Справочник по программированию на Java  
вторая часть

Методическое пособие

**Центр компьютерного обучения «Специалист», 2016**

Справочник по программированию на Java  
вторая часть

Методическое пособие

Все названия программных продуктов являются зарегистрированными торговыми марками соответствующих фирм. Никакая часть настоящего издания ни в каких целях не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами, будь то электронные или механические, включая фотокопирование и запись на электронный носитель, если на это нет письменного разрешения автора.

© Центр компьютерного обучения «Специалист» при МГТУ им. Н.Э. Баумана, 2016 г.

Оглавление

[Многопоточное программирование 5](#_Toc457601008)

[Модель потоков Java 6](#_Toc457601009)

[Приоритеты потоков 7](#_Toc457601010)

[Синхронизация 8](#_Toc457601011)

[Обмен сообщениями 8](#_Toc457601012)

[Класс Thread и интерфейс Runnable 9](#_Toc457601013)

[Главный поток 9](#_Toc457601014)

[Создание потока 11](#_Toc457601015)

[Реализация Runnable 11](#_Toc457601016)

[Расширение Thread 13](#_Toc457601017)

[Создание множества потоков 14](#_Toc457601018)

[Использование isAlive() и join() 15](#_Toc457601019)

[Приоритеты потоков 15](#_Toc457601020)

[Синхронизация 18](#_Toc457601021)

[Использование синхронизированных методов 19](#_Toc457601022)

[Оператор synchronized 21](#_Toc457601023)

[Межпотоковые коммуникации 22](#_Toc457601024)

[Взаимная блокировка 26](#_Toc457601025)

[Приостановка, возобновление и останов потоков 28](#_Toc457601026)

[Использование многопоточности 31](#_Toc457601027)

[Класс Applet 31](#_Toc457601028)

[Два типа аплетов 31](#_Toc457601029)

[Основы аплетов 32](#_Toc457601030)

[Класс Applet 32](#_Toc457601031)

[Архитектура аплетов 33](#_Toc457601032)

[Введение в Swing 35](#_Toc457601033)

[Происхождение и философия Swing 35](#_Toc457601034)

[Компоненты и контейнеры 38](#_Toc457601035)

[Компоненты 38](#_Toc457601036)

[Контейнеры 38](#_Toc457601037)

[Панели контейнеров верхнего уровня 39](#_Toc457601038)

[Менеджеры компоновки 39](#_Toc457601039)

[Простое Swing-приложение 40](#_Toc457601040)

[Обработка событий 45](#_Toc457601041)

[Создание Swing-аплета 48](#_Toc457601042)

[Введение в JavaFX 50](#_Toc457601043)

[Базовые понятия JavaFX 50](#_Toc457601044)

[Пакеты JavaFX 51](#_Toc457601045)

[Классы Stage и Scene 51](#_Toc457601046)

[Узлы и графы сцены 51](#_Toc457601047)

[Панели компоновки 52](#_Toc457601048)

[Класс Application и жизненный цикл приложения 52](#_Toc457601049)

[Запуск приложения JаvаFХ 53](#_Toc457601050)

[Каркас приложения JаvаFХ 53](#_Toc457601051)

[Поток выполнения приложения 56](#_Toc457601052)

[Использование кнопок и событий 57](#_Toc457601053)

[Основные сведения о событиях 57](#_Toc457601054)

[Сервлеты 58](#_Toc457601055)

[Предварительные сведения 58](#_Toc457601056)

[Жизненный цикл сервлета 59](#_Toc457601057)

# Многопоточное программирование

В отличие от многих других языков программирования, Java предлагает встроенную поддержку многопоточного программирования. Многопоточная программа содержит две или более частей, которые могут выполняться одновременно. Каждая часть такой программы называется потоком (thread) или нитью, и каждый поток задает отдельный путь выполнения. То есть, многопоточность – это специализированная форма многозадачности.

Вы почти наверняка знакомы с многозадачностью, поскольку она поддерживается практически всеми современными операционными системами. Однако существуют два отдельных типа многозадачности: многозадачность, основанная на процессах, и многозадачность, основанная на потоках. Важно понимать разницу между ними. Большинству читателей многозадачность, основанная на процессах, является более знакомой формой.

Процесс по сути своей – это выполняющаяся программа. То есть многозадачность, основанная на процессах, представляет собой средство, которое позволяет вашему компьютеру одновременно выполнять две или более программ. Так, например, процессная многозадачность позволяет запускать компилятор Java в то самое время, когда вы используете текстовый редактор. В многозадачности, основанной на процессах, программа представляет собой наименьший элемент кода, которым может управлять планировщик операционной системы.

В среде поточной многозадачности наименьшим элементом управляемого кода является поток. Это означает, что одна программа может выполнять две или более задач одновременно. Например, текстовый редактор может форматировать текст в то же время, когда выполняется его печать – до тех пор, пока эти два действия выполняются двумя отдельными потоками. То есть многозадачность на основе процессов имеет дело с “картиной в целом”, а потоковая многозадачность справляется с деталями.

Многозадачные потоки требуют меньше накладных расходов, чем многозадачные процессы. Процессы – это тяжеловесные задачи, каждая из которых требует своего собственного адресного пространства. Межпроцессные коммуникации дорогостоящи и ограничены. Переключение контекста от одного процесса к другому также обходится дорого. С другой стороны, потоки являются облегченными. Они разделяют одно и тоже адресное пространство и совместно используют один и тот же тяжеловесный процесс.

Коммуникации между потоками являются экономными, а переключения контекста между потоками характеризуется низкой стоимостью. Хотя Java-программы используются в средах процессной многозадачности, многозадачность, основанная на процессах, средствами Java не управляется. А вот многопоточная многозадачность средствами Java управляется.

Многопоточность позволяет вам писать очень эффективные программы, которые по максимуму используют центральный процессор, поскольку время ожидания может быть сведено к минимуму. Это особенно важно для интерактивных сетевых сред, в которых работает Java, так как в них наличие ожидания и простоев – обычное явление. Например, скорость передачи данных по сети намного ниже, чем скорость, с которой компьютер может их обрабатывать. Даже ресурсы локальной файловой системы читаются и пишутся намного медленнее, чем темп их обработки в процессоре. И, конечно, ввод пользователя намного медленнее, чем компьютер. В однопоточных средах ваша программа вынуждена ожидать окончания таких задач, прежде чем переходить к следующей – даже если центральный процессор большую часть времени простаивает. Многопоточность позволяет получить доступ к этому времени ожидания и использовать его рациональным образом.

Если вы программировали для таких операционных систем, как Windows, это значит, что вы уже знакомы с многопоточным программированием. Однако тот факт, что Java управляет потоками, делает многопоточность особенно удобной, поскольку многие детали подконтрольны вам как программисту.

## Модель потоков Java

Система времени выполнения Java зависит от потоков во многих отношениях, и все библиотеки классов спроектированы с учетом многопоточности. Фактически Java использует потоки для того, чтобы обеспечить асинхронность всей среде выполнения. Это позволяет снизить неэффективность за счет предотвращения холостой растраты циклов центрального процессора.

Значение многопоточной среды лучше понимается при сравнении с ее противоположностью. Однопоточные системы используют подход, называемый циклом событий с опросом. В этой модели единственный поток управления выполняется в бесконечном цикле, опрашивая единственную очередь событий, чтобы принять решение о том, что делать дальше. Как только этот механизм опроса возвращает, скажем, сигнал о том, что сетевой файл готов к чтению, цикл событий передает управление соответствующему обработчику событий. До тех пор, пока тот не вернет управление, в системе ничего не может произойти. Это расходует время процессора. Это также может привести к тому, что одна часть программы будет доминировать над другими и не давать возможности обрабатывать любые другие события. Вообще говоря, в однопоточном окружении, когда поток блокируется (то есть приостанавливает выполнение) по причине ожидания некоторого ресурса, выполнение всей программы приостанавливается.

Выгода от многопоточности состоит в том, что основной механизм циклического опроса исключается. Один поток может быть приостановлен без остановки других частей программы. Например, время ожидания при чтении данных из сети либо ожидание пользовательского ввода может быть утилизировано где угодно. Многопоточность позволяет циклам анимации “засыпать” на секунду между показом соседних кадров, не приостанавливая работы всей системы. Когда поток блокируется в программе Java, то останавливается только один-единственный заблокированный поток. Все остальные потоки продолжают выполняться.

Потоки существуют в нескольких состояниях. Поток может выполняться. Он может быть готов к выполнению, как только получит время центрального процессора. Работающий поток может быть приостановлен, что временно прекращает его активность. Выполнение приостановленного потока может быть возобновлено, позволяя ему продолжить работу с того места, где он был приостановлен. Поток может быть заблокирован, когда ожидает какого-то ресурса. В любой момент поток может быть прерван, что немедленно останавливает его выполнение. Однажды прерванный поток уже не может быть возобновлен.

## Приоритеты потоков

Java присваивает каждому потоку приоритет, который определяет поведение данного потока по отношению к другим. Приоритеты потоков задаются целыми числами, определяющими относительный приоритет одного потока по сравнению к другими. Значение приоритета само по себе никакого смысла не имеет – более высокоприоритетный поток не выполняется быстрее, чем низкоприоритетный, когда он является единственным исполняемым потоком в данный момент. Вместо этого приоритет потока используется для принятия решения при переключении от одного выполняющегося потока к другому. Это называется переключением контекста. Правила, которые определяют, когда должно происходить переключение контекста, достаточно просты.

Поток может добровольно уступить управление. Это делается явным уступанием очереди выполнения, приостановкой или блокированием ожидания ввода-вывода.

При таком сценарии все прочие потоки проверяются, и ресурсы процессора передаются потоку с максимальным приоритетом, который готов к выполнению.

Поток может быть прерван другим, более приоритетным потоком. В этом случае низкоприоритетный поток, который не занимает процессор, просто приостанавливается высокоприоритетным потоком, независимо от того, что он делает. В основном, высокоприоритетный поток выполняется, как только он этого “захочет”. Это называется вытесняющей многозадачностью (или многозадачностью с приоритетами).

В случае, когда два потока, имеющие одинаковый приоритет, претендуют на цикл процессора, ситуация усложняется. Для таких операционных систем, как Windows, потоки с одинаковым приоритетом разделяют время в циклическом режиме. Для операционных систем других типов потоки с одинаковым приоритетом должны принудительно передавать управление своим “родственникам”. Если они этого не делают, другие потоки не запускаются.

Внимание! Из-за разницы в способах переключения операционными системами потоковых контекстов могут возникать проблемы переносимости.

## Синхронизация

Поскольку многопоточность дает вашим программам возможность асинхронного поведения, должен существовать способ обеспечить синхронизацию, когда в этом возникает необходимость. Например, если вы хотите, чтобы два потока взаимодействовали и разделяли сложную структуру данных, такую как связный список, то вы нуждаетесь в каком-то способе исключения конфликтов между ними. То есть вы должны предотвратить запись данных в одном потоке, когда другой занимается их чтением. Для этой цели в Java реализован элегантный трюк из старой модели межпроцессной синхронизации, а именно – монитор. Монитор – это управляющий механизм, впервые реализованный Чарльзом Энтони Ричардом Хоаром. Вы можете воспринимать монитор как очень маленький ящик, который принимает только один поток в единицу времени. Как только поток вошел в монитор, все другие потоки должны ждать, пока тот не покинет его. Таким образом, монитор может быть использован для защиты разделяемых ресурсов от одновременного использования более чем одним потоком.

Большинство многопоточных систем применяют мониторы как объекты, которые ваша программа может получить и которыми она может манипулировать. Java предлагает более чистое решение. Не существует отдельного класса монитора вроде “Monitor”.

Вместо этого каждый объект имеет свой собственный неявный монитор, вход в который осуществляется автоматически, когда вызывается синхронизированный метод объекта.

Когда поток находится внутри синхронизированного метода, ни один другой поток не может вызвать никакого синхронизированного метода этого объекта. Это позволяет вам писать очень ясный и краткий многопоточный код, поскольку поддержка синхронизации встроена в язык.

## Обмен сообщениями

После того, как вы разделите вашу программу на отдельные потоки, вам нужно определить, как они будут общаться друг с другом. При программировании на большинстве других языков для установки взаимодействия между потоками вы должны зависеть от операционной системы. То есть, конечно же, появляются накладные расходы. В отличие от них, Java предоставляет ясный и экономичный способ общения двух или более потоков между собой – посредством вызова предопределенных методов, которыми обладают объекты. Система сообщений Java позволяет потоку войти в синхронизированный метод объекта и затем ожидать, пока какой-то другой поток явно не уведомит его о прибытии.

## Класс Thread и интерфейс Runnable

Многопоточная система Java встроена в класс Thread, его методы и дополняющий его интерфейс Runnable. Thread инкапсулирует поток выполнения. Поскольку вы не можете напрямую обратиться к нематериальному состоянию работающего потока, вы имеете дело с его заместителем (proxy) – экземпляром класса Thread, который породил его. Чтобы создать новый поток, ваша программа должна либо расширить Thread, либо реализовать интерфейс Runnable.

В остатке этой главы объясняется, как применять Thread и Runnable для создания и управления потоками, начиная с потока, который есть в каждой программе Java – главного.

## Главный поток

Когда Java-программа стартует, немедленно начинает выполняться один поток. Обычно его называют главным потоком программы, потому что это тот поток, который запускается вместе с вашей программой. Главный поток важен по двум причинам.

Это поток, от которого порождаются все “дочерние” потоки.

Часто он должен быть последним потоком, завершающим выполнение, так как он предпринимает различные завершающие действия.

Несмотря на то что главный поток создается автоматически при запуске программы, им можно управлять через объект Thread. Чтобы делать это, вы должны получить ссылку на него вызовом метода currentThread(), который является общедоступным статическим (public static) методом класса Thread. Его общая форма выглядит следующим образом:

static Thread currentThread()

Этот метод возвращает ссылку на поток, из которого он был вызван. Получив ссылку на главный поток, вы можете управлять им точно так же, как любым другим.

