевых слов заключается только в том, что они представляют собой значения, а не элементы синтаксических конструкций. Их также нельзя переопределять внутри программы — использовать в качестве идентификаторов или изменять значение.

1.4. Управляющие последовательности. Символы Unicode. Специальные символы

Управляющие последовательности

Иногда в тексте программы в строковых константах требуются символы, которые обычным образом в текст ввести нельзя (например, кавычки, а также различные специальные символы). В этом случае прибегают к управляющей последовательности — символу обратной косой черты, после которой следует один управляющий символ.

Управляющие последовательности — символы формирования текста

В табл. 1.3 приведены управляющие последовательности, применяющиеся в языке Java для форматирования текста. Кроме очевидных символов форматирования текста, к ним относятся также кавычки (поскольку одни кавычки могут оказаться внутри других, что требует специальных соглашений в языке программирования).

Таблица 1.3. Управляющие последовательности

Символ	Что означает
<code>\b</code>	Возврат на один символ назад
<code>\f</code>	Перевод на новую страницу
<code>\n</code>	Перевод на новую строку
<code>\r</code>	Возврат к началу строки
<code>\t</code>	Горизонтальная табуляция
<code>\'</code>	Кавычка
<code>\"</code>	Двойные кавычки
<code>\\</code>	Обратная косая черта
<code>\u</code>	Начало кодировки символа Unicode

Управляющие последовательности Java, C и C++ в основном совпадают. В Java отсутствуют только последовательности, использование которых неактуально (`\a`, `\v`) или неудобно (`\?`).

Управляющие последовательности — символы Unicode

Управляющая последовательность может содержать несколько символов. Например, символы национальных алфавитов могут кодироваться последовательностью `\u`, после

которой идет код символа в шестнадцатеричной кодировке для кодовых таблиц UTF-16 или UTF-8, например:

\u0030 – \u0039 — цифры ISO-LATIN от 0 до 9;

\u0024 — знак доллара \$;

\u0041 – \u005a — буквы от A до Z;

\u0061 – \u007a — буквы от a до z.

Простые специальные символы

В табл. 1.4 приведены простые специальные символы (т. е. нецифровые и небуквенные символы, имеющие в языке программирования специальное назначение).

Таблица 1.4. Простые специальные символы

Символ	Что означает
+	Оператор сложения
–	Оператор вычитания
*	Оператор умножения
/	Оператор деления
%	Оператор остатка от целочисленного деления
=	Оператор присваивания
~	Оператор побитового дополнения (побитовое НЕ)
?	Вопросительный знак — часть тернарного (состоящего из трех частей) условного оператора "? :"
:	Двоеточие — часть условного оператора "? :". Также используется для задания метки — ставится после имени метки
^	Оператор XOR (исключающее ИЛИ)
&	Оператор AND (побитовое И)
	Оператор OR (побитовое ИЛИ)
!	Оператор NOT (НЕ)
>	Больше
<	Меньше
{	Левая фигурная скобка — разделитель, открывающий блок кода
}	Правая фигурная скобка — разделитель, закрывающий блок кода
,	Запятая — разделитель в списке параметров оператора; разделитель в составном операторе
.	Точка — десятичный разделитель в числовом литерном выражении; разделитель в составном имени для доступа к элементу пакета, класса, объекта, интерфейса
(Левая круглая скобка — разделитель, открывающий список параметров в операторах и группируемую часть в выражениях

Таблица 1.4 (окончание)

Символ	Что означает
)	Правая круглая скобка — разделитель, закрывающий список параметров в операторах и группируемую часть в выражениях
[Левая квадратная скобка — разделитель, открывающий индекс массива
]	Правая квадратная скобка — разделитель, закрывающий индекс массива
;	Точка с запятой — разделитель, обозначающий окончание оператора
'	Апостроф (одиночная кавычка) — открытие и закрытие символа
"	Двойные кавычки — открытие и закрытие строки символов
\	Обратная косая черта (backslash) — используется для задания управляющих последовательностей символов
	Знак пробела (невидимый)
	Знак табуляции (невидимый)
@	Коммерческое а ("эт") — знак начала аннотации
#	Не имеет специального назначения
¤	То же
«	То же
»	То же
№	То же
§	То же

Составные специальные символы

В табл. 1.5 приведены составные специальные символы — последовательности из двух, трех или четырех нецифровых и небуквенных символов, имеющие в языке программирования специальное назначение.

Таблица 1.5. Составные специальные символы

Символ	Что означает
++	Оператор инкремента (увеличения на 1); $x++$ эквивалентно $x = x + 1$
--	Оператор декремента (уменьшения на 1); $x--$ эквивалентно $x = x - 1$
&&	Оператор AND (логическое И)
	Оператор OR (логическое ИЛИ)
<<	Оператор левого побитового сдвига
>>>	Оператор беззнакового правого побитового сдвига

Таблица 1.5 (окончание)

Символ	Что означает
>>	Оператор правого побитового сдвига с сохранением знака отрицательного числа
==	Равно
!=	Не равно
+=	$y += x$ эквивалентно $y = y + x$
-=	$y -= x$ эквивалентно $y = y - x$
*=	$y *= x$ эквивалентно $y = y * x$
/=	$y /= x$ эквивалентно $y = y / x$
%=	$y \% = x$ эквивалентно $y = y \% x$
=	$y = x$ эквивалентно $y = y x$
^=	$y ^= x$ эквивалентно $y = y ^ x$
>>=	$y >> = x$ эквивалентно $y = y >> x$
>>>=	$y >>> = x$ эквивалентно $y = y >>> x$
<<=	$y << = x$ эквивалентно $y = y << x$
/*	Начало многострочного комментария
/**	Начало многострочного комментария, предназначенного для автоматического создания документации по классу
/	Конец многострочного комментария (открываемого как / или /**)
//	Однострочный комментарий

1.5. Идентификаторы. Переменные и типы. Примитивные и ссылочные типы

Идентификаторы — это имена переменных, подпрограмм-функций и других элементов языка программирования. Идентификаторы могут состоять только из букв и цифр, причем первой всегда должна быть буква (в том числе символы подчеркивания и доллара), а далее может идти их произвольная комбинация. Некоторые символы национальных алфавитов рассматриваются как буквы, поэтому их можно применять в идентификаторах. Но некоторые служат в качестве символов-разделителей, и в идентификаторах недопустимы.

