Справочник по программированию на Java  
вторая часть

Методическое пособие

**Центр компьютерного обучения «Специалист», 2020**

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, 2020 г.

Оглавление

[Многопоточное программирование 5](#_Toc49465568)

[Модель потоков Java 6](#_Toc49465569)

[Приоритеты потоков 7](#_Toc49465570)

[Синхронизация 8](#_Toc49465571)

[Обмен сообщениями 8](#_Toc49465572)

[Класс Thread и интерфейс Runnable 9](#_Toc49465573)

[Главный поток 9](#_Toc49465574)

[Создание потока 11](#_Toc49465575)

[Реализация Runnable 11](#_Toc49465576)

[Расширение Thread 13](#_Toc49465577)

[Создание множества потоков 14](#_Toc49465578)

[Использование isAlive() и join() 15](#_Toc49465579)

[Приоритеты потоков 15](#_Toc49465580)

[Синхронизация 18](#_Toc49465581)

[Использование синхронизированных методов 19](#_Toc49465582)

[Оператор synchronized 21](#_Toc49465583)

[Межпотоковые коммуникации 22](#_Toc49465584)

[Взаимная блокировка 26](#_Toc49465585)

[Приостановка, возобновление и останов потоков 28](#_Toc49465586)

[Использование многопоточности 31](#_Toc49465587)

[Работа с реляционными базами данных (JDBC) 31](#_Toc49465588)

[MySQL Установка драйвера Connector/J 32](#_Toc49465589)

[Подключение к базе данных 33](#_Toc49465590)

[Выполнение команд обновления (executeUpdate) 34](#_Toc49465591)

[PreparedStatement 35](#_Toc49465592)

[Добавление, изменение и удаление данных 36](#_Toc49465593)

[Метод executeQuery. Получение данных 38](#_Toc49465594)

[Введение в JavaFX 40](#_Toc49465595)

[Базовые понятия JavaFX 41](#_Toc49465596)

[Пакеты JavaFX 41](#_Toc49465597)

[Классы Stage и Scene 41](#_Toc49465598)

[Узлы и графы сцены 42](#_Toc49465599)

[Панели компоновки 42](#_Toc49465600)

[Класс Application и жизненный цикл приложения 42](#_Toc49465601)

[Запуск приложения JаvаFХ 43](#_Toc49465602)

[Каркас приложения JаvаFХ 43](#_Toc49465603)

[Поток выполнения приложения 46](#_Toc49465604)

[Использование кнопок и событий 47](#_Toc49465605)

[Основные сведения о событиях 47](#_Toc49465606)

[Wеb приложения 48](#_Toc49465607)

[Сервлеты 48](#_Toc49465608)

[Жизненный цикл сервлета 49](#_Toc49465609)

[Java Server Pages 50](#_Toc49465610)

[Создание Web проекта в Eclipse 52](#_Toc49465611)

[Синтаксис тэгов JSP 55](#_Toc49465612)

[Использование классов Java в JSP 59](#_Toc49465613)

[Получение данных элементов управления формы 59](#_Toc49465614)

[Передача данных на страницу 60](#_Toc49465615)

[Expression Language 63](#_Toc49465616)

[Встроенные объекты Expression Language 66](#_Toc49465617)

[JSP Standard Tag Library 66](#_Toc49465618)

# 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.

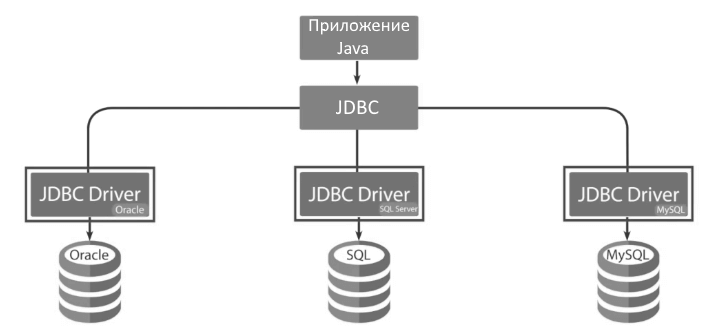
Если вы создадите слишком много потоков, вы можете даже снизить производительность всей программы вместо того, чтобы повысить ее. Помните, что переключение контекстов между потоками требует определенных накладных расходов. Если вы создадите очень много потоков, больше времени процессора будет затрачено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

# Работа с реляционными базами данных (JDBC)

Для хранения данных мы можем использовать различные базы данных - Oracle, MS SQL Server, MySQL, Postgres и т.д. Все эти системы упраления базами данных имеют свои особенности. Главное, что их объединяет это взаимодействие с хранилищем данных посредством команд SQL. И чтобы определить единый механизм взаимодействия с этими СУБД в Java еще начиная с 1996 был введен специальный прикладной интерфейс API, который называется JDBC.