Рассмотрим следующий пример:

// Управление главным потоком.

class CurrentThreadDemo {

public static void main(String args[]) {

Thread t = Thread.currentThread();

System.out.println("Текущий поток: " + t);

// изменить имя потока

t.setName("Мой Thread");

System.out.println("После изменения имени: " + t);

try {

for(int n = 5; n > 0; n--) {

System.out.println(n);

Thread.sleep(1000);

}

} catch (InterruptedException e) {

System.out.println("Главный поток прерван");

}

}}

В этой программе ссылка на текущий поток (в данном случае – главный) получается вызовом currentThread(), и эта ссылка сохраняется в локальной переменной t. Далее программа отображает информацию о потоке. Программа вызывает setName() для изменения внутреннего имени потока. После этого информация о потоке отображается заново. Далее в цикле печатается обратный отсчет с задержкой на 1 секунду после каждой строки. Пауза организуется вызовом метода sleep(). Аргумент sleep() задает период задержки в миллисекундах. Обратите внимание на блок try/catch вокруг цикла. Метод sleep() в Thread может возбудить InterruptException. Это может произойти, если некоторый другой поток захочет прервать выполнение этого спящего потока. Этот пример просто печатает сообщение, если поток прерывается. В реальных программах вы будете обрабатывать подобную ситуацию иначе.

Давайте поближе взглянем на методы, определенные в Thread, которые используются в программе. Метод sleep() заставляет поток, из которого он был вызван, приостановить выполнение на указанное количество миллисекунд. Его общая форма выглядит так:

static void sleep(long миллисекунды) throws InterruptedException

Количество миллисекунд, на которое нужно приостановить выполнение, передается в параметре миллисекунды. Этот метод может возбуждать исключение InterruptedException.

Метод sleep() имеет также вторую форму, показанную ниже, который позволяет задать период в миллисекундах и наносекундах:

static void sleep(long миллисекунды, long наносекунды) throws  
InterruptedException

Вторая форма может применяться только в средах, которые предусматривают задание временных периодов в наносекундах.

Как показано в предыдущей программе, вы можете установить имя потока, используя setName(). Получить имя потока можно вызовом getName() (эта процедура в программе не показана). Эти методы являются членами класса Thread и объявлены следующим образом:

final void setName(String имя\_потока)

final String getName()

Здесь имя\_потока указывает имя потока.

## Создание потока

В наиболее общем смысле вы создаете поток, реализуя объект класса Thread. В Java определены два способа, какими это можно сделать.

* Реализуя интерфейс Runnable.
* Расширяя класс Thread.

## Реализация Runnable

Самый простой способ создания потока – это объявление класса, реализующего интерфейс Runnable. Runnable абстрагирует единицу исполняемого кода. Вы можете конструировать поток из любого объекта, реализующего интерфейс Runnable. Чтобы реализовать Runnable, класс должен объявить единственный метод run():

public void run()

Внутри run() вы определяете код, который, собственно, составляет новый поток.

Важно понимать, что run() может вызывать другие методы, использовать другие классы, объявлять переменные – точно так же, как это делает главный поток. Единственным отличием является то, что run() устанавливает точку входа для другого, параллельного потока внутри вашей программы. Этот поток завершится, когда run() вернет управление.

После того как будет объявлен класс, реализующий интерфейс Runnable, вы создадите объект типа Thread из этого класса. В Thread определено несколько конструкторов.

Тот, который должен использоваться в данном случае, выглядит следующим образом:

Thread(Runnable объект\_потока, String имя\_потока)

В этом конструкторе объект\_потока – это экземпляр класса, реализующего интерфейс Runnable. Он определяет, где начнется выполнение потока. Имя нового потока передается в имя\_потока.

После того, как новый поток будет создан, он не запускается до тех пор, пока вы не вызовете метод start(), объявленный в классе Thread. По сути, start() выполняет вызов run(). Метод start() показан ниже:

void start()

Рассмотрим пример, создающий новый поток и запускающий его выполнение:

// Создание второго потока.

class NewThread implements Runnable {

Thread t;

NewThread() {

// Создать новый, второй поток

t = new Thread(this, "Демонстрационный поток");

System.out.println("Дочерний поток создан: " + t);

t.start(); // Запустить поток

}

// Точка входа второго потока.

public void run() {

try {

for(int i = 5; i > 0; i--) {

System.out.println("Дочерний поток: " + i);

Thread.sleep(500);

}

} catch (InterruptedException e) {

System.out.println("Дочерний поток прерван.");

}

System.out.println("Дочерний поток завершен");

}}

class ThreadDemo {

public static void main(String args[]) {

new NewThread(); // создать новый поток

try {

for(int i = 5; i > 0; i--) {

System.out.println("Главный поток: " + i);

Thread.sleep(1000);

}

} catch (InterruptedException e) {

System.out.println("Главный поток прерван.");

}

System.out.println("Главный поток завершен.");

}}

Внутри конструктора NewThread в следующем операторе создается новый объект Thread:

t = new Thread(this, "Демонстрационный поток");

Передача this в первом аргументе означает, что вы хотите, чтобы новый поток вызвал run() метод объекта this. Далее вызывается start(), чем запускается выполнение потока, начиная с метода run(). Это запускает цикл for дочернего потока. После вызова start() конструктор NewThread возвращает управление main(). Когда главный поток продолжает свою работу, он входит в свой цикл for. После этого оба потока выполняются параллельно, разделяя ресурсы центрального процессора, вплоть до завершения своих циклов.

Как уже упоминалось ранее, в многопоточной программе часто главный поток должен завершать выполнение последним. Фактически, для некоторых старых виртуальных машин Java (JVM), если главный поток завершается до завершения дочерних потоков, то исполняющая система Java может “зависнуть”. Предыдущая программа гарантирует, что главный поток завершится последним, поскольку главный поток “спит” 1000 миллисекунд между итерациями цикла, а дочерний поток “спит” только 500 миллисекунд. Это заставляет дочерний поток завершиться раньше главного. Но далее вы узнаете лучший способ ожидания завершения потоков.

## Расширение Thread

Второй способ создания потока – это объявить класс, расширяющий Thread, а затем создать экземпляр этого класса. Расширяющий класс обязан переопределить метод run(), который является точкой входа для нового потока. Он также должен вызвать start() для запуска выполнения нового потока. Ниже приведен пример предыдущей программы, переписанной с использованием расширения Thread.

// Создание второго потока расширением Thread

class NewThread extends Thread {

NewThread() {

// Создать новый второй поток

super("Демонстрационный поток");

System.out.println("Дочерний поток: " + this);

start(); // Запустить поток

}

// Точка входа второго потока.

public void run() {

try {

for(int i = 5; i > 0; i--) {

System.out.println("Дочерний поток: " + i);

Thread.sleep(500);

}

} catch (InterruptedException e) {

System.out.println("Дочерний поток прерван.");

}

System.out.println("Дочерний поток завершен.");

}}

class ExtendThread {

public static void main(String args[]) {

new NewThread(); // Создать новый поток

try {

for(int i = 5; i > 0; i--) {

System.out.println("Главный поток: " + i);

Thread.sleep(1000);

}

} catch (InterruptedException e) {

System.out.println("Главный поток прерван.");

}

System.out.println("Главный поток завершен.");

}}

Эта программа генерирует точно такой же вывод, что и предыдущая версия. Как вы можете видеть, дочерний поток создается при конструировании объекта NewThread, который наследуется от Thread.

Обратите внимание на super() внутри NewThread. Он вызывает следующую форму конструктора Thread:

public Thread(String имя\_потока)

Здесь имя\_потока указывает имя потока.

## Создание множества потоков

До сих пор вы использовали только два потока: главный и один дочерний. Однако ваша программа может порождать столько потоков, сколько необходимо. Например, в следующей программе создаются три дочерних потока.

// Создание множества потоков.

class NewThread implements Runnable {

String name; // имя потока

Thread t;

NewThread(String threadname) {

name = threadname;

t = new Thread(this, name);

System.out.println("Новый поток: " + t);

t.start(); // запустить поток

}

// Входная точка потока.

public void run() {

try {

for(int i = 5; i > 0; i--) {

System.out.println(name + ": " + i);

Thread.sleep(1000);

}

} catch (InterruptedException e) {

System.out.println(name + " прерван");

}

System.out.println(name + " завершен.");

}}

class MultiThreadDemo {

public static void main(String args[]) {

new NewThread("Один"); // запуск потоков

new NewThread("Два");

new NewThread("Три");

try {

// ожидание завершения других потоков

Thread.sleep(10000);

} catch (InterruptedException e) {

System.out.println("Главный поток прерван");

}

System.out.println("Главный поток завершен.");

}}

Как видите, будучи запущенными, все три дочерних потока разделяют ресурс центрального процессора. Обратите внимание на вызов sleep(10000) в main(). Это заставляет главный поток “уснуть” на 10 секунд и гарантирует, что он будет завершен последним.

## Использование isAlive() и join()

Как упоминалось, часто необходимо, чтобы главный поток завершался последним.

В предыдущих примерах это обеспечивается вызовом sleep() из main() с задержкой, достаточной для того, чтобы гарантировать, что все дочерние потоки завершатся раньше главного. Однако это неудовлетворительное решение, которое вызывает серьезный вопрос: как один поток может знать о том, что другой завершился? К счастью, Thread предлагает средство, которое дает ответ на этот вопрос.

Существуют два способа определить, что поток был завершен. Во-первых, вы можете вызвать метод isAlive() для этого потока. Этот метод определен в классе Thread и его общая форма такова:

final Boolean isAlive()

Метод isAlive() возвращает true, если поток, для которого он вызван, еще выполняется. В противном случае он возвращает false.

В то время как isAlive() применяется изредка, существует метод, который вы будете использовать чаще, чтобы дождаться завершения потока, а именно – join(), показанный ниже:

final void join() throws InetrruptedException

Этот метод ожидает завершения потока, для которого он вызван. Его имя отражает концепцию, что вызывающий поток ожидает, когда указанный поток присоединиться к нему. Дополнительные формы join() позволяют указывать максимальный период времени, которое вы будете ожидать завершения указанного потока.

## Приоритеты потоков

Приоритеты потоков используются планировщиком потоков для принятия решений о том, когда каждому из потоков будет разрешено работать. Теоретически высокоприоритетные потоки получают больше времени процессора, чем низкоприоритетные.

Практически объем времени процессора, который получает поток, часто зависит от нескольких факторов помимо его приоритета. (Например, то, как операционная система реализует многозадачность, может влиять на относительную доступность времени процессора). Высокоприоритетный поток может также выгружать низкоприоритетный.

Например, когда низкоприоритетный поток работает, а высокоприоритетный собирается продолжить свою прерванную работу (в связи с приостановкой или ожиданием завершения операции ввода-вывода), то последний выгружает низкоприоритетный поток.

Теоретически потоки с равным приоритетом должны получать равный доступ к центральному процессору. Но вы должны быть осторожны. Помните, что Java спроектирована для работы в широком спектре сред. Некоторые из этих сред реализуют многозадачность принципиально отлично от других. В целях безопасности потоки, которые разделяют один и тот же приоритет, должны получать управление в равной степени. Это гарантирует, что все потоки получат возможность выполняться в среде операционных систем с не вытесняющей многозадачностью. На практике, даже в средах с не вытесняющей многозадачностью большинство потоков все-таки имеют шанс выполняться, поскольку большинство потоков неизбежно сталкиваются с блокирующими ситуациями, такими как ожидание ввода-вывода. Когда подобное случается, заблокированный поток приостанавливается, и остальные потоки могут работать. Но если вы хотите добиться гладкой многопоточной работы, то не должны полагаться на это. К тому же некоторые типы задач интенсивно нагружают процессор. Такие потоки захватывают процессор. Потокам такого типа вы должны передавать управление от случая к случаю, чтобы дать возможность выполняться другим.

Чтобы установить приоритет потока, используйте метод setPriority(), который является членом класса Thread. Так выглядит его общая форма:

final void setPriority(int уровень)

Здесь уровень специфицирует новый уровень приоритета для вызывающего потока.

Значение уровень должно быть в пределах диапазона от MIN\_PRIORITY до MAX\_PRIORITY.

В настоящее время эти значения равны соответственно 1 и 10. Чтобы вернуть поток приоритет по умолчанию, укажите NORM\_PRIORITY, который в настоящее время равен 5.

Эти приоритеты определены как статические финальные (static final) переменные в классе Thread.

Вы можете получить текущее значение приоритета потока, вызвав метод getPriority() класса Thread, как показано ниже:

final int getPriority()

Реализации Java могут иметь принципиально разное поведение в том, что касается планирования потоков. Версия для Windows XP/98/NT/2000 работает более-менее ожидаемым образом. Однако другие версии могут работать несколько иначе. Большинство несовпадений возникают, когда вы полагаетесь на вытесняющую многозадачность вместо совместного использования времени процессора. Наиболее безопасный способ получить предсказуемое межплатформенное поведение Java – это использовать потоки, которые принудительно осуществляют управление центральным процессором.

В следующем примере демонстрируются два потока с разными приоритетами, которые выполняются на платформе без вытесняющей многозадачности иначе, чем на платформе с упомянутой многозадачностью. Один поток получает приоритет на два уровня выше нормального, как определено Thread.NORM\_PRIORITY, а другой – на два уровня ниже нормального. Потоки стартуют и готовы к выполнению в течение 10 секунд. Каждый поток выполняет цикл, подсчитывающий количество итераций. Через 10 секунд главный поток останавливает оба потока. Затем количество итераций цикла, которое успел выполнить каждый поток, отображается.

// Демонстрация приоритетов потоков.

class clicker implements Runnable {

long click = 0;

Thread t;

private volatile boolean running = true;

BookNew\_JAVA-7.indb 266 02.06.2007 1:07:02

Глава 11. Многопоточное программирование 267

public clicker(int p) {

t = new Thread(this);

t.setPriority(p);

}

public void run() {

while (running) {

click++;

}

}

public void stop() {

running = false;

}

public void start() {

t.start();

}}

class HiLoPri {

public static void main(String args[]) {

Thread.currentThread().setPriority(Thread.MAX\_PRIORITY);

clicker hi = new clicker(Thread.NORM\_PRIORITY + 2);

clicker lo = new clicker(Thread.NORM\_PRIORITY – 2);

lo.start();

hi.start();

try {

Thread.sleep(10000);

} catch (InterruptedException e) {

System.out.println("Главный поток прерван.");

}

lo.stop();

hi.stop();

// Ожидание 10 секунд до прерывания.

try {

hi.t.join();

lo.t.join();

} catch (InterruptedException e) {

System.out.println("Перехвачено исключение InterruptedException");

}

System.out.println("Низкоприоритетный поток: " + lo.click);

System.out.println("Высокоприоритетный поток: " + hi.click);

}}

Вывод этой программы при запуске под Windows показывает, что потоки осуществляли переключение контекста, хотя не было никакого принудительного захвата процессора и никаких блокирующих операций ввода-вывода. Высокоприоритетный поток получил большую часть времени процессора.