Язык Java *регистро-чувствительный*. Это значит, что идентификаторы чувствительны к тому, в каком регистре (верхнем или нижнем) набираются символы. Например, имена `i1` и `l1` соответствуют разным идентификаторам. Это правило привычно для тех, кто знаком с языками C/C++, но может на первых порах вызвать сложности у тех, кто изучал язык Pascal, который допускает произвольный регистр символов.

Длина идентификатора в Java любая (но в пределах разумного). Так, даже при длине идентификатора во всю ширину экрана компилятор NetBeans правильно работает.

Переменная — это именованная ячейка памяти, которой сопоставлен *тип*, и содержимое которой может изменяться. Перед тем как задействовать какую-либо переменную, она должна быть задана в области программы, предшествующей тому месту, где эта переменная используется. При объявлении переменной сначала указывается ее тип, а затем идентификатор. Указание типа позволяет компилятору задавать размер ячейки (объем памяти, выделяемой под переменную или значение данного типа), а также допустимые правила действий с переменными и значениями этого типа.

В Java существует ряд predefined типов:

- ◆ `int` — целое число;
- ◆ `float` — вещественное число;
- ◆ `boolean` — логическое значение;
- ◆ `Object` — самый простой объектный тип (класс) Java;
- ◆ и т. д.

Также имеется возможность задавать собственные объектные типы (классы), о чем будет рассказано позже.

Объявление переменных `a1` и `b1`, имеющих некий тип `MyType1`, осуществляется так:

```
MyType1 a1,b1;
```

При этом `MyType1` — имя типа этих переменных.

Другой пример — объявление переменной `j` типа `int`:

```
int j;
```

Типы бывают predefined и пользовательские.

Например, `int` — predefined тип, а `MyType1` — пользовательский.

Для объявления переменной не требуется никакого зарезервированного слова, а имя типа пишется перед именами задаваемых переменных.

Объявление переменных может сопровождаться их *инициализацией* — присваиванием начальных значений. Приведем пример такого объявления целочисленных переменных `i1` и `i2`:

```
int i1=5;  
int i2=-78;
```

либо

```
int i1=5,i2=-78;
```

Присваивания вида: `int i1 = i2 = 5;`, характерные для C/C++, запрещены.

Для начинающих программистов отметим, что символ "=" используется в Java и многих других языках в качестве символа *присваивания*, а не символа равенства, как это принято в математике. Он означает, что значение, стоящее с правой стороны от этого символа, *копируется* в переменную, стоящую в левой части. Так, присваивание `b = a` означает,

что в переменную (ячейку) с именем `b` надо скопировать значение из переменной (ячейки) с именем `a`. Поэтому неправильное с точки зрения математики выражение:

```
x = x + 1
```

в программировании вполне корректно. Оно означает, что нужно взять значение, хранящееся в ячейке с именем `x`, прибавить к нему `1` (это будет происходить где-то вне ячейки `x`), после чего получившийся результат записать в ячейку `x`, заменив им прежнее значение.

После объявления переменных их можно использовать в выражениях присваивания:

```
переменная = значение;
```

```
переменная = выражение;
```

```
переменная1 = переменная2;
```

и т. д.

Например:

```
i1 = i2 + 5 * i1;
```

Примитивными типами (primitive types) (или, что то же самое, *простыми типами*) называются такие, для которых данные содержатся в одной ячейке памяти, не имеющей вложенных ячеек.

Ссылочными типами называются такие, для которых в ячейке памяти (*ссылочной переменной*) содержатся не сами данные, а только адреса этих данных, т. е. *ссылки* на данные. При присваивании в ссылочную переменную заносится новый адрес, а не сами данные. Но непосредственного доступа к адресу, хранящемуся в ссылочных переменных, нет. Это сделано для обеспечения безопасности работы с данными — как с точки зрения устранения непреднамеренных ошибок, характерных для работы с данными по их адресам в языках C/C++/Pascal, так и для устранения возможности намеренного взлома информации.

Если *ссылочной переменной* не присвоено ссылки, то в ней хранится нулевой адрес, которому дано символическое имя `null`. Ссылки можно присваивать друг другу, если они совместимы по типам, а также присваивать им значение `null`. При этом из одной *ссылочной переменной* в другую копируется адрес. Ссылочные переменные можно проверять на равенство, в том числе на равенство `null`. При этом сравниваются не данные, а их адреса, хранящиеся в *ссылочных переменных*.

В Java все типы делятся на примитивные и ссылочные. К примитивным типам относятся следующие предопределенные типы:

- ◆ целочисленные типы `byte`, `short`, `int`, `long`, `char`;
- ◆ типы данных в формате с плавающей точкой `float`, `double`;
- ◆ булевый (булевский, логический) тип `boolean`;

Все остальные типы Java *ссылочные*.

В Java действуют следующие соглашения о регистре букв в идентификаторах:

- ◆ имена примитивных типов следует писать в нижнем регистре (строчными буквами), например: `int`, `float`, `boolean` и т. д.;

- ◆ имена ссылочных типов следует начинать с заглавной (большой) буквы, а далее для имен, состоящих из одного слова, писать все остальные буквы в нижнем регистре (например, Object, Float, Boolean, Collection, Runnable). Но если имя составное, то новую часть имени начинают с заглавной буквы (например, JButton, JTextField, JFormattedTextField, MyType и т. д.). Обратите внимание, что типы float и Float, boolean и Boolean различны — язык Java чувствителен к регистру букв;
- ◆ для переменных и методов имена, состоящие из одного слова, следует писать в нижнем регистре (например, i, j, object1). Если имя составное, новую часть имени начинают с заглавной буквы: myVariable, jButton2, jTextField2.getText() и т. д.;
- ◆ имена пакетов (см. разд. 1.8) следует писать полностью в нижнем регистре даже в случае, когда имя составное: mypackage, desktopapplication1, org.jdesktop.swingworker и т. д.;
- ◆ имена констант следует писать в верхнем регистре (большими буквами), разделяя входящие в имя составные части символом подчеркивания "_", например:


```
Double.MIN_VALUE,
Double.MAX_VALUE,
OptionPane.INFORMATION_MESSAGE,
MY_CHARS_COUNT
```

 и т. п.;
- ◆ символ подчеркивания "_" рекомендуется для разделения составных частей имени только в именах констант и пакетов.