Данный API входит в состав, в частности, для работы с JDBC в программе Java достаточно подключить пакет java.sql. Для работы в Java EE есть аналогичный пакет javax.sql, который расширяет возможности JDBC.

Однако не все базы данных могут поддерживаться через JDBC. Для работы с определенной СУБД также необходим специальный драйвер. Каждый разработчик определенной СУБД обычно предоставляет свой драйвер для работы с JDBC. То есть для работы с MySQL вам потребуется специальный драйвер именно для MySQL. Как правило, большинство драйверов доступны в свободном доступе на сайтах соответствующих СУБД. Обычно они представляют JAR-файлы. Преимущество JDBC состоит в том, что она обеспечивает абстрагирование от строения конкретной базы данных, но при этом код использует унифицированный интерфейс, который един для всех.



На процесс компиляции необходимость работы с БД никак не сказывается, но влияет на процесс запуска программы. При запуске программы в командной строке необходимо указать путь к JAR-файлу драйвера после параметра -classpath.

java -classpath путь\_к\_файлу\_архива\_драйвера:путь\_к\_программе главный\_класс\_программы

Например:

java -classpath mysql-connector-java-5.1.18.jar;. Program

В принципе мы можем и не использовать параметр -classpath, и запустить програму на выполнение обычным способом с помощью команды "java Program". Но в этом случае путь к драйверу должен быть добавлен в переменную CLASSPATH.

## MySQL Установка драйвера Connector/J

Одной из наиболее популярных СУБД в связке с Java является MySQL. Поэтому рассмотрим, как можно работать с MySQL в Java.

Для работы с MySql в Java необходимо установить официальный драйвер MySQL Connector/J, загрузив его по адресу https://dev.mysql.com/downloads/connector/j/. Внутри архив будет jar файл, который и представляет из себя искомый драйвер.

Проверим работу драйвера с помощью простой программы:

package ru.specialist.db;

import java.sql.\*;

public class Program {

public static final String DRIVER\_NAME = "com.mysql.jdbc.Driver";

public static final String CONNECTION\_STRING =

"jdbc:mysql://localhost/web?user=root&password=demo";

public static void main(String[] args) throws

ClassNotFoundException, SQLException {

Class.forName(DRIVER\_NAME);

}

}

Метод Class.forName() в качестве параметра принимает строку, которая представляет полный путь к классу драйвера с учетом всех пакетов. В случае MySQL это путь "com.mysql.jdbc.Driver". Таким образом, Метод Class.forName загружает класс драйвера, который будет использоваться.

Скомпилируйте и запустите программу на выполнение, не забыв включить jar файл с драйвером в classpath.

## Подключение к базе данных

Вначале создайте на сервере MySQL пустую базу данных, которую назовите web и с которой мы будет работать в приложении на Java. Для создания базы данных применяется выражение SQL:

CREATE DATABASE web;

Его можно выполнить либо из консольного клиента MySQL Command Line Client, либо из графического клиента MySQL Workbench, которые устанавливаются вместе с сервером MySQL. Выполните в указанной базе скрипт web.sql для создания таблиц и заполнения начальными данными.

Для подключения к базе данных необходимо создать объект java.sql.Connection. Для его создаия применяется метод:

Connection DriverManager.getConnection(url, username, password)

Метод DriverManager.getConnection в качестве параметров принимает адрес источника данных, логин и пароль. В качестве логина и пароля передаются логин и пароль от сервера MySQL. Адрес локальной базы данных MySQL указывается в следующем формате:

jdbc:mysql://localhost/{database}

Пример создания подключения к созданной выше локальной базе данных web:

1

Connection connection = DriverManager.getConnection("jdbc:mysql://localhost/web", "root", "demo");

После завершения работы с подключением его следует закрыть с помощью метода close():

package ru.specialist.db;

import java.sql.\*;

public class Program {

public static final String DRIVER\_NAME = "com.mysql.jdbc.Driver";

public static final String CONNECTION\_STRING =

"jdbc:mysql://localhost/web?user=root&password=demo";

public static void main(String[] args) throws ClassNotFoundException, SQLException {

Class.forName(DRIVER\_NAME);

Connection conn = DriverManager.getConnection(CONNECTION\_STRING);

// работа с базой данных

conn.close();

}

}

## Выполнение команд обновления (executeUpdate)

Для взаимодействия с базой данных приложение отправляет серверу MySQL команды на языке SQL. Чтобы выполнить команду, вначале необходимо создать объект Statement.