Конечно, точный вывод, порождаемый этой программой, зависит от скорости вашего процессора и количества задач, выполняемых в системе. Когда та же программа запускается в среде с не вытесняющей многозадачностью, получается другой результат.

Еще одно замечание относительно предыдущей программы. Обратите внимание, что переменной running предшествует слово volatile. Хотя volatile более подробно объясняется в предыдущем курсе, оно используется здесь, чтобы гарантировать, что значение running будет проверяться на каждом шаге итераций цикла.

Без указания volatile Java имеет возможность оптимизировать цикл таким образом, что будет создана локальная копия running. Применение volatile предотвращает эту оптимизацию, сообщая Java, что running может изменяться неявным для кода образом.

## Синхронизация

Когда два или более потоков имеют доступ к одному разделенному ресурсу, они нуждаются в обеспечении того, что ресурс будет использован только одним потоком одновременно. Процесс, с помощью которого это достигается, называется синхронизацией. Как вы увидите, Java предлагает ее уникальную поддержку на уровне языка.

Ключом к синхронизации является концепция монитора (также называемого семафором). Монитор – это объект, который используется, как взаимное исключение (mutually exclusive lock – mutex), или мьютекс. Только один поток одновременно может владеть монитором. Когда поток запрашивает блокировку, говорят, что он входит в монитор. Все другие потоки, которые пытаются войти в заблокированный монитор, будут приостановлены до тех пор, пока первый поток не выйдет из монитора. Обо всех этих прочих потоках говорят, что они ожидают монитора. Поток, который владеет монитором, может повторно войти в него, если пожелает.

Если вы имели дело с синхронизацией в других языках, таких как C или C++, то знаете, что использовать ее не просто. Это потому, что эти языки сами по себе не поддерживают синхронизацию. Вместо этого, чтобы синхронизировать потоки, ваша программа должна использовать примитивы операционной системы. К счастью, поскольку Java реализует синхронизацию через языковые элементы, большая часть сложности, ассоциированная с синхронизацией, исчезает.

Вы можете синхронизировать ваш код двумя способами. Оба предусматривают использование ключевого слова synchronized, и оба способа мы здесь рассмотрим.

## Использование синхронизированных методов

Синхронизация в Java проста, поскольку объекты имеют собственные, ассоциированные с ними неявные мониторы. Чтобы войти в монитор объекта, следует просто вызвать метод, модифицированный ключевым словом synchronized. Когда поток находится внутри синхронизированного метода, все другие потоки, которые пытаются вызвать его (или любые другие синхронизированные методы) на том же экземпляре, должны ожидать.

Чтобы выйти из монитора и передать управление объектом другому ожидающему потоку, владелец монитора просто возвращает управление из синхронизированного метода.

Чтобы понять необходимость синхронизации, давайте начнем с простого примера, который не использует ее, хотя и должен. Следующая программа содержит три простых класса. Первый из них, Callme, имеет единственный метод – call(). Этот метод принимает параметр типа String по имени msg. Этот метод пытается напечатать строку msg внутри квадратных скобок. Интересно отметить, что после того, как call() печатает открывающую скобку и строку msg, он вызывает Thread.sleep(1000), который приостанавливает текущий поток на одну секунду.

Конструктор следующего класса, Caller, принимает ссылку на экземпляр класса Callme и String, которые сохраняются соответственно в target и msg. Конструктор также создает новый поток, который вызовет метод run() объекта. Поток стартует немедленно. Метод run() класса Caller вызывает метод call() на экземпляре target класса Callme, передавая ему строку msg. Наконец, класс Synch начинает с создания единственного экземпляра Callme и трех экземпляров Caller, каждый с уникальной строкой сообщения. Один экземпляр Callme передается каждому Caller.

// Эта программа не синхронизирована.

class Callme {

void call(String msg) {

System.out.print("[" + msg);

try {

Thread.sleep(1000);

} catch(InterruptedException e) {

System.out.println("Прервано");

}

System.out.println("]");

}}

class Caller implements Runnable {

String msg;

Callme target;

Thread t;

public Caller(Callme targ, String s) {

target = targ;

msg = s;

t = new Thread(this);

t.start();

}

public void run() {

target.call(msg);

}}

class Synch {

public static void main(String args[]) {

Callme target = new Callme();

Caller ob1 = new Caller(target, "Добро пожаловать");

Caller ob2 = new Caller(target, "в синхронизированный");

Caller ob3 = new Caller(target, "мир!");

// wait for threads to end

try {

ob1.t.join();

ob2.t.join();

ob3.t.join();

} catch(InterruptedException e) {

System.out.println("Прервано");

}

}}

Как видите, вызывая sleep(), метод call() позволяет переключиться на выполнение другого потока. Это приводит к смешанному выводу трех строк сообщений. В этой программе нет ничего, что предотвращает вызов потоками одного и того же метода на одном и том же объекте в одно и то же время. Это называется состоянием гонок, поскольку три потока соревнуются друг с другом в окончании выполнения метода. Этот пример использует sleep(), чтобы сделать эффект повторяемым и наглядным. В большинстве ситуаций этот эффект более неуловим и менее предсказуем, поскольку вы не можете предвидеть, когда произойдет переключение контекста. Это может привести к тому, что программа один раз отработает правильно, а другой раз – нет.

Чтобы исправить эту программу, вы должны сериализировать доступ к call(). То есть вы должны разрешить доступ к этому методу одновременно только одному потоку.

Чтобы сделать это, вам нужно просто предварить объявление call() ключевым словом synchronized, как показано ниже:

class Callme {

synchronized void call(String msg) {

...

Это предотвратит доступ другим потокам к call(), когда один из них уже использует его.

Всякий раз, когда у вас есть метод, или группа методов, которые манипулируют внутренним состоянием объекта в многопоточной среде, вы должны использовать ключевое слово synchronized, чтобы исключить ситуацию с гонками. Помните, что как только поток входит в любой синхронизированный метод на экземпляре, ни один другой поток не может войти ни в один синхронизированный метод на том же экземпляре. Однако не синхронизированные методы экземпляра по-прежнему остаются доступными для вызова.

## Оператор synchronized

Хотя создание synchronized методов в ваших классах – простой и эффективный способ достижения синхронизации, все же он работает не во всех случаях. Чтобы понять, почему, рассмотрим следующее. Предположим, что вы хотите синхронизировать доступ к объектам классов, которые не были предназначены для многопоточного доступа. То есть класс не использует методов synchronized. Более того, класс был написан не вами, а независимым разработчиком, и у вас нет доступа к его исходному коду. Значит, вы не можете добавить слово synchronized к объявлению соответствующих методов класса. Как может быть синхронизирован доступ к объектам такого класса? К счастью, существует довольно простое решение этой проблемы: вы просто заключаете вызовы методов этого класса в блок synchronized.

Вот общая форма оператора synchronized:

synchronized(объект) {

// операторы, подлежащие синхронизации

}

Здесь объект – это ссылка на синхронизируемый объект. Блок synchronized гарантирует, что вызов метода-члена объекта произойдет только тогда, когда текущий поток успешно войдет в монитор объекта.

Ниже показана альтернативная версия предыдущего примера с использованием синхронизированного блока внутри метода run().

// Эта программа использует синхронизированный блок.

class Callme {

void call(String msg) {

System.out.print("[" + msg);

try {

Thread.sleep(1000);

} catch (InterruptedException e) {

System.out.println("Interrupted");

}

System.out.println("]");

}}

class Caller implements Runnable {

String msg;

Callme target;

Thread t;

public Caller(Callme targ, String s) {

target = targ;

msg = s;

t = new Thread(this);

t.start();

}

// синхронизированные вызовы call()

public void run() {

synchronized(target) { // синхронизированный блок

target.call(msg);

}

}}

class Synch1 {

public static void main(String args[]) {

Callme target = new Callme();

Caller ob1 = new Caller(target, "Добро пожаловать");

Caller ob2 = new Caller(target, "в синхронизированный");

Caller ob3 = new Caller(target, "мир!");

// wait for threads to end

try {

ob1.t.join();

ob2.t.join();

ob3.t.join();

} catch(InterruptedException e) {

System.out.println("Прервано");

}

}}

Здесь метод call() не модифицирован словом synchronized. Вместо этого используется оператор synchronized внутри метода run() класса Caller. Это позволяет получить тот же корректный результат, что и предыдущий пример, поскольку каждый поток ожидает окончания выполнения своего предшественника.

## Межпотоковые коммуникации

Предыдущие примеры, безусловно, блокировали другие потоки от асинхронного доступа к некоторым методам. Это использование неявных мониторов объектов Java является мощным средством, но вы можете достичь более тонкого уровня контроля посредством межпроцессных коммуникаций. Как вы увидите, это особенно просто в Java.

Как обсуждалось ранее, многопоточность заменила программирование на основе циклов событий за счет разделения ваших задач на дискретные, логически обособленные единицы. Потоки также предоставляют вторичную выгоду: они исключают опрос. Опрос обычно реализуется в виде цикла, используемого для периодической проверки некоторого условия. Как только условие истинно, выполняется определенное действие. Это расходует время процессора. Например, рассмотрим классическую проблему, когда один поток генерирует некоторые данные, а другой принимает их. Чтобы сделать проблему более интересной, предположим, что поставщик данных должен ожидать, когда потребитель завершит, прежде чем поставщик сгенерирует новые данных. В системах с опросом потребитель данных тратит много циклов процессора на ожидание данных от поставщика. Как только поставщик завершает, он должен начать опрос, расходующий циклы процессора в ожидании завершения работы потребителя данных, и так далее. Понятно, что такая ситуация нежелательна.

Чтобы избежать опроса, Java включает элегантный механизм межпроцессных коммуникаций посредством методов wait(), notify() и notifyAll(). Эти методы реализованы как final в классе Object, поэтому они доступны всем классам. Все три метода могут быть вызваны только из synchronized-контекста. Хотя с точки зрения компьютерной науки они концептуально сложны, правила применения этих методов достаточно просты.

wait() принуждает вызывающий поток отдать монитор и приостановить выполнение до тех пор, пока какой-нибудь другой поток не войдет в тот же монитор и не вызовет notify().

notify() возобновляет работу потока, который вызвал wait() на том же самом объекте.

notifyAll() возобновляет работу всех протоков, который вызвали wait() на том же самом объекте. Одному из потоков дается доступ.

Эти методы объявлены в Object, как показано ниже:

final void wait() throws InterruptedException

final void notify()

final void notifyAll()

Существуют дополнительные формы wait(), позволяющие указать время ожидания.

Прежде чем рассматривать пример, демонстрирующий межпотоковое взаимодействие, необходимо сделать одно важное замечание. Хотя wait() обычно ожидает до тех пор пока не будет вызван notify() или notifyAll(), существует вероятность, что в очень редких случаях ожидающий поток может быть разбужен поддельным сигналом. При этом ожидающий поток возобновляется без вызова notify() или notifyAll(). (По сути, поток возобновляется без явных причин.) Из-за этой маловероятной возможности Sun рекомендует выполнять вызовы wait() внутри цикла, проверяющего условие, по которому поток ожидает. В приведенном ниже примере показан такой подход.

А пока рассмотрим пример, использующий wait() и notify(). Для начала проанализируем следующий простой пример программы, некорректно реализующий задачу “поставщик/потребитель”. Она состоит из четырех классов: Q – очередь, которую нужно синхронизировать, Producer – объект-поток, который генерирует элементы очереди, Consumer – объект-поток, принимающий элементы очереди, и PC – крошечный класс, который создает объекты Q, Producer и Consumer.

// Неправильная реализация поставщика и потребителя.

class Q {

int n;

synchronized int get() {

System.out.println("Получено: " + n);

return n;

}

synchronized void put(int n) {

this.n = n;

System.out.println("Отправлено: " + n);

}}

class Producer implements Runnable {

Q q;

Producer(Q q) {

this.q = q;

new Thread(this, "Поставщик").start();

}

public void run() {

int i = 0;

while(true) {

q.put(i++);

}

}}

class Consumer implements Runnable {

Q q;

Consumer(Q q) {

this.q = q;

new Thread(this, "Потребитель").start();

}

public void run() {

while(true) {

q.get();

}

}}

class PC {

public static void main(String args[]) {

Q q = new Q();

new Producer(q);

new Consumer(q);

System.out.println("Для останова нажмите Control-C.");

}}

Несмотря на то что методы put() и get() в Q синхронизированы, ничто не остановит переполнение потребителя поставщиком, как и ничто не помешает потребителю извлечь один и тот же компонент очереди дважды.

Правильный способ написания этой программы на Java заключается в том, чтобы применить wait() и notify(), чтобы передавать сигналы в обоих направлениях, как показано ниже:

// Правильная реализация поставщика и потребителя.

class Q {

int n;

boolean valueSet = false;

synchronized int get() {

while(!valueSet)

try {

wait();

} catch(InterruptedException e) {

System.out.println("InterruptedException перехвачено");

}

System.out.println("Получено: " + n);

valueSet = false;

notify();

return n;

}

synchronized void put(int n) {

while(valueSet)

try {

wait();

} catch(InterruptedException e) {

System.out.println("InterruptedException перехвачено");

}

this.n = n;

valueSet = true;

System.out.println("Отправлено: " + n);

notify();

}}

class Producer implements Runnable {

Q q;

Producer(Q q) {

this.q = q;

new Thread(this, "Поставщик").start();

}

public void run() {

int i = 0;

while(true) {

q.put(i++);

}

}}

class Consumer implements Runnable {

Q q;

Consumer(Q q) {

this.q = q;

new Thread(this, "Потребитель").start();

}

public void run() {

while(true) {

q.get();

}

}}

class PCFixed {

public static void main(String args[]) {

Q q = new Q();

new Producer(q);

new Consumer(q);

System.out.println("Для останова нажмите Control-C.");

}}

Внутри get() вызывается wait(). Это приостанавливает работу потока до тех пор, пока Producer не известит вас о том, что данные прочитаны. Когда это случается, выполнение внутри get() продолжается. После получения данных get() вызывает notify().