Перечисленные соглашения о написании идентификаторов заданы в качестве стандарта языка Java, но, в отличие от соглашений о синтаксисе языка, компиляторами не отслеживаются. Хотя следование им заметно повышает читаемость программного кода, а нарушение — свидетельствует о низкой квалификации программиста.

Переменная примитивного типа может быть отождествлена с ячейкой, в которой хранятся данные. У нее всегда есть имя. Присваивание переменной примитивного типа какого-либо выражения меняет значение данных. Для ссылочных переменных при присваиваниях действия производятся с адресами ячеек, в которых хранятся данные, а не с самими данными.

Для чего нужны такие усложнения? Ведь человеку гораздо естественнее работать с ячейками памяти, в которых хранятся сами данные, а не адреса этих данных. Ответ заключается в том, что в программах часто требуются динамически создаваемые и уничтожаемые данные. Для них нельзя заранее создать необходимое число переменных, т. к. это число неизвестно на этапе написания программы и зависит от внешних параметров и выбора пользователя. Такие данные приходится помещать в динамически создаваемые и уничтожаемые ячейки. А с этими ячейками удастся работать только с помощью ссылочных переменных.

Ссылочные типы Java используются в объектном программировании (в частности, для работы со строками, файлами, элементами пользовательского интерфейса). Все *пользовательские типы* (задаваемые программистом) являются ссылочными (в том числе типы-перечисления).

1.6. Процедурное и объектно-ориентированное программирование. Инкапсуляция

Объектно-ориентированное программирование (ООП) — это методология программирования, опирающаяся на три базовых принципа:

- ◆ инкапсуляцию;
- ◆ наследование;
- ◆ полиморфизм.

Язык Java объектно-ориентированный и в полном объеме реализует эти принципы. В данном разделе рассматривается принцип инкапсуляции. Наследованию и полиморфизму посвящены отдельные разделы.

Построение программ, основанных на ООП, принципиально отличается от более ранней методики *процедурного программирования*, в которой основой служили *подпрограммы*.

Программа — это набор инструкций процессору и данных, объединенных в единую функционально законченную последовательность, позволяющую выполнять какую-нибудь конкретную деятельность.

Подпрограмма — это набор инструкций и данных, объединенных в относительно самостоятельную последовательность, позволяющую выполнять какую-нибудь конкретную деятельность внутри программы. При этом подпрограмма не может работать самостоятельно — она запускается из программы, может получать из нее данные или передавать их в программу.

Подпрограммы принято делить на подпрограммы-процедуры и подпрограммы-функции. Подпрограмму-процедуру вызывают для выполнения каких-либо действий (например, для распечатки текста на принтере). Подпрограмма-функция выполняет какие-либо действия и возвращает некоторое значение. Например, вычисляет синус и возвращает вычисленное значение или создает сложно устроенный объект и возвращает ссылку на него (адрес ячейки, в которой он находится).

Программы, написанные в соответствии с принципами процедурного программирования, состоят из набора подпрограмм, причем для решения конкретной задачи программист явно указывает на каждом шаге, *что* делать и *как* делать. Эти программы практически полностью (процентов на девяносто) состоят из решения конкретных задач.

Программы, написанные в соответствии с принципами ООП, пишут совершенно иначе. В них основное время занимает продумывание и описание того, как устроены классы. Код с описанием классов предназначен для многократного использования без внесения каких-либо изменений. При этом обеспечивается разграничение областей влияния частей программного кода друг на друга и упорядочение способов взаимодействия этих частей благодаря ограничению области видимости элементов программ. В результате только небольшая часть времени посвящается решению конкретной задачи — написанию на основе классов такого кода, который в других задачах подобного типа не применить.

Именно благодаря такому подходу объектное программирование приобрело огромную популярность — при необходимости решения сходных задач можно взять уже готовый

код, модифицируя только ту часть программы, которая относится к решению конкретной задачи.

Подробно мы разберем принципы ООП позже. Пока же в общих чертах разъясним их суть.

Самый простой из указанных в начале раздела принципов — *инкапсуляция*. Это слово в общем случае означает "заключение внутрь капсулы" (т. е. ограничение доступа к внутреннему содержимому снаружи и отсутствие такого ограничения внутри капсулы). В объектном программировании инкапсуляция означает использование *классов* — таких *типов*, в которых кроме *данных* описаны подпрограммы, позволяющие работать с этими данными, а также выполнять другие действия. Такие подпрограммы, инкапсулированные в класс, называют *методами*. Поля данных и методы, заданные в классе, часто называют *членами класса* (class members).

Класс — это описание того, как будет устроен *объект*, являющийся *экземпляром данного класса*, а также какие методы объект может вызывать. Заметим, что методы, в отличие от других подпрограмм, могут напрямую обращаться к данным своего объекта. Поскольку экземплярами классов ("воплощением" в реальность того, что описано в классе) служат объекты, то классы называют *объектными типами*.

Все объекты, являющиеся экземплярами некоторого класса, имеют одинаковые наборы полей данных (*атрибуты* объекта), но значения этих полей свои для каждого объекта. Поля данных — это переменные, заданные на уровне описания класса, а не при описании метода. В процессе жизни объекта эти значения могут изменяться. Значения полей данных объекта задают его *состояние*. А методы задают *поведение* объекта. Причем в общем случае на это поведение влияет состояние объекта — методы пользуются значениями его полей данных.

Классы в Java задают следующим образом. Сначала указывают зарезервированное слово `class`, затем имя класса, после чего в фигурных скобках записывают реализацию класса — задают его поля (*глобальные переменные*) и методы.

Объектные переменные — такие переменные, которые имеют объектный тип. В Java объектные переменные — это не сами объекты, а только *ссылки* на них (т. е. все объектные типы являются ссылочными).

Объектную переменную объявляют так же, как и другие переменные: сначала пишут тип, а затем через пробел имя объявляемой переменной.