Для его создания у объекта Connection вызывается метод createStatement():

Statement statement = conn.createStatement();

Для выполнения команд SQL в классе Statement определено три метода:

executeUpdate: выполняет такие команды, как INSERT, UPDATE, DELETE, CREATE TABLE, DROP TABLE. В качестве результата возвращает количество строк, затронутых операцией (например, количество добавленных, измененных или удаленных строк), или 0, если ни одна строка не затронута операцией или если команда не изменяет содержимое таблицы (например, команда создания новой таблицы)

executeQuery: выполняет команду SELECT. Возвращает объект ResultSet, который содержит результаты запроса.

execute(): выполняет любые команды и возвращает значение boolean: true - если команда возвращает набор строк (SELECT), иначе возвращается false.

## PreparedStatement

Кроме класса Statement в java.sql мы можем использовать для выполнения запросов еще один класс - PreparedStatement. Кроме собственно выполнения запроса этот класс позволяет подготовить запрос, отформатировать его должным образом.

На месте параметров в тексте SQL запроса подставляются знаки ?.

Чтобы связать отдельные знаки подстановки с конкретными значениями у класса PreparedStatement определен ряд методов для различных типов данных. Все методы, которые поставляют значения вместо знаков подстановки, в качестве первого параметра принимают порядковый номер знака подстановки (нумерация начинается с 1), а в качестве второго параметра - собственно значение, которое вставляется вместо знака подстановки.

Например, первый знак подстановки ? в выражении sql представляет значение для столбца title, который хранит строку. Поэтому для связи значения с первым знаком подстановки применяется метод preparedStatement.setString(1, title).

Второй знак подстановки должен передавать значение для столбца Length, который хранит целые числа. Поэтому для вставик значения используется метод preparedStatement.setInt(2, length)

Кроме setString и setInt PreparedStatement имеет еще ряд подобных методов, которые работают подобным образом. Некоторые из них:

setBigDecimal

setBoolean

setDate

setDouble

setFloat

setLong

setNull

setTime

Для выполнения запроса PreparedStatement имеет три метода:

boolean execute(): выполняет любую SQL-команду

ResultSet executeQuery(): выполняет команду SELECT, которая возвращает данные в виде ResultSet

int executeUpdate(): выполняет такие SQL-команды, как INSERT, UPDATE, DELETE, CREATE и возвращает количество измененных строк

При этом в отличие от методов Statement эти методы не принимают SQL-выражение.

## Добавление, изменение и удаление данных

Для добавления, редактирования и удаления данных мы можем иcпользовать рассмотренный в прошлой теме метод executeUpdate. С помощью результата метода мы можем проконтролировать, сколько строк было добавлено, изменено или удалено.

import java.sql.\*;

import static java.lang.System.out;

import java.util.Scanner;

public class Program {

public static final String DRIVER\_NAME =

"com.mysql.jdbc.Driver";

public static final String CONNECTION\_STRING =

"jdbc:mysql://localhost:3306/web?user=root&password=demo";

public static void main(String[] args)

throws ClassNotFoundException, SQLException {

Class.forName(DRIVER\_NAME);

Scanner sc = new Scanner(System.in);

out.print("Название курса: ");

String title = sc.nextLine().trim();

out.print("Длительность курса: ");

int length = sc.nextInt();

if (sc.hasNextLine()) sc.nextLine();

out.print("Описание курса: ");

String description = sc.nextLine().trim();

try (Connection conn = DriverManager.getConnection(CONNECTION\_STRING))

{

try

{

conn.setTransactionIsolation(Connection.TRANSACTION\_READ\_COMMITTED);

conn.setAutoCommit(false);

String sql = "INSERT INTO Courses (title,length, description)"

+ " VALUES (?, ?, ?)";

PreparedStatement cmd =

conn.prepareStatement(sql, Statement.RETURN\_GENERATED\_KEYS);

cmd.setString(1, title);

cmd.setInt(2, length);

cmd.setString(3, description);

if (cmd.executeUpdate() == 1)

{

// out.printf("Курс добавлен.");

conn.commit();

try (ResultSet keys = cmd.getGeneratedKeys())

{

if (keys.next())

{

int id = keys.getInt(1);

out.printf("Курс добавлен. id: %d\n", id);

}

} // keys.close()\*/

}

}

catch(SQLException ex)

{

conn.rollback();

}

} //conn.close();

}

}

Как видно из примеров, не так сложно взаимодействовать с базой данных. Достаточно передать в метод executeUpdate нужную команду SQL.