Это сообщает Producer, что все в порядке, и можно помещать в очередь следующий элемент данных. Внутри put() метод wait() приостанавливает выполнение до тех пор, пока Consumer не извлечет элемент из очереди. Когда выполнение возобновится, следующий элемент данных помещается в очередь и вызывается notify(). Это сообщает Consumer, что он теперь может извлечь его.

## Взаимная блокировка

Особый тип ошибок, которого следует избегать, имеющий отношение к многозадачности – это взаимная блокировка (deadlock), которая происходит, когда потоки имеют циклическую зависимость от пары синхронизированных объектов. Например, предположим, что один поток входит в монитор объекта X, а другой – в монитор объекта Y. Если поток в X попытается вызвать любой синхронизированный метод Y, он заблокируется, как и ожидалось. Однако если поток Y, в свою очередь, попытается вызвать любой синхронизированный метод X, то поток будет ожидать вечно, потому что для получения доступа к X он должен снять свой собственный блок на Y, чтобы первый поток мог отработать.

Взаимная блокировка является ошибкой, которую трудно отладить, по двум описанным ниже причинам.

В общем, она случается довольно редко, когда выполнение двух потоков точно совпадает по времени.

Она может происходить, когда в этом участвует более двух потоков и двух синхронизированных объектов. (То есть взаимная блокировка может случиться в результате более сложной последовательности событий, чем в приведенном примере.)

Чтобы полностью разобраться с этим явлением, лучше рассмотреть его в действии.

Следующий пример создает два класса – A и B, с методами foo() и bar() соответственно, которые приостанавливаются непосредственно перед попыткой вызова метода другого класса. Главный класс, названный Deadlock, создает экземпляры A и B, затем запускает второй поток, устанавливающий состояние взаимной блокировки. Методы foo() и bar() используют sleep(), чтобы стимулировать появление взаимной блокировки.

// Пример взаимной блокировки.

class A {

synchronized void foo(B b) {

String name = Thread.currentThread().getName();

System.out.println(name + " вошел в A.foo");

try {

Thread.sleep(1000);

} catch(Exception e) {

System.out.println("A прерван");

}

System.out.println(name + " пытается вызвать B.last()");

b.last();

}

synchronized void last() {

System.out.println("внутри A.last");

}}

class B {

synchronized void bar(A a) {

String name = Thread.currentThread().getName();

System.out.println(name + " вошел в B.bar");

try {

Thread.sleep(1000);

} catch(Exception e) {

System.out.println("B прерван");

}

System.out.println(name + " пытается вызвать A.last()");

a.last();

}

synchronized void last() {

System.out.println("внутри A.last");

}}

class Deadlock implements Runnable {

A a = new A();

B b = new B();

Deadlock() {

Thread.currentThread().setName("MainThread");

Thread t = new Thread(this, "RacingThread");

t.start();

a.foo(b); // получить блокировку внутри этого потока.

System.out.println("Назад в главный поток");

}

public void run() {

b.bar(a); // получить блокировку b в другом потоке.

System.out.println("Назад в другой поток");

}

public static void main(String args[]) {

new Deadlock();

}}

Поскольку эта программа заблокирована, вам придется нажать <CTRL+C> для завершения программы. Вы можете видеть весь поток и дамп кэша монитора, нажав <CTRL+BREAK>. Вы увидите, что RacingThread владеет монитором на b, в то время как последний ожидает монитора на a. В то же время MainThread владеет a и ожидает b. Эта программа никогда не завершится. Как иллюстрирует этот пример, если ваша многопоточная программа неожиданно зависла, то первое, что вы должны проверить – возможность взаимной блокировки.

## Приостановка, возобновление и останов потоков

Иногда возникает необходимость в приостановке выполнения потоков. Например, отдельный поток может использоваться для отображения времени дня. Если пользователю не нужны часы, то этот поток можно приостановить. В любом случае приостановка потока – простая вещь. Выполнение приостановленного потока может быть легко возобновлено.

Хотя применение методов класса Thread по именам suspend(), resume() и stop() выглядит как исключительно разумный и удобный подход к управлению выполнением потоков, они не должны использоваться в новых Java-программах. И вот почему. Метод suspend() класса Thread несколько лет назад был объявлен нежелательным в Java 2.

Это было сделано потому, что иногда он способен порождать серьезные системные сбои.

Предположим, что поток пытается получить блокировки на критичных структурах данных. Если поток приостановит в этот момент, блокировки не будут установлены. Другие потоки, которые могут ожидать этих ресурсов, могут оказаться взаимно заблокированными.

Метод resume() также нежелателен. Он не вызовет проблем, но не может быть использован без метода suspend() как своего дополнения.

Метод stop() класса Thread также объявлен устаревшим в Java 2. Это было сделано потому, что он также иногда может послужить причиной серьезных системных сбоев. Предположим, что поток выполняет запись в критически важную структуру данных, и успел выполнить только частичное обновление. Если его остановить в этот момент, структура данных может оказаться в поврежденном состоянии.

Поскольку вы не можете теперь использовать методы suspend(), resume() или stop() для управления потоками, то можете подумать, что теперь вообще нет способа приостановить, возобновить или прервать поток. К счастью, это не так. Вместо этого поток должен быть проектирован так, чтобы метод run() периодически проверял, должно ли выполнение потока быть приостановлено, возобновлено или прервано. Обычно это достигается использованием переменной-флага, указывающей состояние потока. До тех пор, пока этот флаг имеет значение “запущен”, метод run() должен продолжать выполнение. Если переменная имеет значение “прерван”, поток должен приостановиться. Если флаг получает значение “стоп”, то поток должен завершиться. Конечно, существует множество способов написать такой код, но основной принцип остается неизменным для всех программ.

В следующем примере показано как методы wait() и notify(), унаследованные от Object, могут применяться для управления выполнением потока.

Рассмотрим работу этой программы.

Класс NewThread содержит переменную экземпляра типа boolean по имени suspendFlag, используемую для управления выполнением потока. Конструктор инициализирует ее значением false. Метод run() содержит блок synchronized, который проверяет состояние suspendFlag. Если ее значение равно true, вызывается метод wait() для приостановки выполнения потока. Метод mysuspend() устанавливает значение suspendFlag в true. Метод myresime() устанавливает suspendFlag в false и вызывает notify() для того, чтобы “разбудить” поток. И, наконец, метод main() модифицирован для вызова методов mysuspend() и myresime().

// Приостановка и возобновление потока современным способом.

class NewThread implements Runnable {

String name; // имя потока

Thread t;

boolean suspendFlag;

NewThread(String threadname) {

name = threadname;

t = new Thread(this, name);

System.out.println("Новый поток: " + t);

suspendFlag = false;

t.start(); // запустить поток

}

// Точка входа потока.

public void run() {

try {

for(int i = 15; i > 0; i--) {

System.out.println(name + ": " + i);

Thread.sleep(200);

synchronized(this) {

while(suspendFlag) {

wait();

}

}

}

} catch (InterruptedException e) {

System.out.println(name + " прерван.");

}

System.out.println(name + " завершен.");

}

void mysuspend() {

suspendFlag = true;

}

synchronized void myresume() {

suspendFlag = false;

notify();

}}

class SuspendResume {

public static void main(String args[]) {

NewThread ob1 = new NewThread("Один");

NewThread ob2 = new NewThread("Два");

try {

Thread.sleep(1000);

ob1.mysuspend();

System.out.println("Приостановка потока Один");

Thread.sleep(1000);

ob1.myresume();

System.out.println("Возобновление потока Один");

ob2.mysuspend();

System.out.println("Приостановка потока Два");

Thread.sleep(1000);

ob2.myresume();

System.out.println("Возобновление потока Два");

} catch (InterruptedException e) {

System.out.println("Главный поток прерван");

}

// ожидание завершения потоков

try {

System.out.println("Ожидание завершения потоков.");

ob1.t.join();

ob2.t.join();

} catch (InterruptedException e) {

System.out.println("Главный поток прерван");

}

System.out.println("Главный поток завершен");

}}

Хотя этот метод не так “чист”, как старый, его следует придерживаться, дабы избежать ошибок времени выполнения. Это подход, который должен применяться во всем новом коде.

## Использование многопоточности

Ключ к эффективному использованию многопоточный средств Java лежит в том, чтобы думать параллельно вместо того, чтоб думать последовательно. Например, когда вы имеете две подсистемы в программе, которые могут выполняться одновременно, оформите их в виде отдельных потоков. При взвешенном применении многопоточности вы будете писать очень эффективные программы. Однако следует проявлять осторожность.

Если вы создадите слишком много потоков, вы можете даже снизить производительность всей программы вместо того, чтобы повысить ее. Помните, что переключение контекстов между потоками требует определенных накладных расходов. Если вы создадите очень много потоков, больше времени процессора будет затрачено на переключение контекста, нежели на само выполнение программы.

# Класс Applet

В настоящей главе рассматривается класс для создания аплетов. Класс Applet содержится в пакете Applet, который обеспечивает основу java.applet. Applet включает несколько методов, которые предоставляют возможность тонкого контроля выполнения аплета. В дополнение java.applet также определяет три интерфейса: AppletContext, AudioClip и AppletStub.

## Два типа аплетов

Важно подчеркнуть, что существует два варианта аплетов. Первый основан непосредственно на классе Applet, описанном в настоящей главе. Эти аплеты используют Abstract Window Toolkit (AWT) для предоставления пользовательского графического интерфейса (или вообще его не используют). Этот вид аплетов доступен с самого начала существования Java.

Второй тип аплетов основан на Swing-классе JApplet. Аплеты Swing используют классы Swing для построения графического интерфейса пользователя. Swing предлагает более богатый и часто более легкий в применении пользовательский интерфейс, чем AWT. Поэтому аплеты на основе Swing в настоящее время наиболее популярны. Однако традиционные аплеты на основе AWT все еще применяются, особенно тогда, когда требуется построить очень простой пользовательский интерфейс. А потому и аплеты на основе AWT, и аплеты на основе Swing совершенно законны.

Поскольку JApplet наследуется от Applet, все средства Applet также доступны в JApplet, и большая часть информации настоящей главы касается обоих типов аплетов. А потому, даже если вас интересуют только аплеты Swing, информация, приведенная в этой главе, все равно будет полезна и необходима. Однако следует понимать, что при создании аплетов на основе Swing существуют некоторые дополнительные ограничения.

## Основы аплетов

Все аплеты являются подклассами (прямыми или непрямыми) класса Applet. Аплеты не являются самостоятельными программами. Вместо этого они выполняются либо внутри Web-браузера, либо в средстве просмотра аплетов.

Выполнение аплета не начинается с main(). На самом деле лишь немногие аплеты имеют методы main(). Вместо этого выполнение аплета запускается и управляется посредством совершенно другого механизма, который будет описан ниже. Вывод окна вашего аплета не осуществляется посредством System.out.println(). Вместо этого в не-Swing аплетах вывод обрабатывается различными методами AWT, такими как drawString(), который направляет строку в точку с указанными координатами X,Y. Ввод также обрабатывается иначе, чем в консольных приложениях. (Помните, что аплеты на основе Swing используют классы Swing для обработки взаимодействия с пользователем, и они также будут описаны далее в настоящей книге.)

Для применения аплета его необходимо специфицировать в файле HTML. Одним из способов сделать это является дескриптор APPLET. (Дескриптор OBJECT также может быть использован, но Sun в настоящее время рекомендует дескриптор APPLET, и именно этот дескриптор мы используем в примерах настоящей книги.) Аплет будет запущен Web-браузером с поддержкой Java, когда он встретит дескриптор APPLET внутри файла HTML. Чтобы просматривать и тестировать аплет более удобно, просто включите комментарий в заголовок файла вашего исходного кода Java, который содержит дескриптор APPLET. Таким образом, код будет документирован HTML-конструкциями, необходимыми вашему аплету, и вы сможете протестировать скомпилированный аплет, запустив средство просмотра аплетов с файлом исходного кода Java, специфицированным в качестве целевого.

## Класс Applet

Класс Applet определяет методы, показанные в табл. 21.1. Applet обеспечивает всю необходимую поддержку для выполнения аплета, такую как его запуск и останов. Кроме того, он предоставляет методы для загрузки и отображения графических образов, а также методы для загрузки и воспроизведения аудиоклипов. Applet расширяет AWT-класс Panel. В свою очередь, Panel расширяет Container, который расширяет Component. Все эти классы обеспечивают поддержку графического оконного интерфейса на базе Java.Таким образом, Applet обеспечивает всю необходимую поддержку деятельности на основе окон. (AWT будет подробно рассматриваться в последующих главах.)

## Архитектура аплетов

Аплет — это оконная программа, и потому его архитектура отличается от консольных программ, показанных в первой части нашей книги. Если вы знакомы с программированием под Windows, то при написании аплетов почувствуете себя буквально как дома. Если же нет, то вы должны понимать некоторые концепции, которые будут описаны ниже.

Во-первых, аплеты управляются событиями. Хотя мы не будем рассматривать обработку событий до следующей главы, важно понимать в общем, как управляемая событиями архитектура влияет на дизайн аплетов. Аплет включает в себя набор служебных процедур, обрабатывающих прерывания. Вот как это работает. Аплет ожидает, пока не произойдет событие. Исполняющая система извещает аплет о событии, вызывая обработчик события, предоставленный аплетом. Как только это случилось, аплет должен предпринять соответствующие действия и немедленно возвратить управление. Это — важнейший момент. Большей частью ваш аплет не должен входить в “режим” операций, в которых он будет удерживать управление в течение длительного периода. Вместо этого он должен выполнить определенные действия в ответ на события и затем вернуть управление исполняющей системе. В тех ситуациях, когда вашему аплету нужно выполнять какое-то повторяющееся действие (например, отображать в окне прокручивающееся сообщение), вы должны запустить дополнительный поток выполнения.