Например, задать переменную `obj1` типа `Circle` ("окружность") можно так:

```
Circle obj1;
```

Объектную переменную связывают с объектом путем присваивания. Поскольку объектная переменная ссылочная, в правой части присваивания необходимо указать ссылочное значение: либо функцию, возвращающую ссылку на объект (адрес объекта), либо имя другой ссылочной переменной, либо ссылочную константу. Если ссылочной переменной не присвоено никакого значения, то в ней хранится значение по умолчанию — константа `null`. Объектные переменные можно проверять на равенство, в том числе на равенство `null`. При этом сравниваются не сами объекты, а их адреса, хранящиеся в объектных переменных.

Создается объект с помощью вызова специальной подпрограммы, задаваемой в классе и называемой *конструктором*. Конструктор возвращает ссылку на созданный объект. Имя конструктора в Java *всегда совпадает с именем класса*, экземпляр которого создается. Перед именем конструктора во время вызова ставится оператор `new` — "новый", означающий, что создается новый объект. Например, вызов

```
obj1 = new Circle();
```

означает, что создается новый объект типа `Circle` (окружность), и ссылка на него (адрес объекта) записывается в переменную `obj1`. Переменная `obj1` до этого уже должна быть объявлена. Оператор `new` отвечает за динамическое выделение памяти под создаваемый объект.

Часто совмещают задание объектной переменной и назначение ей объекта. В нашем случае оно будет выглядеть так:

```
Circle obj1 = new Circle();
```

У конструктора, как и у любой подпрограммы, может быть список параметров. Они нужны для того, чтобы задать начальное состояние объекта при его создании. Например, мы хотим, чтобы у создаваемой окружности можно было при вызове конструктора задать координаты x , y ее центра и радиус r . Тогда при написании класса `Circle` можно предусмотреть конструктор, в котором первым параметром задается координата x , вторым — y , третьим — радиус окружности r . Тогда задание переменной `obj1` может выглядеть так:

```
Circle obj1 = new Circle(130,120,50);
```

Эта запись означает, что создается объект-окружность с центром в точке с координатами $x = 130$, $y = 120$ и радиусом $r = 50$.

Если разработчики класса не создали ни одного конструктора, то в реализации класса автоматически создается конструктор по умолчанию, имеющий пустой список параметров. И его можно вызывать в программе так, как мы это первоначально делали для класса `Circle`.

Отметим еще одно правило, касающееся используемых имен. Как мы помним, имена объектных типов принято писать с заглавной буквы, а имена объектных переменных — со строчной. Если объектная переменная имеет тип `Circle`, то она служит ссылкой на объекты-окружности. Поэтому имя `obj1` не очень удачно — мы выбрали его только для того, чтобы подчеркнуть, что именно с помощью этой переменной осуществляется связь с объектом, и чтобы читатель не путал тип переменной, ее имя и имя конструктора. В Java принято называть объектные переменные так же, как их типы, но начинать имя со строчной буквы. Поэтому предыдущий оператор мог бы выглядеть так:

```
Circle circle = new Circle(130,120,50);
```

Если требуется работа с несколькими объектными переменными одного типа, то их принято называть в соответствии с указанным выше правилом, но добавлять порядковый номер. Следующие строки программного кода создают два независимых объекта с одинаковыми начальными параметрами:

```
Circle circle1 = new Circle(130,120,50);
```

```
Circle circle2 = new Circle(130,120,50);
```

С помощью объектных переменных осуществляется доступ к полям данных или методам объекта: сначала указывают имя переменной, затем точку, после чего пишут имя поля данных или метода. Например, если имя объектной переменной `obj1`, а имя целочисленного поля данных `x`, то присваивание ему нового значения будет выглядеть так:

```
obj1.x = 5;
```

А если имя подпрограммы `show`, у нее нет параметров, и она не возвращает никакого значения, то ее вызов будет выглядеть так:

```
obj1.show();
```

Методы делятся на *методы объектов* и *методы классов*. Чаще всего встречаются методы объектов. Они так называются потому, что пользуются полями данных объектов, и поэтому их можно вызывать только из самих объектов. Методы классов, напротив, не пользуются полями данных объектов, они могут работать при отсутствии объекта. Поэтому их можно вызывать как из классов, так и из объектов. Формат вызова:

```
имяКласса.имяМетода(список параметров)
```

или

```
имяОбъекта.имяМетода(список параметров).
```

Модификатором называется зарезервированное слово, задающее правила доступа к элементу или особенность его реализации. Его ставят перед типом и именем задаваемого поля, метода или класса. Разрешается писать подряд несколько модификаторов, поясняющих разные особенности элемента.

Про то, как задавать методы в классе, какие бывают модификаторы и какие существуют правила видимости у методов, полей данных и классов, будет рассказано в *главе 6*, где излагаются начальные сведения об объектном программировании.

При задании метода класса перед его именем необходимо поставить модификатор `static` — "статический". Мы никогда не будем употреблять для таких методов название "статические методы", так как оно крайне неудачно, а будем называть их методами класса, как это принято в теории программирования.

Точно так же переменные (поля данных) делятся на *переменные объектов* и *переменные классов*. При задании переменной класса перед ее именем необходимо поставить модификатор `static`. Переменные класса, как и методы класса, можно вызывать как из классов, так и из объектов.

Формат вызова:

```
имяКласса.имяПеременной
```

или

```
имяОбъекта.имяПеременной.
```

Не следует путать классы, объекты и объектные переменные. Класс — это тип, т. е. описание того, как устроена ячейка памяти, в которой будут располагаться поля данных объекта. Объект — это содержимое данной ячейки памяти. А в переменной объектного типа содержится адрес объекта, т. е. адрес ячейки памяти. Сказанное относится только к языкам с динамической объектной моделью, в частности к Java. Для C++ это не так.