## Метод executeQuery. Получение данных

Для выборки данных с помощью команды SELECT применяется метод executeQuery:

ResultSet executeQuery("SQL COMMAND")

Метод возвращает объект ResultSet, который содержит все полученные данные. Как эти данные получить?

В объекте ResultSet итератор устанавливается на позиции перед первой строкой. И чтобы переместиться к первой строке (и ко всем последующим) необходимо вызвать метод next(). Пока в наборе ResultSet есть доступные строки, метод next будет возвращать true. Типичное перемещение по набору строк:

package ru.specialist.db;

import java.sql.\*;

import java.util.Scanner;

public class Program {

public static final String DRIVER\_NAME = "com.mysql.jdbc.Driver";

public static final String CONNECTION\_STRING =

"jdbc:mysql://localhost/web?user=root&password=demo";

public static void main(String[] args) throws ClassNotFoundException, SQLException {

Class.forName(DRIVER\_NAME);

Connection conn = DriverManager.getConnection(CONNECTION\_STRING);

CallableStatement sp = conn.prepareCall("call countCourses(?)");

// sp.setInt(1, x)

sp.execute();

int count = sp.getInt(1);

System.out.printf("Всего курсов: %d\n", count);

System.out.print("Поиск: ");

Scanner sc = new Scanner(System.in);

String search = sc.nextLine().trim();

String sql = "SELECT title, length FROM courses WHERE title LIKE ? ORDER BY title";

PreparedStatement cmd = conn.prepareStatement(sql);

cmd.setString(1, "%"+search+"%");

ResultSet result = cmd.executeQuery();

while (result.next()) {

String title = result.getString("title");

int length = result.getInt("length");

System.out.printf("%-30s : %d\n", title, length);

}

result.close();

conn.close();

}

}

То есть пока в resultSet есть доступные строки, будет выполняться цикл while, который будет переходить к следующей строке в наборе.

После перехода к строке мы можем получить ее содержимое. Для этого у ResultSet определен ряд методов. Некоторые из них:

getBoolean() возвращает значение boolean

getDate() возвращает значение Date

getDouble() возвращает значение double

getInt() возвращает значение int

getFloat() возвращает значение float

getLong() возвращает значение long

getNString() возвращает значение String

getString() возвращает значение String

В зависимости от того, данные какого тип хранятся в том или ином столбце, мы можем использовать тот или иной метод. Каждый из этих методов имеет две версии:

int getInt(int columnIndex)

int getInt(String columnLabel)

Первая версия получает данные из столбца с номером columnIndex. Вторая версия получает данные из столбца с названием columnLabel.

Отметим, что индексация столбцов начинается с 1, а не с 0.

# Введение в JavaFX

Напомним, что в Java первой библиотекой для создания настольных (оконных) приложений 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).

# Wеb приложения

Речь пойдет о программах, которые выполняются на стороне сервера Web-соединения.. Сервлетами (servlet) называются классы, динамически расширяющие 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.

## Сервлеты

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.

## 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.

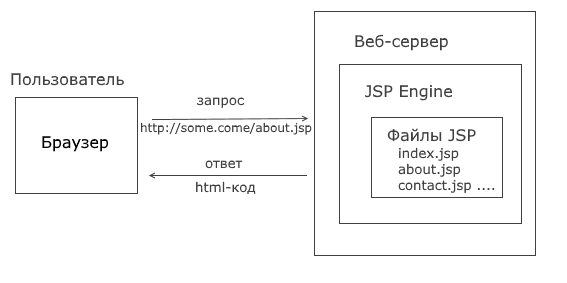
И, наконец, сервер может принять решение загрузить сервлет из памяти. Для принятия этого решения каждый сервер использует различные алгоритмы. Для освобождения ресурсов, таких как индексы файлов, выделенных для сервлета, сервер вызывает метод destroy(). Важные данные могут быть сохранены на постоянном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

## Java Server Pages

Java Server Pages представляет технологию, которая позволяет создавать динамические веб-страницы. Изначально JSP (вместе с сервлетами) на заре развития Java EE являлись доминирующим подходом к веб-разработке на языке Java. И хотя в настоящее время они уступило свое место другой технологии - JSF, тем не менее JSP продолжают широко использоваться.