Во-вторых, пользователь инициирует взаимодействие с аплетом — и никакого пути в обход! Как вы знаете, в неоконных программах, когда программе необходим ввод, она выводит приглашение пользователю и затем вызывает некоторый метод ввода, например, readLine(). В аплетах все работает не так. Вместо этого пользователь взаимодействует с аплетом так, как он желает и когда хочет. Эти взаимодействия отсылаются аплету в виде событий, на которые тот должен отреагировать. Например, когда пользователь щелкает кнопкой мыши внутри окна аплета, генерируется событие щелчка. Если пользователь нажимает клавишу, когда окно аплета имеет фокус ввода, генерируется событие нажатия клавиши. Как вы увидите в последующих главах, аплеты могут содержать различные элементы управления, такие как экранные кнопки и флажки. Всякий раз, когда пользователь взаимодействует с одним из этих элементов управления, генерируются события.

Хотя архитектуру аплетов понять не так просто, как архитектуру консольных программ, Java облегчает это, насколько возможно. Если вы писали программы под Windows, то знаете, насколько устрашающе может выглядеть эта среда. К счастью, Java предоставляет в ваше распоряжение намного более ясный подход, которым можно овладеть гораздо быстрее.

Скелет аплета

Все аплеты, кроме только наиболее тривиальных, переопределяют методы, обеспечивающие базовые механизмы взаимодействия браузера или средства просмотра аплетов с самим аплетом и управляющие его выполнением. Четыре из этих методов — init(), start(), stop() и destroy() — применимы ко всем аплетам; они определены в классе Applet. Предусмотрены реализации по умолчанию всех этих методов. Однако в их переопределении не нуждаются лишь очень простые аплеты.

Аплеты на базе AWT (вроде тех, что обсуждаются в настоящей главе) также будут переопределять метод paint(), который определен в AWT-классе Component. Этот метод вызывается, когда вывод аплета должен быть заново отображен. (Аплеты на основе Swing используют другой механизм для решения этой задачи.) Эти пять методов могут быть собраны в “скелет”, показанный ниже.

// Скелет аплета.

import java.awt.\*;

import java.applet.\*;

/\*

<applet code="AppletSkel" width=300 height=100>

</applet>

\*/

public class AppletSkel extends Applet {

// Вызывается первым.

public void init() {

// инициализация

}

/\* Вызывается вторым, после init(). Также вызывается

при перезапуске аплета. \*/

public void start() {

// запускает или возобновляет выполнение

}

// Вызывается при останове аплета.

public void stop() {

// приостановка выполнения

}

/\* Вызывается перед уничтожением аплета. Это - последний

выполняемый метод. \*/

public void destroy() {

// выполняет завершающие действия

}

// Вызывается, когда окно аплета должно быть восстановлено.

public void paint(Graphics g) {

// перерисовка содержимого окна

}

}

Хотя этот скелет ничего и не делает, его можно скомпилировать и запустить.

# Введение в Swing

Консольные программы весьма удобны для обучения основам Java и эффективно используются в целом ряде специализированых приложений, например в серверном коде, но в большинстве реальных приложений имеется графический пользовательский интерфейс. Во время написания данной книги наиболее популярным средством для создания подобных Jаvа-приложений была библиотека Swing.

Библиотека Swing предоставляет коллекцию классов и интерфейсов, поддерживающих богатый набор визуальных компонентов, таких как кнопки, поля для ввода текста, полосы прокрутки, флажки, деревья узлов и таблицы. Наличие столь широкой палитры элементов управления позволяет создавать чрезвычайно эффективные и вместе с тем простые в использовании графические интерфейсы. Учитывая необычайную популярность библиотеки Swing, ее можно с уверенностью отнести к категории средств, с которыми должен быть знаком любой специалист, пишущий программы на Java.

Необходимо с самого начала подчеркнуть, что тема Swing очень обширна, и для ее полного обсуждения понадобилась бы отдельная книга. Поэтому в данной главе мы коснемся лишь самых важных вопросов. Однако и этого будет достаточно для того, чтобы вы получили общее представление о том, что такое библиотека Swing, ознакомились с историей ее создания, основными концепциями и философией проектирования. В этой главе рассматриваются пять наиболее часто используемых компонентов (элементов GUI), создаваемых средствами Swing: ярлыки, кнопки, текстовые поля, флажки и списки. Завершает главу демонстрационный пример, в котором показано, как создавать аплеты на основе Swing. Несмотря на то что ниже описана лишь небольшая часть инструментальных средств Swing, изучив их, вы сможете самостоятельно создавать несложные программы с GUI-поддержкой. Кроме того, это подготовит вас к последующему более детальному изучению всех возможностей Swing.

## Происхождение и философия Swing

В ранних версиях Java средства Swing отсутствовали. Их появление было обусловлено стремлением устранить недостатки, свойственные оригинальной подсистеме GUI Java, реализованной в виде библиотеки AWT (Abstract Window Toolkit). Библиотека AWT содержит базовый набор компонентов, поддерживающих создание вполне работоспособных, но ограниченных по своим возможностям графических пользовательских интерфейсов. Ограниченность библиотеки AWT объясняется, в частности, тем, что различные ее визуальные компоненты транслируются в соответствующие платформенно-зависимые эквиваленты, так называемые равноправные компоненты (peers). Отсюда следует, что внешний вид компонентов AWT определяется не средствами Java, а платформой. Поскольку в компонентах AWT используются ресурсы в виде машинно-зависимого кода, их называют тяжеловесными (heavyweight).

Использование машинно-зависимых равноправных компонентов порождает ряд проблем. Во-первых, из-за отличий в операционных системах компоненты могут выглядеть и даже вести себя по-разному на различных платформах. Это нарушает основополагающий принцип Java: "написано однажды, работает везде". Во-вторых, внешний вид каждого компонента остается фиксированным, и изменить его очень трудно (причина та же – зависимость от конкретной платформы). И в-третьих, применение тяжеловесных компонентов влечет за собой ряд новых ограничений. В частности, тяжеловесный компонент всегда имеет прямоугольную форму и является непрозрачным.

Вскоре после выпуска первоначальной версии Java стало очевидным, что ограничения AWT настолько серьезны, что дпя их преодоления требуется совершенно иной подход. В итоге в 1997 году появилась библиотека компонентов Swing, включенная в состав набора библиотек классов JFC (Java Foundation Classes). Первоначально библиотека Swing использовалась в версии Java 1.1 как отдельная библиотека. Но в версии Java 1.2 средства Swing (как, впрочем, и остальные элементы JFC) были полностью интегрированы в Java.

Swing устраняет ограничения, присущие компонентам АWТ, благодаря использованию двух основных средств: легковесных компонентов и подключаемых стилей оформления. Несмотря на то что программисту почти не приходится использовать эти средства напрямую, именно они составляют фундамент философии проектирования, заложенной в Swing, и в значительной мере обусловливают возможности и удобство использования этой библиотеки. Рассмотрим каждое из них в отдельности.

За небольшим исключением все компоненты Swing являются легковесными. Это означает, что они написаны полностью на Java и не зависят от конкретной платформы, поскольку не опираются на платформенно-зависимые равноправные компоненты. Легковесные компоненты обладают рядом существенных преимуществ, к числу которых относятся эффективность и гибкость. Например, легковесный компонент может быть прозрачным, а его форма может отличаться от прямоугольной. Легковесные компоненты не транслируются в платформенно-зависимые равноправные компоненты, и поэтому их внешний вид определяет библиотека Swing, а не базовая операционная система.

Следовательно, элементы пользовательского интерфейса, созданные средствами Swing, выглядят одинаково на разных платформах.

Благодаря тому что каждый компонент Swing визуализируется кодом Java, а не платформенно-зависимыми равноправными компонентами, становится возможным раздельное управление внешним видом компонента и логикой его функционирования, и именно эту задачу решает Swing. Такое разделение предоставляет значительное преимущество: оно позволяет изменить внешний вид компонента, не затрагивая другие его свойств. Иными словами, появляется возможность "подключать" новый стиль оформления к компоненту, не создавая никаких побочных эффектов в коде, использующем данный компонент.

Java предоставляет различные стили оформления, такие как "металлик" и Nimbus, доступные каждому пользователю Swing. Металлический стиль также называют стилем оформления Java. Это платформенно-независимый стиль оформления, доступный во всех средах выполнения программ на Java. Он же применяется по умолчанию, поэтому именно он и будет использоваться в примерах, представленных в этой главе.

Реализация подключаемых стилей оформления в Swing стала возможной благодаря тому, что при создании Swing был использован видоизмененный вариант классической архитектуры модель-представление-контроллер (MVC). В терминологии MVC модель соответствует информации о состоянии, ассоциированном с компонентом. Например, в случае флажка модель содержит поле, указывающее на состояние флажка. Представление определяет, как выглядит компонент на экране, включая любые аспекты представления, на которые может влиять текущее состояние модели. Контроллер определяет реакцию компонента на действия пользователя. Так, если пользователь щелкнет мышью на флажке, контроллер отреагирует, изменив модель таким образом, чтобы отразить выбор пользователя (установку или сброс флажка). В ответ на действия пользователя обновляется и представление. Разделение компонента на модель, представление и контроллер, позволяет добиться того, что особенности реализация одной из этих составляющих не будут влиять на две другие. Например, в некоторых реализациях представления один и тот же компонент может отображаться разными способами, а модель и контроллер – оставаться без изменения.

Несмотря на всю концептуальную привлекательность архитектуры МУС и лежащих в ее основе принципов, для компонентов Swing разделение функций между представлением и контроллером не обеспечило заметных преимуществ. В связи с этим в Swing используется видоизмененный вариант MVC, в котором представление и контроллер объединены в единую логическую сущность, называемую делегатом пользовательского интерфейса. Поэтому принятый в Swing подход называется архитектурой модельделегат, или архитектурой с отделяемой моделью. Таким образом, компоненты Swing нельзя рассматривать как классическую реализацию архитектуры MVC, хотя их архитектура и опирается на нее. В процессе разработки вам не придется иметь дело непосредственно с моделями или делегатами пользовательского интерфейса, но они будут незримо присутствовать в создаваемых вами программах.

## Компоненты и контейнеры

Графический пользовательский интерфейс Swing состоит из двух ключевых элементов: компонентов и контейнеров. Такое разделение во многом условно, поскольку все контейнеры одновременно являются компонентами. Различие между ними кроется в их предполагаемом назначении. В общепринятом понимании этого термина компонент – это независимый визуальный элемент управления (например, кнопка или поле ввода текста), тогда как контейнер может содержать в себе несколько компонентов. Следовательно, контейнер – это особая разновидность компонента. Для того чтобы отобразить компонент на экране, его следует поместить в контейнер. Поэтому в графическом интерфейсе должен присутствовать хотя бы один контейнер. А так как контейнеры одновременно являются компонентами, то один контейнер может содержать в себе другой.

Это дает возможность сформировать так называемую иерархию включения, на вершине которой находится контейнер верхнего уровня.

## Компоненты

Подавляющее большинство компонентов Swing являются производными от класса JComponent. (Единственное исключение из этого правила – четыре контейнера верхнего уровня, которые будут описаны в следующем разделе.) В классе JComponent реализуются функциональные возможности, общие для всех компонентов. Например, в нем поддерживаются подключаемые стили оформления. Этот класс наследует свойства классов Container и Component из библиотеки AWT. Таким образом, компоненты Swing создаются на основе компонентов AWT и совместимы с ними.

Все компоненты Swing представляются классами, находящимися в пакете javax.swing.

## Контейнеры

В Swing определены два типа контейнеров. К первому типу относятся следующие контейнеры верхнего уровня: JFrame, JApplet, JWindow и JDialog. Эти контейнеры не наследуют класс JComponent. В то же время они наследуют классы Component и Container библиотеки AWT. В отличие от других, легковесных компонентов Swing, контейнеры верхнего уровня являются тяжеловесными. Именно поэтому они образуют отдельную группу в библиотеке Swing.

Как следует из названия контейнеров верхнего уровня, они должны находиться на вершине иерархии контейнеров и не могут содержаться в других контейнерах. Более того, любая иерархия должна начинаться именно с контейнера верхнего уровня. В прикладных программах чаще всего используется контейнер типа JFrame, а в аплетах – контейнер типа JApplet.

Контейнеры второго типа являются легковесными и происходят от класса JComponent. В качестве примера легковесных контейнеров можно привести классы JPanel, JScrollPane и JRootPane. Легковесные контейнеры могут содержаться в других контейнерах, и поэтому они нередко используются для объединения группы взаимосвязанных компонентов.

## Панели контейнеров верхнего уровня

В каждом контейнере верхнего уровня определен набор панелей. На вершине иерархии находится корневая панель – экземпляр класса JRootPane, который представляет собой легковесный контейнер, предназначенный для управления другими панелями. Он также позволяет управлять строкой меню. Корневая панель включает в себя "стеклянную" панель, панель содержимого и многослойную панель.

"Стеклянная" (иначе прозрачная) панель является панелью верхнего уровня. Она располагается поверх всех остальных панелей и полностью покрывает их. Прозрачная панель позволяет управлять событиями мыши, относящимися ко всему контейнеру (а не к отдельным элементам управления), и выполнять операции рисования поверх любого другого компонента. В большинстве случаев у вас не будет возникать необходимость в непосредственном использовании прозрачной панели. Многослойная панель позволяет задавать глубину расположения компонента, определяя порядок перекрытия одних компонентов другими. (Таким образом, многослойная панель позволяет упорядочивать компоненты по координате Z, хотя это требуется не так уж часто.) В состав многослойной панели входит панель содержимого и (необязательно) строка меню. Несмотря на то что прозрачная и многослойная панели являются неотъемлемыми частями контейнера верхнего уровня и выполняют важные функции, их действия по большей части скрыты не только от пользователей, но и от разработчиков прикладных программ.

Ваше приложение в основном будет взаимодействовать с панелью содержимого, в которую добавляются визуальные компоненты. Иными словами, добавляя компонент, например кнопку, в контейнер верхнего уровня, вы на самом деле добавляете его в панель содержимого.

## Менеджеры компоновки

Прежде чем приступить к написанию программ средствами Swing, вам необходимо получить хотя бы общее представление о менеджерах компоновки. Менеджер компоновки управляет размещением компонентов в контейнере. В Java определено несколько таких менеджеров. Большинство из них входит в состав AWT (т.е. в пакет j ava. awt), но Swing предоставляет также ряд дополнительных менеджеров компоновки. Все менеджеры компоновки являются экземплярами классов, реализующих интерфейс LayoutManager. Некоторые из менеджеров компоновки реализуют интерфейс Layoutмanager2. Менеджеры компоновки, как и многие другие компоненты Swing, невозможно обсудить во всех подробностях в одной главе. Поэтому ограничимся рассмотрением только двух из них: BorderLayout и FlowLayout.