Как уже было сказано, кроме полей данных в классе описываются методы. Несмотря на схожесть задания в классе полей и методов их реальное размещение во время работы программы отличается. Методы не хранятся в объектах, но объекты могут их вызывать. Каждый объект имеет свой комплект полей данных — "носит свои данные с собой". Если есть сотня объектов одного типа, т. е. экземпляров одного и того же класса, то в памяти компьютера будет сотня ячеек памяти, устроенных так, как это описано в классе. Причем у каждого объекта значения этих данных могут быть свои. Например, если объект является окружностью, отрисовываемой на экране, то у каждой окружности будет свой набор координат, радиусов и цветов. Если мы будем рисовать окружности с помощью метода `show()`, нет необходимости в каждом объекте хранить код этого метода — он для всех ста объектов будет одним и тем же. Поэтому методы хранятся не в объектах, а в классах.

Класс — более общая сущность, чем объект, и до того, как во время работы программы в памяти компьютера будет создан объект, сначала должен быть загружен в память соответствующий ему класс. В Java можно создавать переменные типа "класс", и с их помощью обращаться к классам так, будто это объекты особого рода. Но, в отличие от обычных объектов, такие "объекты" не могут существовать в нескольких экземплярах, и правила работы с ними принципиально отличаются от работы с объектами. Такие сущности называются *метаобъектами*.

Переменные можно объявлять либо в классе, либо в методе. В первом случае мы будем говорить, что переменная *глобальная*, т. е. является полем данных объекта. Во втором — что она *локальная*. Переменные, объявленные в классе, можно использовать внутри методов независимо от того, в каком месте исходного кода задана эта переменная — до или после объявления соответствующего метода. Считается, что область задания элементов класса предшествует области кода внутри заданных в этом классе методов.

1.7. Работа со ссылочными переменными. Сборка мусора

Объекты, как мы уже знаем, — это экземпляры ссылочных типов. Работа с переменными ссылочного и примитивного типов принципиально различается. В каждой переменной примитивного типа содержится свое значение, и его изменение не влияет на те значения, которые можно получить с помощью других переменных. Причем имя переменной примитивного типа можно рассматривать как имя ячейки с данными, и у такой ячейки может быть только одно имя, которое не может меняться по ходу работы программы. Для ссылочных переменных это не так.

Переменные ссылочного типа содержат не сами данные, а их адреса. Поэтому присваивания для таких переменных меняют адреса, но не данные. Кроме того, из-за этого под них выделяется одинаковый объем памяти независимо от типа объектов, на которые они ссылаются. А имена ссылочных переменных можно рассматривать как псевдонимы имен ячеек с данными — у одной и той же ячейки с данными может быть сколько угодно псевдонимов, т. к. адрес одной и той же ячейки можно копировать в произвольное число переменных соответствующего типа. И все они будут ссылаться на одну и ту же ячейку с данными.

В Java ссылочные переменные служат для работы с объектами: в этом языке программирования используется *динамическая объектная модель*, и все объекты создаются динамически, с явным указанием в программе момента их создания. Это отличает Java от C++, языка со *статической объектной моделью*. В C++ могут существовать как статически заданные объекты, так и динамически создаваемые. Но это не преимущество, как могло бы показаться, а проблема, т. к. в ряде случаев могут возникнуть принципиально неразрешимые ситуации.

Следует отметить, что сами объекты безымянны, и доступ к ним осуществляется только через ссылочные переменные. Часто говорят про ссылочную переменную как про сам объект, поскольку долго и неудобно произносить "объект, на который ссылается данная переменная". Но этого по мере возможности следует избегать.

Ссылочной переменной любого типа может быть присвоено значение `null`, означающее, что она никуда не ссылается. Попытка доступа к объекту через такую переменную вызовет ошибку. В Java все ссылочные переменные первоначально инициализируются значением `null` (если им не назначена ссылка на объект прямо в месте объявления). Очень часто встречающаяся ошибка — попытка доступа к полю или методу с помощью ссылочной переменной, которой не сопоставлен объект. Такая ошибка не обнаруживается на этапе компиляции и проявляется при выполнении (*Run-time error*). При этом приложение Java генерирует исключительную ситуацию (ошибку) попытки доступа к объекту через ссылку `null` с сообщением примерно такого вида:

```
Exception in thread "AWT-EventQueue-0" java.lang.NullPointerException.
```

Последовательность действий со ссылочными переменными и объектами наглядно иллюстрируют рисунки, на которых каждой такой переменной и каждому объекту сопоставлен прямоугольник — символическое изображение ячейки (рис. 1.1). Около ячейки пишут ее имя (если оно есть), а внутри — значение, содержащееся в ячейке.



Рис. 1.1. Ссылочная переменная и связанный с ней объект

Если переменная является ссылочной, из нее выходит стрелочка-ссылка. Она кончается на том объекте, на который указывает ссылка. Если в ссылочной переменной содержится значение `null` (ссылка "в никуда", то есть `адрес==0`), рисуется "висящая" короткая стрелка, у которой находится надпись `null`. Отметим, что в Java символом равенства служит `==`, а не символ `=`, который является оператором присваивания.

При использовании предлагаемых обозначений в публикациях или при обсуждениях удобно руководствоваться следующими правилами. Если ссылка "перещелкивается", то либо создается новый рисунок (в книжке), либо перечеркивается крестиком прежняя стрелочка и рисуется новая (на листе бумаги или на доске). При новом перещелкивании эта "старая" стрелка перечеркивается двумя крестиками, а "более свежая" (которая была перещелкнута) — одним крестиком, и т. д. Такая система обозначений позволяет нагляд-

но представить, что происходит при работе со ссылками, и не запутаться в том, куда они указывают и с какими ссылками в какой момент связаны динамически создаваемые объекты.

При выполнении оператора `new`, после которого указан вызов конструктора, динамически выделяется новая безымянная ячейка памяти, имеющая тип, соответствующий типу конструктора, а сам конструктор после окончания работы возвращает адрес этой ячейки. Если у нас в левой части присваивания стоит ссылочная переменная, то в результате ссылка "перещелкивается" на динамически созданную ячейку.

Рассмотрим этот процесс подробнее. Будем считать, что сначала в ячейке `circle1` типа `Circle` хранится нулевой адрес (значение ссылки равно `null`). Изобразим это как стрелку "в никуда" с надписью `null` (рис. 1.2).