По сути Java Server Page или JSP представляет собой html-код с вкраплениями кода Java. В то же время станицы jsp - это не стандартные html-страницы. Когда приходит запрос к определенной странице JSP, то сервер обрабатывает ее, генерирует из нее код html и отправляет его клиенту. В итоге пользователь после обращения к странице JSP видит в своем браузере обычную html-страницу.

Как и обычные статические веб-страницы, файлы JSP необходимо размещать на веб-сервере, к которому обычные пользователи могут обращаться по протоколу http, например, набирая в адресной строке браузера нужный адрес. Однако, чтобы сервер мог обрабатывать файлы JSP, он должен использовать движок JSP (JSP engine), который также называют JSP-контейнером. Есть множество движков JSP, и все они реализуют одну и ту же спецификацию и в целом работают одинаково. Однако тем не менее при переносе кода с одного веб-сервера на другой могут потребоваться небольшие изменения.



Для работы с JSP мы будем использовать Apache Tomcat, который одновременно является и веб-сервером и контейнером сервлетов и JSP.

Создадим простейшую страницу JSP. Для этого где-нибудь на жестком диске определим файл index.jsp. Все станицы JSP имеют расширение jsp. Откроем этот файл в любом текстовом редакторе и определим в нем следующий код:

<%

String header = "Hello from Apache Tomcat";

%>

<!DOCTYPE html>

<html>

    <head>

        <meta charset="UTF-8" />

        <title>First JSP App</title>

    </head>

    <body>

        <h2><%= header %></h2>

        <p>Today <%= java.time.LocalDate.now() %></p>

    </body>

</html>

С помощью тегов <% ... %> мы можем определить код Java на странице JSP. В данном случае мы просто определяем переменную типа String, которая называется header.

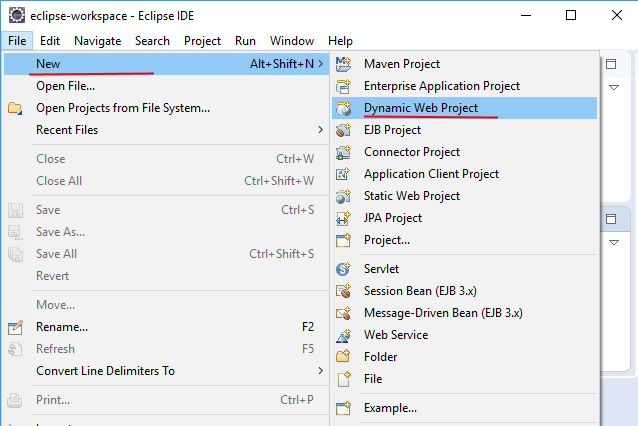
Затем идет стандартный код страницы html. Чтобы внедрить код java внутрь html-страницы применяются теги <%= %> - после знака равно указывается выражение Java, результат которого будет выводиться вместо этих тегов. В данном случае, используются две таких вставки. Первая вставка - значение переменной header, которая была определена выше. Вторая вставка - выражение java.timeLocalDate.now(), которое возвращает текущую дату.

Для тех, кто знаком с веб-разработкой на PHP, это может напоминать оформление файлов php, которые также содержать код html и код php.

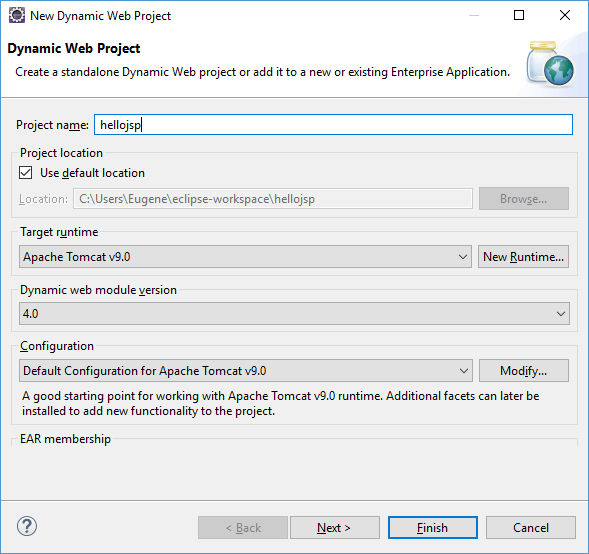
## Создание Web проекта в Eclipse

Хотя мы вполне можем создавать страницы JSP в обычном текстовом редакторе, но также мы можем использовать и среды разработки, например, Eclipse.