Для панели содержимого менеджером компоновки по умолчанию является BorderLayout. Этот менеджер определяет в составе контейнера пять областей, в которые могут помещаться компоненты. Первая область располагается посредине и называется центральной. Остальные четыре располагаются в соответствии со сторонами света и носят названия северная, южная, восточная и западная. По умолчанию компонент, добавляемый в панель содержимого, располагается в центральной области. Для того чтобы расположить компонент в другой области, следует указать ее имя.

Несмотря на то что возможностей, предоставляемых менеджером компоновки BorderLayout, зачастую оказывается достаточно, иногда возникает потребность в других менеджерах компоновки. К числу самых простых относится менеджер компоновки FlowLayout, который размещает компоненты построчно: слева направо и сверху вниз.

Заполнив текущую строку, он переходит к следующей. Такая компоновка предоставляет лишь ограниченный контроль над расположением компонентов, хотя и проста в применении. Однако при изменении размеров контейнера расположение компонентов может измениться.

## Простое Swing-приложение

Swing-программы отличаются и от консольных программ, и от AWT-программ, показанных в этой книге. Например, они используют другие наборы компонентов и другие иерархии контейнеров, чем AWT. Swing-программы имеют также особые требования, которые относятся к потоковой обработке. Самый лучший способ понять структуру Swingпрограммы – испытать в деле рабочий пример. Существуют два типа Java-программ, в которых обычно используется Swing. Первый тип – это настольные приложения, а второй тип – аплеты. В этом разделе будет рассмотрен пример создания Swing-приложения.

Созданием Swing-аплета мы тоже будем заниматься в этой главе, но чуть позже.

Несмотря на то что следующая программа довольно небольшая, она демонстрирует один из способов написания Swing-приложения. Плюс ко всему, она иллюстрирует несколько ключевых возможностей Swing. В ней используется два Swing-компонента: JFrame и JLabel. JFrame – контейнер верхнего уровня, который обычно используется в Swing-приложениях. JLabel – Swing-компонент, создающий метку, которая является компонентом, отображающим информацию. Метка – это самый простой Swing-компонент, поскольку она является пассивным компонентом. То есть, метка не реагирует на действия пользователя. Она служит для отображения выходных данных. Программа использует контейнер JFrame для хранения экземпляра JLabel. Метка отображает короткое текстовое сообщение.

// Простое Swing-приложение.

import javax.swing.\*;

class SwingDemo {

SwingDemo() {

// Создание нового контейнера JFrame.

JFrame jfrm = new JFrame("A Simple Swing Application");

// JFrame jfrm = new JFrame("Простое Swing-приложение");

// Задаем фрейму исходный размер.

jfrm.setSize(275, 100);

// Прекращаем работу программы, если пользователь закрывает приложение.

jfrm.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

// Создаем метку с текстом.

JLabel jlab = new JLabel("Swing means powerful GUIs.");

// JLabel jlab = new JLabel("Swing означает мощный GUI.");

// Добавляем метку на панель содержимого.

jfrm.add(jlab);

// Отображаем фрейм.

jfrm.setVisible(true);

}

public static void main(String args[]) {

// Создаем фрейм в потоке диспетчеризации событий.

SwingUtilities.invokeLater(new Runnable() {

public void run() {

new SwingDemo();

}

});

}

}

Поскольку программа SwingDemo иллюстрирует несколько основополагающих концепций Swing, мы постараемся разобраться с ней тщательно, строка за строкой. Программа начинается с импорта javax.swing. Как уже упоминалось, этот пакет содержит компоненты, определяемые Swing. Например, javax.swing определяет классы, реализующие метки, кнопки, текстовые элементы управления и меню. Он будет включен во все программы, использующие Swing.

Затем объявляется класс SwingDemo и конструктор для этого класса. Конструктор – это объект, в котором выполняется большинство действий программы. Он начинается с создания JFrame с помощью следующей строки кода:

JFrame jfrm = new JFrame("A Simple Swing Application");

В результате будет создан контейнер jfrm, определяющий прямоугольное окно со строкой заголовка, кнопками закрытия, сворачивания, разворачивания и восстановления, а также с системным меню. Таким образом, программа создает стандартное окно верхнего уровня. Конструктору передается заголовок окна.

Затем задаются размеры окна с помощью следующего оператора:

jfrm.setSize(275, 100);

Метод setSize() (он наследуется классом JFrame из класса AWT Component) задает размеры окна, определяемые в пикселях. Он имеет такую форму:

void setSize(int width, int height)

В этом примере ширина окна (width) равняется 275 пикселям, а высота (height) – 100.

По умолчанию, когда закрывается окно верхнего уровня (например, когда пользователь щелкает на кнопке закрытия), окно удаляется из экрана, но работа приложения не прекращается. Несмотря на то что это поведение в некоторых ситуациях является полезным, для большинства приложений оно не подходит. Чаще всего при закрытии окна верхнего уровня нужно будет просто прекращать работу всего приложения. Это можно сделать двумя способами. Самый простой из них – вызов метода setDefaultCloseOperation(), что и делается в программе:

jfrm.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

Теперь работа всего приложения будет прекращаться при закрытии окна. Общая форма метода setDefaultCloseOperation() выглядит так:

void setDefaultCloseOperation(int what)

Значение what определяет, что происходит при закрытии окна. Помимо JFrame. EXIT\_On\_CLOSE доступно несколько других опций:

* JFrame.DISPOSE\_ON\_CLOSE
* JFrame.HIDE\_ON\_CLOSE
* JFrame.DO\_NOTHING\_ON\_CLOSE

В их именах отражены выполняемые действия. Эти константы объявлены в классе WindowConstants, который является интерфейсом, объявленным в пакете javax.swing, реализуемым контейнером JFrame:

JLabel jlab = new JLabel("Swing means powerful GUIs.");

JLabel – самый простой и легкий в использовании компонент, так как он не принимает от пользователя входных данных. Он просто отображает информацию, которая может представлять текст, значок или являться их комбинацией. Метка, созданная программой, содержит только текст, который передается ее конструктору.

Следующая строка кода добавляет метку в панель содержимого фрейма:

jfrm.add(jlab);

Как было сказано ранее, все контейнеры верхнего уровня имеют панель содержимого, в которой размещаются компоненты. Таким образом, чтобы добавить компонент во фрейм, нужно добавить его в панель содержимого фрейма. Это можно сделать, вызвав метод add() для ссылки на JFrame (в данном случае jfrm). Общая форма метода add() показана ниже:

Component add(Component comp)

Метод add() является наследником JFrame из AWT-класса Container.

По умолчанию панель содержимого, связанная с JFrame, использует граничную компоновку. Только что показанный вариант метода add() добавляет метку и помещает ее в центре. Другие варианты метода add() позволяют задать одну из граничных областей.

Когда компонент добавляется в центр, его размер подгоняется автоматически таким образом, чтобы компонент смог уместиться в центре.

Прежде чем продолжить, нужно сделать одно важное замечание. До выхода JDK 5, когда компонент добавляли в панель содержимого, нельзя было вызывать метод add() непосредственно на экземпляре JFrame. Вместо этого нужно было вызывать метод add() для панели содержимого объекта JFrame. Панель содержимого можно было получить путем вызова метода getContentPane() на экземпляре JFrame. Метод getContentPane() показан ниже:

Container getContentPane( )

Container получает ссылку на окно содержимого. После этого производился вызов метода add() по этой ссылке для добавления компонента в панель содержимого. Таким образом, чтобы добавить jlab в jfrm, раньше нужно было использовать следующий оператор:

jfrm.getContentPane().add(jlab); // старый стиль

Здесь метод getContentPane() сначала получает ссылку на панель содержимого, после чего метод add() добавляет компонент в контейнер, присоединенный к этому окну.

Эту же процедуру нужно было выполнять для вызова метода remove(), когда требовалось удалить компонент, и метода setLayout(), чтобы задать диспетчер компоновки для окна содержимого. В коде, написанном на Java до версии 5.0, вам будут часто попадаться вызовы метода getContentPane(). Однако теперь использовать этот метод больше не нужно. Можно просто вызывать методы add(), remove() и setLayout() непосредственно на JFrame, так как они были изменены специально для того, чтобы автоматически работать с окном содержимого.

Последний оператор в конструкторе SwingDemo нужен для того, чтобы сделать окно видимым:

jfrm.setVisible(true);

Метод setVisible() является наследником AWT-класса Component. Если его аргумент будет равен true, то окно будет отображаться. В противном случае оно будет скрыто. По умолчанию JFrame является невидимым, поэтому чтобы показать его, нужно вызвать метод setVisible(true).

Внутри метода main() создается объект SwingDemo, который отображает окно и метку. Обратите внимание, что конструктор SwingDemo вызывается с помощью следующих трех строк кода:

SwingUtilities.invokeLater(new Runnable() {

public void run() {

new SwingDemo();

}

});

Выполнение этой последовательности приводит к созданию объекта SwingDemo в потоке диспетчеризации событий, а не в главном потоке приложения. И вот почему.

В общем случае Swing-программы управляются событиями. Например, когда пользователь взаимодействует с компонентом, генерируется событие. Событие передается в приложение путем вызова обработчика событий, определенного в приложении. Однако обработчик выполняется в потоке диспетчеризации событий, поддерживаемом Swing, а не в главном потоке приложения. Таким образом, хотя обработчики событий и определены в программе, они вызываются в потоке, который не был создан вашей программой.

Чтобы избежать возникновения этой проблемы (включая вероятность возникновения взаимной блокировки), все GUI-компоненты Swing нужно создавать и обновлять из потока диспетчеризации событий, а не из главного потока приложения. А метод add() выполняется в главном потоке. Таким образом, метод main() не может напрямую наследовать объект SwingDemo. Наоборот, он должен создать объект Runnable, который выполняется в потоке диспетчеризации событий, и заставить этот объект создать GUI.

Чтобы код GUI можно было создать в потоке диспетчеризации событий, необходимо использовать один из двух методов, определенных в классе SwingUtilities. Это методы invokeLater() и invokeAndWait():

static void invokeLater(Runnable obj)

static void invokeAndWait(Runnable obj)

throws InterruptedException, InvocationTargetException

Здесь obj – это объект Runnable, метод run() которого будет вызываться потоком диспетчеризации событий. Единственное различие между этими двумя методами заключается в том, что метод invokeLater() возвращает результат немедленно, а метод invokeAndWait() ожидает возврата результата obj.run(). Вы можете использовать эти методы для вызова метода, создающего GUI для вашего Swing-приложения, или использовать их каждый раз, когда вам нужно будет изменить состояние GUI из кода, не выполняющегося в потоке диспетчеризации событий. Как правило, вам нужно будет использовать метод invokeLater(), как это было в предыдущей программе. А при создании исходного GUI для аплета вам понадобится метод invokeAndWait().

## Обработка событий

В предыдущем примере была показана базовая форма Swing-программы, однако в ней не хватает одной важной части: обработки событий. Поскольку метка JLabel не принимает входных данных от пользователя, она не генерирует события, поэтому и обработка событий не была нужна. Однако остальные Swing-компоненты реагируют на вводимые пользователем данные, вследствие чего возникает необходимость в обработке событий, которые возникают в результате таких взаимодействий. Например, событие генерируется тогда, когда таймер завершает отсчет. В любом случае, обработка событий является большой частью любого приложения, построенного на основе Swing.

Механизм обработки событий, используемый в Swing, ничем не отличается от механизма, применяемого в AWT.

// Обработка события в Swing-программе.

import java.awt.\*;

import java.awt.event.\*;

import javax.swing.\*;

class EventDemo {

JLabel jlab;

EventDemo() {

// Создание нового контейнера JFrame.

JFrame jfrm = new JFrame("An Event Example");

// JFrame jfrm = new JFrame("Пример обработки событий");

// Определение класса FlowLayout для диспетчера компоновки.

jfrm.setLayout(new FlowLayout());

// Установка исходных размеров фрейма.

jfrm.setSize(220, 90);

// Прекращение работы программы, если пользователь закрывает приложение.

jfrm.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

// Создаем две кнопки.

JButton jbtnAlpha = new JButton("Alpha");

JButton jbtnBeta = new JButton("Beta");

// Добавляем слушатель действий для Alpha.

jbtnAlpha.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent ae) {

jlab.setText("Alpha was pressed.");

// jlab.setText("Нажата кнопка Alpha.");

}

});

// Добавляем слушатель действий для Beta.

jbtnBeta.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent ae) {

jlab.setText("Beta was pressed.");

// jlab.setText("Нажата кнопка Beta.");

}

});

// Добавляем кнопки в панель содержимого.

jfrm.add(jbtnAlpha);

jfrm.add(jbtnBeta);

// Создаем текстовую метку.

jlab = new JLabel("Press a button.");

// jlab = new JLabel("Нажмите кнопку.");

// Добавляем метку в панель содержимого.

jfrm.add(jlab);

// Отображаем фрейм.

jfrm.setVisible(true);

}

public static void main(String args[]) {

// Создаем фрейм в потоке диспетчеризации событий.

SwingUtilities.invokeLater(new Runnable() {

public void run() {

new EventDemo();

}

});

}

}

Для начала обратите внимание на то, что программа теперь импортирует пакеты java.awt и java.awt.event. Пакет java.awt нужен по той причине, что в нем содержится класс FlowLayout, поддерживающий стандартный диспетчер компоновки потоков, который используется для размещения компонентов во фрейме. (Диспетчеры компоновки рассматривались в главе 24.) Пакет java.awt.event необходим потому, что он определяет интерфейс ActionListener и класс ActionEvent.

Конструктор EventDemo начинает работу с создания контейнера JFrame по имени jfrm. Затем он устанавливает диспетчер компоновки для панели содержимого jfrm в FlowLayout. Вспомните, что по умолчанию панель содержимого использует диспетчер компоновки BorderLayout. Однако для данного примера удобнее применить именно Flowlayout. Обратите внимание на то, что FlowLayout назначается с помощью следующего оператора:

jfrm.setLayout(new FlowLayout());

Как уже говорилось, раньше приходилось явным образом вызывать метод getContentPane(), чтобы задать диспетчер компоновки для окна содержимого. После выхода JDK 5 этого делать не нужно.