Переменная `circle1` типа `Circle`

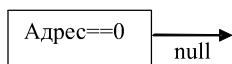


Рис. 1.2. Начальное состояние ссылочной переменной `circle1`

После оператора

```
circle1 = new Circle(x1, y1, r1);
```

в динамически выделенной безымянной ячейке памяти будет создан объект-окружность с координатами центра `x1`, `y1` и радиусом `r1` (это какие-то значения, конкретная величина которых в данном случае не важна) (рис. 1.3).

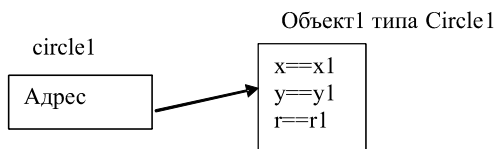


Рис. 1.3. Ссылочная переменная `circle1` и связанный с ней объект типа `Circle`

Поля объекта доступны через ссылку как по чтению, так и по записи. До тех пор пока ссылочная переменная `circle1` содержит адрес `Объект1`, имя `circle1` является псевдонимом, заменяющим имя этого объекта. Его можно использовать так же, как имя обычной переменной в любых выражениях и операторах, не связанных с изменением адреса в переменной `circle1`.

Поскольку между ссылочными переменными одного типа разрешены присваивания, переменная по ходу программы может сменить объект, на который она ссылается. Если `circle2` также имеет тип `Circle`, то допустимы присваивания вида:

```
circle2 = circle1;
```

Такие присваивания изменяют адреса в ячейках ссылочных переменных, но не меняют содержания объектов.

Рассмотрим следующий участок кода (рис. 1.4):

```
Circle circle1 = new Circle(x1, y1, r1);
Circle circle2 = new Circle(x2, y2, r2);
Circle circle3;
```

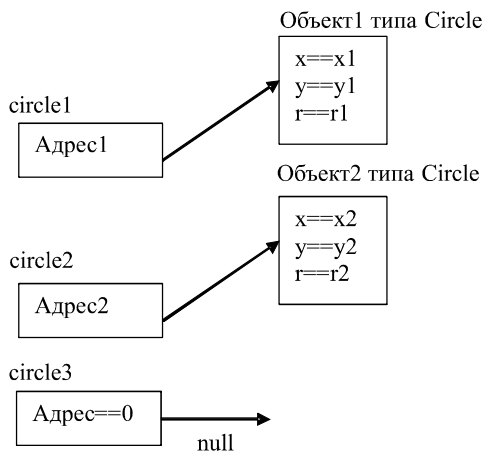


Рис. 1.4. Ссылочные переменные и связанные с ними объекты (состояние до присваивания)

Проведем присваивание:

```
circle3 = circle2;
```

В результате получится состояние переменных и объектов, изображенное на рис. 1.5.

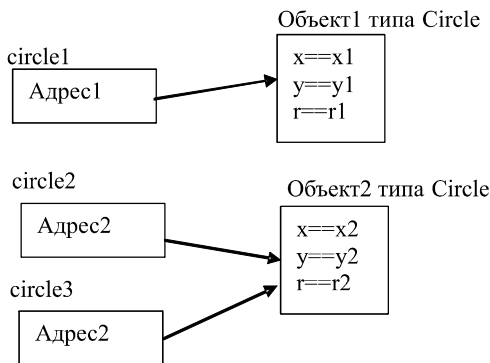


Рис. 1.5. Ссылочные переменные и связанные с ними объекты (состояние после присваивания `circle3=circle2`)

Обе переменные (как `circle2`, так и `circle3`) теперь ссылаются на один и тот же объект — в них находится один и тот же `Адрес2` (т. е. оба имени — синоним имени `Объект2`). Напомним, что сам объект, как все динамически создаваемые величины, безымянный.

Таким образом, `circle2.x` даст значение `x2`, точно так же, как и `circle3.x`. Более того, если мы изменим значение `circle2.x`, это приведет к изменению `circle3.x` — ведь это одно и то же поле `x` нашего `Объект2`.

Рассмотрим теперь, что произойдет при присваивании:

```
circle1 = circle2;
```

Этот случай (рис. 1.6) отличается от предыдущего только тем, что переменная `circle1` до присваивания уже была связана с объектом.

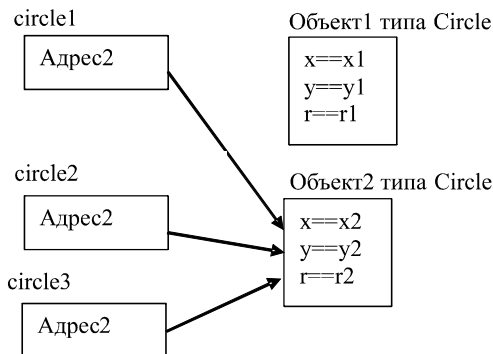


Рис. 1.6. Ссылочные переменные и связанные с ними объекты (состояние после присваивания `circle1=circle2`)

В результате у `Объект2` окажется сразу три ссылочные переменные, которые с ним связаны и имена которых являются его псевдонимами в данном месте программы: `circle1`, `circle2` и `circle3`. При этом программная связь с `Объект1` окажется утерянной — он занимает место в памяти компьютера, но программный доступ к нему невозможен, поскольку адрес этого объекта программой утерян. Таким образом, он бесполезен и напрасно занимает ресурсы компьютера.

Подобные ячейки памяти называют *мусором*. В Java предусмотрен механизм высвобождения памяти, занятой такими бесполезными объектами, — "сборка мусора" (garbage collection), работающий автоматически. Этим в фоновом режиме занимается специальная часть виртуальной Java-машины — *сборщик мусора*. При программировании на Java, в отличие от таких языков, как C/C++ или Object Pascal, программисту нет необходимости самому заботиться о высвобождении памяти, занятой под динамически создаваемые объекты.

Следует подчеркнуть, что намеренная потеря связи ссылочной переменной с ненужным уже объектом — это одно, а непреднамеренная — совсем другое. Если вы не планировали потерю связи с объектом, а она произошла, возникнет логическая ошибка. И хотя она не приведет к зависанию программы или ее неожиданному закрытию, такая программа будет работать не так, как вы предполагали, т. е. неправильно или не совсем правильно (что иногда еще опасней, т. к. ошибку можно не заметить или, заметив, очень трудно понять ее причину).