Для работы с JSP в Eclipse создадим новый проект. Для этого перейдем к пункту меню File -> New -> Dynamic Web Project



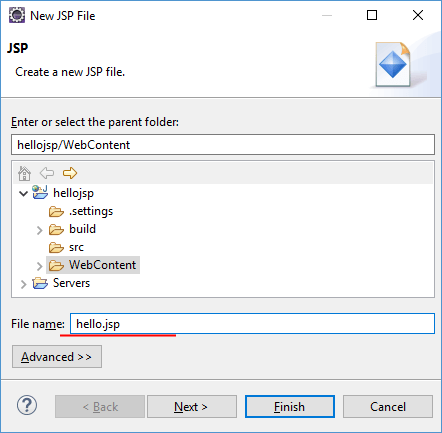
Далее нам откроется окно создания нового проекта.

**

В поле **Project Name** укажем для проекта какое-нибудь имя, например, **hellojsp** и нажмем на кнопку Finish для создания проекта.

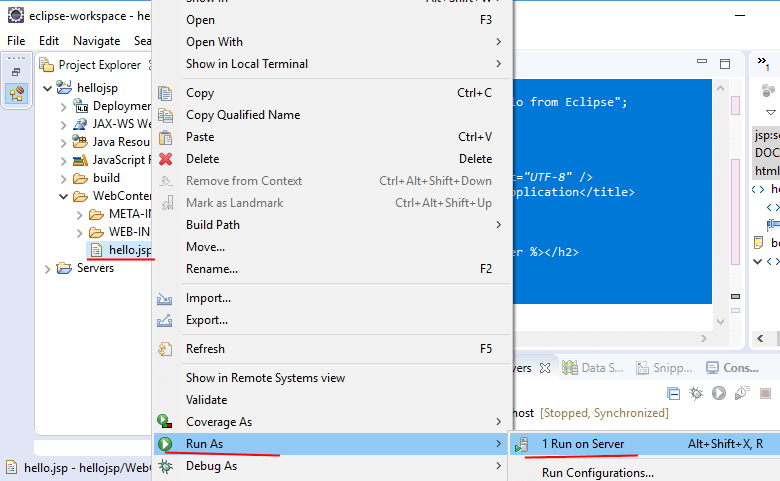
После этого будет создан новый проект, который по умолчанию будет иметь некоторое содержимое. Все файлы с исходным кодом, в том числе jsp-страницы, помещаются в проекте в каталог **WebContent**. Нажмем на него правой кнопкой мыши и в контекстном меню выберем пункт **New -> JSP File**

Далее откроется окно добавления нового файла:



Укажем для нового файла имя, например, hello.jsp и нажмем на кнопку Finish. После этого в проект будет добавлен новый файл hello.jsp.

Теперь запустим данный файл. Для этого нажмем на него правой кнопкой мыши и в контекстном меню выберем **Run As -> Run on Server**:



Далее откроется окно для выбора сервера. Среди серверов уже значится Apache Tomcat 9.0, соответственно он и будет использоваться. Опционально можно отметить поле **Always use this server when running this project**, что позволит в дальнейшем упростить запуск проекта.

Следует учитывать, что если ранее уже был запущен Tomcat, например, при автозапуске и вручную через панель служб, то надо его остановить. Иначе при запуске из Eclipse мы столкнемся с ошибкой, что Eclipse не может запустить Tomcat, так как его порт уже занят.

## Синтаксис тэгов JSP

Содержимое страницы JSP фактически делится на код html (а также css/javascript) и код на языке java. Для вставки кода Java на страницу JSP можно использовать пять основных элементов:

Выражения JSP (JSP Expression)

Скриплет JSP (JSP Scriplet)

Объявления JSP (JSP Declaration)

Директивы JSP (JSP Directive)

Комментарии JSP

JSP Expression

JSP Expression представляет выражение, заключенное между тегами <%= и %>. При обращении к JSP вычисляется значение этого выражения.

Например, определим следующую страницу JSP:

<!DOCTYPE html>

<html>

    <head>

        <meta charset="UTF-8" />

        <title>First JSP App</title>

    </head>

    <body>

        <p>2 + 2 = <%= 2 + 2 %></p>

        <p>5 > 2 = <%= 5 > 2 %></p>

        <p><%= new String("Hello").toUpperCase() %></p>

        <p>Today <%= java.time.LocalDate.now() %></p>

    </body>

</html>

Здесь используются четыре JSP-выражения. Первое выражение - простая арифметическая операция сложения. При обработке страницы вместо

<p>2 + 2 = <%= 2 + 2 %></p>

будет сгенерирована следующая html-разметка:

<p>2 + 2 = 4</p>

Второе выражение - операция сравнения во многом аналогична. Третье выражение - создание объекта String и вызов у него метода toUpperCase(), который возвращает строку в верхнем регистре. То есть выражение также может представлять вызов метода.