После определения размеров и стандартной операции при закрытии метод

EventDemo() создает две кнопки, как показано ниже:

JButton jbtnAlpha = new JButton("Alpha");

JButton jbtnBeta = new JButton("Beta");

Первая кнопка будет содержать текст “Alpha”, а вторая – “Beta”. Кнопки класса Swing являются экземплярами JButton. JButton предлагает несколько конструкторов. Одним из используемых здесь конструкторов является следующий:

JButton(String msg)

Параметр msg определяет строку, которая будет отображаться внутри кнопки.

При щелчке на кнопке генерируется событие ActionEvent. Таким образом, JButton предлагает метод addActionListener(), который используется для добавления слушателя событий. (JButton предлагает также метод removeActionListener() для удаления слушателя, однако он в этой программе не используется.)

После этого показанный ниже код добавляет слушатели событий для событий действий с кнопками:

// Добавляем слушатель действий для кнопки Alpha.

jbtnAlpha.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent ae) {

jlab.setText("Alpha was pressed.");

}

});

// Добавляем слушатель действий для кнопки Beta.

jbtnBeta.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent ae) {

jlab.setText("Beta was pressed.");

}

});

Здесь анонимные внутренние классы используются для того, чтобы предоставить обработчики событий двум кнопкам. Всякий раз при щелчке на кнопке строка, отображенная в jlab, изменяется в зависимости от того, на какой кнопке был произведен щелчок.

Затем кнопки добавляются в панель содержимого:

jfrm.add(jbtnAlpha);

jfrm.add(jbtnBeta);

И, наконец, в панель содержимого добавляется jlab, и окно становится видимым.

Когда вы запустите программу, то при каждом щелчке на кнопке в метке будет отображаться сообщение, показывающее, какая кнопка была нажата.

И еще одно замечание: следует помнить о том, что все обработчики событий наподобие actionPerformed() вызываются в потоке диспетчеризации событий. Таким образом, обработчик событий должен быстро дать результат, чтобы не допустить замедления работы приложения. Если вашему приложению нужно сделать что-то такое, для чего потребуется много времени, и что будет расцениваться как событие, то оно должно использовать отдельный поток.

## Создание Swing-аплета

Вторым типом программы, которая обычно использует классы Swing, является аплет.

Аплеты, созданные на основе Swing, подобны аплетам, созданным на основе AWT, но у них есть одно существенное отличие: Swing-аплет расширяет класс JApplet, а не Applet.

Таким образом, JApplet включает все функциональные возможности Applet и добавляет поддержку Swing. JApplet является Swing-контейнером верхнего уровня; это означает, что он не является наследником JComponent. Так как JApplet является контейнером верхнего уровня, он включает различные панели, описанные ранее. Это означает, что все компоненты добавляются в панель содержимого JApplet точно так же, как и компоненты, добавляемые в панель содержимого JFrame.

// Простой аплет, основанный на Swing

import javax.swing.\*;

import java.awt.\*;

import java.awt.event.\*;

/\*

Этот HTML-код можно использовать для запуска аплета:

<object code="MySwingApplet" width=220 height=90>

</object>

\*/

public class MySwingApplet extends JApplet {

JButton jbtnAlpha;

JButton jbtnBeta;

JLabel jlab;

// Инициализация аплета.

public void init() {

try {

SwingUtilities.invokeAndWait(new Runnable () {

public void run() {

makeGUI(); // инициализация GUI

}

});

} catch(Exception exc) {

System.out.println("Can't create because of "+ exc);

// System.out.println("Невозможно создать из-за "+ exc);

}

}

// Этому аплету не нужно переопределять методы start(), stop()

// или destroy().

// Настройка и инициализация GUI.

private void makeGUI() {

// Настройка аплета для использования компоновки потоков.

setLayout(new FlowLayout());

// Создание двух кнопок.

jbtnAlpha = new JButton("Alpha");

jbtnBeta = new JButton("Beta");

// Добавление слушателя действий для кнопки Alpha.

jbtnAlpha.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent le) {

jlab.setText("Alpha was pressed.");

// jlab.setText("Нажата кнопка Alpha.");

}

});

// Добавление слушателя действий для кнопки Beta.

jbtnBeta.addActionListener(new ActionListener() {

public void actionPerformed(ActionEvent le) {

jlab.setText("Beta was pressed.");

// jlab.setText("Нажата кнопка Beta.");

}

});

// Добавление кнопок в панель содержимого.

add(jbtnAlpha);

add(jbtnBeta);

// Создание текстовой метки.

jlab = new JLabel("Press a button.");

// jlab = new JLabel("Нажмите кнопку.");

// Добавление метки в панель содержимого.

add(jlab);

}

}

Необходимо сделать пару важных замечаний касательно аплетов. Во-первых, MySwingApplet расширяет JApplet. Как мы уже говорили, все аплеты, основанные на Swing, расширяют класс JApplet, а не Applet. Во-вторых, метод init() инициализирует Swing-компоненты в потоке диспетчеризации событий, устанавливая вызов метода makeGUI(). Обратите внимание, что это достигается путем использования метода invokeAndWait(), а не invokeLater(). Аплеты должны использовать метод invokeAndWait() потому, что метод init() не должен возвращать результат до тех пор, пока не будет выполнен весь процесс инициализации. По сути, метод start() нельзя вызывать прежде, чем закончится инициализация; это означает, что нужно полностью создать GUI.

Внутри метода makeGUI() создаются две кнопки и метка, а к кнопкам добавляются слушатели действий. Наконец, компоненты добавляются в панель содержимого. Несмотря на то что пример является довольно-таки простым, этот подход нужно применять при создании любого GUI Swing, который будет использован аплетом.

# Введение в JavaFX

Напомним, что в Java первой такой библиотекой была AWT. Вслед за AWT была разработана библиотека Swing, которая намного превосходила по возможностям свою предшественницу. Несмотря на успешность библиотеки Swing, создавать с ее помощью всевозможные визуальные эффекты, столь востребованные во многих современных приложениях, довольно затруднительно. Кроме того, изменения коснулись самих концептуальных основ проектирования пользовательских интерфейсов, что заставляло искать новые подходы к разработке GUI. Ответом разработчиков на запросы Jаvа-сообщества стала библиотека JavaFX, представляющая собой GUl-фреймворк следующего поколения. Данная глава содержит минимальный набор необходимых сведений, знание которых позволит вам в кратчайшие сроки приступить к работе с этой новой мощной системой.

Важно отметить, что развитие библиотеки JavaFX происходило в два этапа. Ранние версии JavaFX базировались на языке сценариев JavaFX Script. Однако в более поздних версиях, начиная с JavaFX 2.0, этот язык уже не поддерживается, и вместо него предлагается новый программный интерфейс для создания JаvаFХ-приложений полностью на языке Java. В JDK 7 (обновление 4) и более поздних версиях Java библиотека JavaFX входит в стандартный комплект поставки. Последней версией библиотеки, включенной в комплект JDK 8, является JavaFX 8. (При этом, чтобы положить начало согласованной нумерации версий Java и JavaFX, номера версий JavaFX от 3 до 7 были пропущены.)

Прежде чем продолжить, следует дать ответ на один вопрос, который естественным образом возникает в отношении JavaFX: предназначалась ли библиотека JavaFX для того, чтобы заменить собой Swing? По сути, так оно и было. Однако некоторое время Swing еще будет оставаться неотъемлемой частью программирования на языке Java. Это обусловлено наличием больших объемов унаследованного кода с графическим интерфейсом на основе Swing. Немаловажен и тот факт, что в настоящее время огромное количество программистов, освоивших технологию Swing, продолжают использовать ее в своих разработках. Тем не менее абсолютно очевидно, что будущее принадлежит JavaFX.

Иными словами, любой программист, пишущий программы на Java, должен владеть технологией JavaFX.

## Базовые понятия JavaFX

Прежде чем мы приступим к созданию приложения JavaFX, вам необходимо ознакомиться с основными понятиями и возможностями этой технологии. Несмотря на некоторое сходство JavaFX с другими графическими интерфейсами Java, такими как AWT и Swing, между ними имеются существенные различия. Аналогично Swing, компоненты JavaFX относятся к категории легковесных, а способы обработки событий просты и интуитивно понятны. Но если говорить об общих принципах организации библиотеки и взаимосвязи ее основных компонентов между собой, то JavaFX значительно отличается как от Swing, так и от AWT. Поэтому вам стоит внимательно изучить материал, изложенный в следующих разделах.

## Пакеты JavaFX

Библиотека JavaFX содержится в пакетах, имена которых начинаются с префикса javafx. К моменту написания данной книги API библиотеки включал более 30 пакетов. В качестве примера приведем имена следующих пакетов: javafx. application, javafx. stage, javafx. scene и javafx. scene. layout. В этой главе нам понадобятся всего лишь несколько пакетов JavaFX, однако вам стоит потратить некоторое время на краткое ознакомление с остальными пакетами этой библиотеки, поскольку спектр ее возможностей очень обширен.

## Классы Stage и Scene

В качестве центральной метафоры, на основе которой создавалась библиотека JavaFX, разработчики выбрали театральные подмостки (stage). Как и в любом реальном театре, подмостки служат сценической площадкой, на которой разыгрываются сцены (scenes). Образно говоря, подмостки, или театральная платформа, определяют пространственные границы для сцен, которые, в свою очередь, формируются из других элементов. Аналогично этому любое JаvаFХ-приложение содержит по крайней мере одну платформу и одну сцену. В JavaFX API эти элементы инкапсулируются классами Stage и Scene. Чтобы создать JаvаFХ-приложение, вы должны добавить в объект Stage хотя бы один объект Scene. Рассмотрим более детально, что собой представляют эти два класса.

Класс Stage – это контейнер верхнего уровня. Все приложения JavaFX автоматически получают доступ к одному контейнеру класса Stage, называемому основной платформой (primary stage). Основная платформа предоставляется исполняющей системой при запуске приложения. Несмотря на возможность создания нескольких платформ, в большинстве случаев одной платформы оказывается достаточно. Как уже отмечалось, класс Scene – это контейнер для элементов, составляющих сцену. Этими элементами могут быть кнопки и флажки, текст и графика. Для создания сцены вы будете добавлять эти элементы в экземпляр класса Scene.

## Узлы и графы сцены

Отдельные элементы сцены называют узлами (nodes). Например, узлом является кнопка. В то же время узлы сами по себе могут состоять из групп узлов. Кроме того, у любого узла могут быть дочерние узлы. Узел, имеющий дочерние узлы, называют родительским узлом (parent node), или узлом ветвления (branch node). Узлы, не имеющие дочерних узлов, являются оконечными и называются листьями (leaves). Совокупность всех узлов сцены называется графом сцены (scene graph) и образует дерево (tree), т.е. иерархическую структуру узлов.

Особую роль в графе сцены играет корневой узел, или корень (root). Им является узел верхнего уровня, и это единственный узел в графе сцены, не имеющий родительского узла. Таким образом, за исключением корневого узла, все остальные узлы имеют родителей и являются непосредственными или косвенными потомками корневого узла.

Класс Node является базовым для всех типов узлов. Сушествуют также другие классы, являющиеся прямыми или косвенными наследниками класса Node. В частности, таковыми являются классы Parent, Group, Region и Control.

## Панели компоновки

Библиотека JavaFX предоставляет нескольких панелей компоновки, с помощью которых можно управлять процессом размещения элементов в сцене. Например, класс FlowPane обеспечивает плавающую компоновку, а класс GridPane – табличную компоновку элементов в виде строк и столбцов. Также доступен ряд других менеджеров компоновки, например BorderPane (аналогичен компоновщику BorderLayout библиотеки AWT). Соответствующие классы находятся в пакете j avafx. scene. layout.

## Класс Application и жизненный цикл приложения

Приложение JavaFX должно быть подклассом класса Application, находящегося в пакете javafx.application. Таким образом, класс приложения должен расширять класс Application. Класс Application определяет три метода, управляющих жизненным циклом приложения: init () , start () и stop () , которые приложение может переопределить.

Метод init () вызывается в начале выполнения приложения. Он используется для инициализации всех необходимых переменных. Однако, как будет показано далее, его нельзя использовать для создания платформы или формирования сцены. Если никакая инициализация не требуется, данный метод можно не переопределять, поскольку его пустая версия предоставляется по умолчанию.

Метод start () вызывается после метода init (). Именно с него начинается работа приложения, и его можно использовать для конструирования и установки параметров сцены. Обратите внимание на то, что в качестве аргумента ему передается ссылка на объект Stage. Этот объект и есть та самая основная платформа, которую предоставляет исполняющая система. Заметьте также, что этот метод объявлен как абстрактный, и поэтому должен переопределяться в приложении.

При прекращении работы приложения вызывается метод stop (). Именно в нем организуется выполнение всех рутинных операций, связанных со сборкой мусора и освобождением ресурсов, захваченных приложением. Если такие действия не требуются, метод можно не переопределять, поскольку по умолчанию предоставляется его пустая версия.

## Запуск приложения JаvаFХ

Чтобы запустить автономное JаvаFХ-приложение, вы должны вызвать метод launch (), определенный в интерфейсе Application. Этот метод имеет две формы объявления. Ниже приведена та из них, которая используется в этой главе.

public static void launch(String ... аргументы)

Здесь параметр аргументы – список строк (возможно, пустой), обычно являющихся аргументами командной строки. Вызов метода launch () приводит к загрузке приложения, сопровождающейся последующими вызовами методов init () и start ().

Возврат из метода launch () происходит лишь тогда, когда приложение завершает работу. Данная версия метода launch () загружает приложение в класс, который расширяет класс Application и является точкой входа в приложение. Вторая форма метода launch () позволяет указать в качестве точки входа класс, отличный от того, в котором вызывается метод.

Прежде чем продолжить, необходимо сделать одно важное замечание: приложения, упакованные с помощью средства javafxpackager (или эквивалентного ему средства интегрированной среды разработки), не нуждаются в вызове метода launch (). Вместе с тем его включение в приложение во многих случаях упрощает процессы тестирования и отладки и позволяет использовать программу, не создавая JАR-файл. Поэтому в данной главе вызов метода launch () всегда включается в приложение.