1.8. Проекты. Пакеты.

Уровни видимости классов. Импорт классов

Современное программное обеспечение построено по модульному (блочному) принципу. Программы давно перестали состоять из одного файла. Поэтому слово "программа" лучше заменять на "проект". Тем более что термин "программа", как уже говорилось, неоднозначен.

Идеология Java подразумевает работу в компьютерных сетях и возможность подгрузки в необходимый момент через сеть требуемых классов и ресурсов, в которых нуждается программа и которые не были до того загружены. Для обеспечения такого рода работы приложения Java разрабатываются и распространяются в виде большого числа независимых классов. Однако такой способ разработки приводит к чрезвычайно высокой фрагментации программы. Даже небольшие учебные проекты часто состоят из десятков классов, а реальные проекты — из сотен. При этом каждому общедоступному (public) классу соответствует свой файл, имеющий то же имя. Для того чтобы справиться с таким обилием файлов, в Java предусмотрено специальное средство группировки классов, называемое пакетом (package). Пакеты обеспечивают независимые *пространства имен* (namespaces), а также ограничение доступа к классам.

Классы всегда задают в каком-либо пакете. Пакеты могут быть вложенными с произвольным уровнем вложения (ограничивается только операционной системой и, как правило, не менее 256). Каждому пакету сопоставлена папка с исходными кодами соответствующих классов, при этом пакету, вложенному в другой, соответствует папка, вложенная в другую.

Для того чтобы поместить класс в пакет, требуется продекларировать имя пакета в начале файла, в котором объявлен класс, в следующем виде:

```
package имя_пакета;
```

Кроме того, необходимо поместить исходный код класса в папку, соответствующую пакету.

Если декларация имени пакета отсутствует, считается, что класс принадлежит пакету с именем default.

Вложенным пакетам соответствуют составные имена. Например, если мы имеем пакет с именем pkg1, в который вложен пакет с именем pkg2, в который вложен пакет с именем pkg3, то объявление, что класс с именем MyClass1 находится в пакете pkg3, будет выглядеть так:

```
package pkg1.pkg2.pkg3;
class MyClass1 {
    ....
}
```

Внутри фигурных скобок должно содержаться описание класса. Оно заменено многоточием.

Разделителем имен пакетов в программе служит точка независимо от типа операционной системы. Хотя в разных операционных системах вложенность папок будет обозначаться по-разному:

- ◆ в MS Windows — `pkg1\pkg2\pkg3\`
- ◆ в UNIX и Linux — `pkg1/pkg2/pkg3/`
- ◆ в Mac OS — `pkg1:pkg2:pkg3:`

При создании проекта в среде NetBeans класс автоматически помещается в пакет.

Не следует путать пакеты (packages) Java и модули (packs) среды NetBeans. Модули обеспечивают добавление новых возможностей среде NetBeans, а пакеты — средство группировки классов в языке Java.

При декларации класса можно указывать, что он общедоступен, с помощью *модификатора доступа* `public`:

```
public class MyClass2 {  
    ....  
}
```

При этом возможен доступ к данному классу из других пакетов.

Если же модификатор `public` отсутствует, как в случае `MyClass1`, то доступ к классу разрешен только из классов, находящихся с ним в одном пакете. Про такие файлы говорят, что у них *пакетный* вариант доступа (видимости). В C++ аналогичный вид доступа называется "дружественным" — `friend`, а в Java официальное, но не очень часто используемое название для пакетного способа видимости — `default`, то есть "видимость по умолчанию".

В файле с расширением `.java` можно располагать только один общедоступный класс и произвольное число классов с пакетным уровнем видимости.

Класс может использовать общедоступные (`public`) классы из других пакетов напрямую, с указанием полного имени класса в пространстве имен, включающего имя пакета. Например, доступ к классу `MyClass2` в таком варианте осуществляется так:

```
pkg1.pkg2.pkg3.MyClass2
```

Для того чтобы задать переменную объектного типа, нужно прежде указать ее класс. В нашем случае это будет выглядеть так:

```
pkg1.pkg2.pkg3.MyClass2 myObject;
```

Чтобы отличать имена классов от имен пакетов, в Java принято имена пакетов писать только строчными буквами, имена классов начинать с заглавной буквы, а имена полей данных (в том числе имена объектных переменных) и методов начинать со строчной буквы. Если имя класса, поля данных или метода (но не пакета!) состоит из нескольких слов, то каждое новое слово принято писать с заглавной буквы. Новое слово также можно отделять от предыдущего символом подчеркивания. Таким образом, из названия:

```
javax.swing.JMenuItem
```

понятно, что:

```
javax и swing — пакеты,
```

```
JMenuItem — имя класса.
```

Существует способ доступа к именам из другого пакета напрямую, без указания каждый раз полного пути в пространстве имен. Это делается с помощью оператора `import`. Если

мы хотим импортировать имя класса `MyClass2` из пакета `pkg3`, то после объявления имени нашего пакета (например, `myPack1`), но до объявления нашего класса (например, `MyClass3`) пишут

```
import pkg1.pkg2.pkg3.MyClass2;
```

При этом в классе `MyClass3` имя `MyClass2` можно указывать напрямую, без имени пакета перед ним: `pkg1.pkg2.pkg3`. Задание переменной будет выглядеть так:

```
MyClass2 myObject;
```

Но если мы импортируем пакеты, содержащие классы с одинаковыми именами, то требуется указание полного имени класса, квалифицированного именем пакета.

Если мы хотим импортировать имена всех классов из пакета, в операторе `import` после имени пакета вместо имени класса следует написать `*`.

Пример:

```
import pkg1.pkg2.pkg3.*;
```

Заметим, что импортируются только имена файлов, находящихся точно на уровне указанного пакета. Импорта имен из вложенных в него пакетов не происходит. Например, если записать

```
import pkg1.*;
```

или

```
import pkg1.pkg2.*;
```

то имя класса `MyClass2` не будет импортировано, поскольку он находится внутри `pkg1` и `pkg2` не непосредственно, а через пакет `pkg3`.

Имеется одно исключение из правила для импорта: классы ядра языка Java, содержащиеся в пакете `java.lang`, импортируются автоматически без указания имени пакета.