И четвертое выражение - вызов статического метода класса LocalDate, который создает объект с текущей датой.

Когда придет запрос к этой странице, из нее будет сгенерирован следующий код:

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title>First JSP App</title>

</head>

<body>

<p>2 + 2 = 4</p>

<p>5 > 2 = true</p>

<p>HELLO</p>

<p>Today Fri Aug 15 12:37:26 MSK 2020</p>

</body>

</html>

**JSP Scriplet** представляет одну или несколько строк на языке Java. Скриплет заключается внутри следующих тегов:

<%

код Java

%>

Например, определим следующую страницу JSP:

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title>First JSP App</title>

</head>

<body>

<%

for(int i = 1; i < 5; i++){

out.println("<br>Hello " + i);

}

%>

</body>

</html>

В данном случае скриплет представляет цикл for, в котором генерируется вывод с помощью метода out.println(). В итоге в браузере будет выведено четыре раза слово Hello с соответствующей цифрой.

Другой пример - определим переменную и массив и выведем их содержимое на страницу:

<%

    String[] people = new String[]{"Tom", "Bob", "Sam"};

    String header = "Users list";

%>

<!DOCTYPE html>

<html>

    <head>

        <meta charset="UTF-8" />

        <title>First JSP App</title>

    </head>

    <body>

        <h3><%= header %></h3>

        <ul>

        <%

            for(String person: people){

                out.println("<li>" + person + "</li>");

            }

        %>

        </ul>

    </body>

</html>

**JSP Declaration**

JSP Declaration позволяют определить метод, который мы затем можем вызывать в скриплетах или в JSP-выражениях. Определение метода помещается между тегами <%! и %>. Например, определим следующую JSP-страницу:

<%!

int square(int n){

return n \* n;

}

%>

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title>First JSP App</title>

</head>

<body>

<p><%= square(6) %></p>

<ul>

<%

for(int i = 1; i <= 5; i++){

out.println("<li>" + square(i) + "</li>");

}

%>

</ul>

</body>

</html>

В данном случае метод square возвращает квадрат числа. Затем этот метод используется в выражении JSP и в скриплете в цикле for.

**Директивы**

Директивы предназначены для установки условий, которые применяются ко всей странице JSP. Наиболее используемая директива - это директива page. Например, с помощью атрибута import этой директивы мы можем импортировать пакеты или отдельные классы на страницу jsp.

Например, в начале использовалось выражение java.time.LocalDate. Но мы можем использовать данное выражение на странице многократно, либо использовать другие классы из пакета java.time. И в этом случае мы можем импортировать нужные нам классы или пакеты:

<%@ page import="java.time.LocalDate " %>

Импорт всего пакета:

<%@ page import="java.time.\*" %>

Если необходимо импортировать несколько классов и(или) пакетов, то они перечисляются через запятую:

<%@ page import="java.time.LocalData, java.text.\*" %>

Другой полезный и часто используемый атрибут - pageEncoding, который задает кодировку jsp. Например:

<%@ page import="java.time.LocalDate" pageEncoding="UTF-8" %>

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>JSP Application</title>

</head>

<body>

<h2>Сегодня: <%= LocalDate.now() %></h2>

</body>

</html>

**Комментарии**

Комментарии JSP добавляются с помощью тега

<%-- Текст\_комментария --%>

## Использование классов Java в JSP

Страницы JSP могут включать некоторый код на Java. Таким образом, мы можем определять прямо в JSP переменные, методы, некоторую логику. Однако если программной логики на Java довольно много, то лучше выносить ее в отдельные классы. Однако в данном случае может возникнуть вопрос, как организовать взаимодействие классов на Java и JSP.

Для этого достаточно создать необходимые вам классы, как вы делали это и раньше, а затем импортировать их на страницу.

## Получение данных элементов управления формы

Станицы JSP могут получать отправленные данные, например, через параметры или в виде отправленных форм, так же, как это происходит в сервлете. Для этого внутри страницы jsp доступен объект **request**, который позволяет получить данные посредством следующих методов:

getParameter(String param) –

возвращает значение определенного параметра, название которого передается в метод. Если указанного параметра в запросе нет, то возвращается значение null.

getParameterValues(String param) -

возвращает массив значений, который представляет определенный параметр. Если указанного параметра в запросе нет, то возвращается значение null.