## Каркас приложения JаvаFХ

Все приложения JavaFX создаются на основе одного и того же базового каркаса. Поэтому, прежде чем использовать другие возможности, полезно изучить, что собой представляет этот каркас. Это позволит не только продемонстрировать общую структуру JаvаFХ-приложения, но и показать, как запускается приложение и вызываются методы жизненного цикла. Приложение будет выводить на консоль сообщения, подсказывающие, когда именно вызывается тот или иной метод. Обратимся к приведенному ниже коду.

// Каркас приложения JavaFX

irnport javafx.application.\*;

irnport javafx.scene.\*;

irnport javafx.stage.\*;

irnport javafx.scene.layout.\*;

public class JavaFXSkel extends Application

public static void rnain(String[] args) {

Systern.out.println("Зaпycк приложения JavaFX");

// Запустить приложение JavaFX, вызвав метод launch()

launch(args);

// Переопределить метод init()

public void init() {

Systern.out.println("B теле метода init()");

// Переопределить метод start()

public void start(Stage rnyStage)

Systern.out.println("B теле метода start{)");

// Установить заголовок окна приложения

rnyStage.setTitle("Kapкac приложения JavaFX");

// Создать корневой узел. В данном случае

// используется плавающая компоновка, но возможны

// и другие варианты.

FlowPane rootNode = new FlowPane();

// Создать сцену

Scene rnyScene = new Scene(rootNode, 300, 200);

// Установить сцену на платформе

rnyStage.setScene(rnyScene) ;

// Отобразить платформу вместе с ее сценой

rnyStage.show() ; Оrображенне сценw

// Переопределить метод stop{).

public void stop() {

Systern.out.println("B теле метода stop()"};

Конечно, это совсем небольшое приложение, но его можно скомпилировать и выполнить.

Перейдем к подробному рассмотрению программы. Она начинается с импорта четырех пакетов. Первым импортируется пакет javafx.application, в котором содержится класс Application. В пакете javafx.scene находится класс Scene, а в пакете javafx. stage – класс Stage. Пакет javafx.scene.layout предоставляет ряд панелей компоновки. В программе используется панель FlowPane.

Далее создается класс приложения JavaFXSkel. Заметьте, что он расширяет класс Application. Как уже отмечалось, Application – это класс, наследованием которого создаются все приложения JavaFX. Класс JavaFXSkel содержит четыре метода. Первый из них – метод main () – используется для загрузки приложения посредством вызова метода launch (). Обратите внимание на то, что методу launch () передается параметр args, принимаемый методом main (). Такой подход является обычным, однако методу launch () можно передать другой набор параметров, в том числе пустой. Еще один важный момент: метод launch () требуется только автономным приложениям; во всех остальных случаях он не нужен. Однако в силу причин, приведенных выше, все программы в этой главе включают как метод main (). так и метод launch ().

Когда запускается приложение, исполнительная среда JavaFX в первую очередь вызывает метод init (). В данном случае этот метод просто выводит на консоль некоторое сообщение исключительно для того, чтобы сделать пример более наглядным, но обычно в нем выполняются все необходимые действия по инициализации приложения. Разумеется, если инициализация не требуется, то в переопределении метода init () нет никакой необходимости, поскольку по умолчанию всегда предоставляется его пустая реализация. Следует еще раз подчеркнуть, что метод init () не может быть использован для создания основной платформы или сцены GUI. Эти элементы должны конструироваться и отображаться методом start ().

Когда метод init () заканчивает свою работу, вызывается метод start (). Именно в этом методе создается начальная сцена и устанавливается основное окно приложения.

Проанализируем этот метод строка за строкой. Прежде всего обратите внимание на передаваемый ему параметр типа Stage. При вызове метода start () этот параметр получает ссылку на основную платформу приложения. Именно этот контейнер будет содержать сцену, используемую приложением.

После вывода на консоль сообщения, уведомляющего о начале работы метода start (), вызывается метод setTitle (), устанавливающий заголовок окна:

myStage.setTitle("Kapкac приложения JavaFX");

Поступать так вовсе необязательно, но в случае автономных приложений такая практика является общепринятой. Этот заголовок становится именем основного окна приложения.

На следующем этапе создается корневой узел сцены. Корневой узел является единственным узлом графа сцены, не имеющим родительского узла. В данном случае корневой узел – это объект типа FlowPane, но существуют и друтие классы, которые могут служить таким узлом.

FlowPane rootNode = new FlowPane();

Как уже отмечалось, панель FlowPane использует плавающую компоновку. Этот тип компоновки характеризуется тем, что элементы последовательно располагаются в строках с автоматическим переходом на следующую строку, если для размещения очередного элемента в текущей строке не хватает места. (Следовательно, здесь мы имеем дело с тем же типом компоновки, что и в случае класса FlowLayout, входящего в библиотеки AWT и Swing.) В данном примере элементы компонуются построчно в горизонтальном направлении, однако также возможна компоновка по вертикальным столбцам. И хотя в данном приложении этого не требуется, существует возможность задания других видов компоновки, например, компоновка элементов в горизонтальном или вертикальном направлении с указанием зазора между соседними элементами или их выравнивания.

В следующей строке кода корневой узел используется для создания объекта сцены:

Scene myScene = new Scene(rootNode, 300, 200);

В следующей строке программы объект myScene устанавливается в качестве сцены для платформы myStage:

myStage.setScene(myScene);

где setScene () – метод, определенный в классе Stage, который настраивает параметры сцены в соответствии с переданным ему аргументом.

В тех случаях, когда сцена в дальнейшем не используется, два предьщущих вызова могут быть объединены в один:

myStage.setScene(new Scene(rootNode, 300, 200));

В последующих примерах преимущество будет отдаваться именно этой форме вызова методов ввиду ее компактности.

Последняя строка метода start () отображает платформу и сцену:

myStage.show();

По существу, метод show () отображает окно, совместно создаваемое платформой и сценой.

При закрытии приложения его окно удаляется с экрана, и исполнительная система JavaFX вызывает метод stop (). В данном случае этот метод выводит сообщение на консоль, тем самым подтверждая факт своего вызова. Однако в реальных приложениях он, как правило, не выводит никакой информации. Кроме того, если выполнять какие-либо специальные действия при прекращении работы приложения не требуется, то отпадает и необходимость в переопределении метода stop (), поскольку его пустая реализация предоставляется по умолчанию.

## Поток выполнения приложения

В предыдущем обсуждении уже отмечалось, что метод init () не может быть использован для конструирования платформы или сцены. Эти элементы нельзя создавать и в конструкторе приложения. Причина в том, что и платформа, и сцена должна конструироваться в потоке приложения. При этом конструктор приложения и метод init () вызываются в основном потоке, который также называют стартовым потоком.

Вот почему их нельзя использовать для вызова конструкторов платформы и сцены. Вместо этого для создания начального графического интерфейса должен вызываться метод start (), как было сделано в примере, поскольку он вызывается в потоке приложения.

Кроме того, из потока приложения должны выполняться и любые изменения текущего состояния GUI. К счастью, в JavaFX события передаются программе через поток приложения. Поэтому для взаимодействия с графическим интерфейсом могут использоваться обработчики событий. Метод stop () также вызывается в потоке приложения.

## Использование кнопок и событий

Обработка событий играет важную роль, поскольку большинство элементов управления GUI генерируют события, которые обрабатываются вашей программой. Например, когда вы используете кнопки, флажки или списки, все они генерируют события. Обработка событий в JavaFX во многом напоминает обработку событий в Swing, о которой шла речь в предыдущей главе, но выполняется гораздо проще. Одним из наиболее часто используемых элементов управления является кнопка, и поэтому ее события приходится обрабатывать чаще других. Следовательно, использование кнопки для знакомства с обработкой событий в JavaFX будет весьма полезно. В связи с этим обработка событий и свойства кнопок JavaFX рассматриваются совместно.

## Основные сведения о событиях

Базовым классом событий JavaFX является класс Event, находящийся в пакете javafx.event. Класс Event наследует класс java.util.eventObject, а это означает, что события JavaFX разделяют общую функциональность с другими событиями Java.

Для класса Event определено несколько подклассов, из которых мы далее будем использовать только класс ActionEvent. Этот класс инкапсулирует события действий, генерируемые кнопкой.

Вообще говоря, используемый для обработки событий JavaFX подход в целом основан на модели делегатов. Чтобы обработать событие, вы должны сначала зарегистрировать обработчик, выступающий в качестве слушателя события. При наступлении какого-либо события вызывается соответствующий слушатель. Слушатель должен отреагировать на событие и после этого вернуть управление. В этом отношении управление событиями JavaFX осуществляется во многом так же, как и событиями Swing.

Обработка событий требует реализации интерфейса EventHandler, который также находится в пакете javafx.event. Обычно обработчики событий реализуются посредством использования анонимных внутренних классов или лямбда-выражений, но для этого могут использоваться и независимые классы, если такое решение больше подходит для конкретного приложения (например, в тех случаях, когда один и тот же обработчик должен обрабатывать события, поступающие от разных источников).

# Сервлеты

В этой главе речь пойдет о программы, которые выполняются на стороне сервера Web-соединения. Подобно сервлетах. Сервлетами (servlet) называются небольшие аплетам, которые динамические расширяют функциональные возможности Web-браузера, сервлеты динамически расширяют функциональность Web-сервера. Тема сервлетов является довольно обширной и ее невозможно рассмотреть полностью в рамках одной главы. Поэтому мы сосредоточимся на рассмотрении концепций, интерфейсов и классов, а также проанализируем некоторые примеры.

## Предварительные сведения

Чтобы разобраться с преимуществами сервлетов, вы должны иметь общее представление о том, как Web-браузеры и сервлеты работают сообща для предоставления содержимого пользователю. Рассмотрим запрос статической Web-страницы. Пользователь вводит в окне браузера URL-адрес (Uniform Resource Locator — унифицированный указатель информационного ресурса). Браузер генерирует HTTP-запрос к соответствующему Web-серверу. Web-сервер устанавливает соответствие между запросом и конкретным файлом. Этот файл возвращается браузеру в виде HTTP-отклика. HTTP-заголовок в отклике указывает тип содержимого. Для этой цели используется набор стандартов MIME (Multipurpose Internet Mail Extensions — многоцелевые расширения электронной почты).

Например, обычный текст в формате ASCII имеет MIME-тип text/plain. Исходный код HTML (Hypertext Markup Language — язык разметки гипертекста) имеет MIME-тип text/html.

Теперь рассмотрим динамическое содержимое. Предположим, что магазин, работающий в онлайновом режиме, использует базу данных для хранения информации о своей бизнес-деятельности. База данных (БД) может включать элементы для регистрации продаж, прайс-листов, наличия товара, счетов и т.п. Руководство магазина решило сделать так, чтобы эта информация была доступна покупателям через Web-страницы. Содержимое этих Web-страниц должно генерироваться динамически, чтобы отражать самую последнюю информацию в базе данных.

На ранних этапах существования системы Web сервер мог динамически формировать страницу, создавая отдельный процесс для обработки каждого запроса клиента. Чтобы получить необходимую информацию, процесс мог открывать соединения с одной или несколькими БД. Связь с сервером осуществлялась посредством интерфейса CGI (Common Gateway Interface — общий шлюзовой интерфейс). CGI позволял отдельным процессам считывать данные из HTTP-запроса и записывать данные в HTTP-отклик. Для написания CGI-программ применялись самые разные языки программирования. В их число входили C, C++ и Perl.

Однако у CGI имелись серьезные проблемы, связанные с производительностью. Этот интерфейс был дорогим в плане потребления ресурсов процессора и памяти, необходимых для создания отдельного процесса для каждого запроса клиента. Он был также дорогим в плане открытия и закрытия соединений с БД для каждого запроса клиента.

Помимо всего этого, работа CGI -программ зависела от конкретной платформы. Потому были предложены другие технологии, к числу которых относятся и сервлеты.

По сравнению с CGI сервлеты обладают некоторыми преимуществами. Во-первых, их производительность заметно выше. Сервлеты выполняются внутри адресного пространства Web-сервера. Чтобы выполнить обработку каждого запроса клиента, не обязательно создавать отдельный процесс. Во-вторых, работа сервлетов не зависит от платформы, поскольку все они пишутся на языке Java. В-третьих, диспетчер безопасности Java на сервере реализует серию ограничений для защиты ресурсов на компьютере-сервере. И, наконец, сервлету доступны абсолютно все функциональные возможности библиотек классов Java.

Сервлет может работать с аплетами, БД и другим программным обеспечением посредством сокетов и механизмов RMI, которые уже рассматривались в этой книге ранее.

## Жизненный цикл сервлета

Жизненный цикл сервлета определяют три основных метода: init(), service() и destroy(). Они реализуются каждым сервлетом и вызываются сервером в определенное время. Сейчас мы рассмотрим обычный пользовательский сценарий, который поможет понять, когда происходит вызов этих методов.

Во-первых, предположим, что пользователь ввел в окне браузера URL-адрес. На основании этого URL-адреса браузер генерирует HTTP -запрос, посылаемый соответствующему серверу.

Во-вторых, этот HTTP-запрос принимает Web-сервер. Сервер находит соответствие между запросом и конкретным сервлетом. Сервлет динамически принимается и загружается в адресное пространство сервера.

В-третьих, сервер вызывает метод init() сервлета. Этот метод вызывается только тогда, когда сервлет впервые загружается в память компьютера. Сервлету можно передавать параметры инициализации, поэтому он может конфигурировать себя самостоятельно.

В-четвертых, сервер вызывает метод service() сервлета. Этот метод вызывается для обработки HTTP-запроса. Вы увидите, что сервлет может считывать данные, содержащиеся в HTTP -запросе. Он может также сформулировать HTTP-отклик клиенту.

Сервлет остается в адресном пространстве и является доступным для обработки любых других HTTP-запросов, полученных от клиентов. Метод service() вызывается для каждого HTTP-запроса.

И, наконец, сервер может принять решение загрузить сервлет из памяти. Для принятия этого решения каждый сервер использует различные алгоритмы. Для освобождения ресурсов, таких как индексы файлов, выделенных для сервлета, сервер вызывает метод destroy(). Важные данные могут быть сохранены на постоянном носителе. Память, отведенная для сервлета и его объектов, впоследствии может быть утилизирована в процессе сборки мусора.