Например, объявить графический объект `g`, имеющий тип `java.awt.Graphics`, можно тремя способами:

◆ напрямую с указанием имени пакета и класса:

```
java.awt.Graphics g;
```

◆ с предварительным импортом класса `Graphics` из пакета `java.awt` и последующим указанием имени этого класса без его спецификации именем пакета:

```
import java.awt.Graphics;
...
Graphics g;
```

◆ с предварительным импортом всех классов (в том числе `Graphics`) из пакета `java.awt` и последующим указанием имени этого класса без его спецификации именем пакета:

```
import java.awt.*;
...
Graphics g;
```

Следует отметить, что обращение к переменным класса и методам класса должно идти только через имя класса. Например, константа `PI` и тригонометрические функции `sin`, `cos` и др. заданы в классе `java.lang.Math`.

Поэтому присваивание, выглядящее в математической форме записи как $y = \frac{\sin \pi x}{\pi x}$,

в программе Java в традиционном варианте выглядит очень не похоже на оригинал, приходится писать совершенно нечитаемый текст:

```
y=Math.sin(Math.PI*x)/(Math.PI*x);
```

Пакет `java.lang` и все входящие в него классы импортируются по умолчанию, поэтому нам не придется писать `java.lang.Math.PI` или `java.lang.Math.sin`. Но писать просто `PI` или `sin` раньше было невозможно, либо приходилось в каждом классе задавать константу:

```
final double PI=Math.PI;
```

и функцию:

```
double sin(double x){
    return Math.sin(x);
}
```

Если же была нужна не только функция `sin`, то приходилось перезадавать все другие математические функции. Характерная особенность Java — небольшой размер классов и методов, так что такой подход оказывался бесполезным.

Начиная с JDK 5 в Java появилась возможность *статического импорта* из пакета классов переменных класса и методов класса.

Можно импортировать как отдельные переменные или методы класса:

```
import static java.lang.Math.PI;
import static java.lang.Math.sin;
```

так и все члены класса:

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

После такого импорта выражение приобретает более читаемый вид:

```
y = sin(PI * x) / (PI * x);
```

Имена предопределенных пакетов Java начинаются с имени `java` или `javax`. Пользовательским пакетам задавать такие имена нельзя.

При работе с пакетами возникает проблема конфликта имен. Допустим, два разработчика, участвующие в одном проекте, создали пакеты с одинаковыми именами. Или же вы пытаетесь использовать библиотеки классов от двух разных производителей и обнаруживаете, что в них содержатся пакеты с одинаковыми именами, хотя эти пакеты не имеют друг к другу никакого отношения. Возникает конфликт имен, т. к. некоторые классы пакетов имеют одинаковые имена. Очевидно, что попытки замены одних осмысленных названий пакетов и содержащихся в них классов на другие не дадут никакой гарантии. Поэтому должна быть выстроена некоторая система построения уникальных имен.

В Java предлагается методика, гарантирующая отсутствие совпадений имен пакетов (а значит, и имен классов, т. к. полное имя класса состоит из имени пакета и "короткого" имени класса). Предлагается давать пакетам имена, совпадающие с доменными WWW-адресами, но записанными в обратном порядке.

Например, если у физического факультета СПбГУ доменное имя `phys.spbu.ru` (веб-адрес <http://www.phys.spbu.ru/>), то имя головного пакета для всех разработчиков с физическо-

го факультета должно быть `ru.spbu.phys`. При этом работающие на кафедре вычислительной физики (`computational physics`) сотрудники будут создавать свои пакеты и классы в пакете `ru.spbu.phys.cph`, т. к. для данной кафедры сокращенное имя в сети `cph`. А работающие на кафедре высшей математики и математической физики (`mathematical physics`) сотрудники будут создавать свои пакеты и классы в пакете `ru.spbu.phys.mph`, т. к. для данной кафедры сокращенное имя в сети `mph`.

Имя самого внешнего пакета для всех организаций России будет `ru`, для Франции — `fr` и т. д.

Поэтому, если вы будете изучать состав папок и файлов какого-нибудь пакета (например, в JDK), не удивляйтесь, если сначала будет папка, в которую вложена только одна папка, в ней — в свою очередь только одна папка, и т. д. И только в самой внутренней папке найдутся файлы классов. Такая система не только гарантирует отсутствие конфликта имен, но и дает возможность легко найти сайт разработчика и подразделение, в котором разработан соответствующий пакет.

1.9. Базовые пакеты и классы Java

Стандартные библиотеки являются гораздо более важной частью языка программирования, чем это кажется. Начинающие программисты часто сравнивают, "какой язык программирования лучше", но автору ни разу не довелось услышать при этом обсуждения достоинств и недостатков стандартных библиотек этих языков. В то же время сам по себе язык программирования — это лишь инструмент для создания программ ("строительный кран"), а "строительный материал", из которого, как из кирпичиков и блоков, создается строение (программа), — библиотеки. Конечно, и в самом языке обычно имеются конструкции, позволяющие программисту самостоятельно создавать "кирпичики" и "строительные блоки" программы. Но в нашей аналогии это скорее песок и цемент. При строительстве реального (а не игрушечного) дома они годятся только для того, чтобы создать бетон, скрепляющий готовые блоки или кирпичи. Ведь не будете же вы строить дом, разрабатывая для него каждый кирпич из отдельных песчинок!

Совершенно аналогично постройке домов идет строительство современных программных комплексов. Операторы языка программирования используются в основном для того, чтобы скреплять между собой уже готовые программные конструкции, оформленные в виде библиотек. В Java такие библиотеки оформляют в виде стандартных пакетов.

В состав JDK 1.6 входят следующие пакеты: `java`, `javax`, `org` и `com.sun`. В них, в свою очередь, находится большое число вложенных пакетов.

В табл. 1.6 приведено содержимое пакета `java`.

Таблица 1.6. Содержимое пакета `java`

Пакет, класс	Краткое описание
<code>java.applet</code>	Поддержка работы с апплетами
<code>java.awt</code>	Базовый пакет работы с графическим пользовательским интерфейсом (Abstract Window Toolkit — Абстрактный инструментальный графического окна)