<%@ page language="java" contentType="text/html; charset=UTF-8"

pageEncoding="UTF-8"%>

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>Hello JSP page</title>

</head>

<body>

<h1>

<%

String userName = null, hello;

if (request.getMethod().equals("POST")) {

request.setCharacterEncoding("utf-8");

userName = request.getParameter("userName");

}

if (userName == null || userName.trim().isEmpty()) {

hello = "Привет";

userName = "";

} else

hello = String.format("Привет, %s!", userName);

out.print(hello);

%>

</h1>

<form method="POST">

<label>Ваше Имя: </label>

<input type="text" name="userName" value="<%= userName %>">

<input type="submit" value="Привет">

</form>

</body>

</html>

## Передача данных на страницу

Нередко страница jsp обрабатывает запрос вместе сервлетом. В этом случае сервлет определяет логику, а jsp - визуальную часть. И при обработке запроса сервлет может перенаправить дальнейшую обработку странице jsp. Соответственно может возникнуть вопрос, как передать данные из сервлета в jsp?

Есть несколько способов передачи данных из сервлета в jsp, которые заключаются в использовании определенного контекста или scope. Есть несколько контекстов для передачи данных:

**request** (контекст запроса): данные сохраняются в HttpServletRequest

**session** (контекст сессии): данные сохраняются в HttpSession

**application** (контекст приложения): данные сохраняются в ServletContext

Данные из контекста запроса доступны только в пределах текущего запроса. Данные из контекста сессии доступны только в пределах текущего сеанса. А данные из контекста приложения доступны постоянно, пока работает приложение.

Но вне зависимости от выбранного способа передача данных осуществляется с помощью метода **setAttribute(name, value)**, где name - строковое название данных, а value - сами данные, которые могут представлять различные данные.

Наиболее распространенный способ передачи данных из сервлета в jsp представляют атрибуты запроса. То есть у объекта HttpServletRequest, который передается в сервлет, вызывается метод **setAttribute()**. Этот метод устанавливает атрибут, который можно получить в jsp.

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>JSP Application</title>

</head>

<body>

<p>Name: ${name}</p>

<p>Age: ${age}</p>

</body>

</html>

import java.io.IOException;

import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

@WebServlet("/hello")

public class HelloServlet extends HttpServlet {

    protected void doGet(HttpServletRequest request, HttpServletResponse response)

            throws ServletException, IOException {

        request.setAttribute("name", "Tom");

        request.setAttribute("age", 34);

        getServletContext().getRequestDispatcher("/basic.jsp").forward(request, response);

    }

}

Сервлет устанавливает два атрибута - "name" и "age" через объект HttpServletRequest и затем перенаправляет запрос странице basic.jsp.

Использование контекста приложения представляет применение объекта ServletContext, который можно получить в сервлете с помощью метода getServletContext():

import java.io.IOException;

import javax.servlet.ServletContext;

import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;

@WebServlet("/hello")

public class HelloServlet extends HttpServlet {

    protected void doGet(HttpServletRequest request, HttpServletResponse response)

            throws ServletException, IOException {

        ServletContext selvletContext = getServletContext();

        selvletContext.setAttribute("name", "Tom");

        selvletContext.setAttribute("age", 35);

        getServletContext().getRequestDispatcher("/basic.jsp").forward(request, response);

    }

}

Подобным образом можно передать данные в jsp через сессию:

|  |
| --- |
| import java.io.IOException;  import javax.servlet.ServletException;  import javax.servlet.annotation.WebServlet;  import javax.servlet.http.HttpServlet;  import javax.servlet.http.HttpServletRequest;  import javax.servlet.http.HttpServletResponse;  import javax.servlet.http.HttpSession;    @WebServlet("/hello")  public class HelloServlet extends HttpServlet {        protected void doGet(HttpServletRequest request, HttpServletResponse response)              throws ServletException, IOException {            HttpSession session = request.getSession();          session.setAttribute("name", "Tom");          session.setAttribute("age", 34);            getServletContext().getRequestDispatcher("/basic.jsp").  forward(request, response);      }  } |

## Expression Language

Expression Language или сокращенно EL предоставляет компактный синтаксис для обращения к массивам, коллекциям, объектам и их свойствам внутри страницы jsp. Он довольно прост. Вставку открывает знак $, затем в фигурные скобки {} заключается выводимое значение:

${attribute}

${object.property}

Откуда эти данные берутся? EL пытается найти значения для этих данных во всех доступных контекстах.

И EL просматривает все эти контексты в следующем порядке:

Контекст страницы (данные сохраняются в PageContext)

Контекст запроса

Контекст сессии

Контекст приложения

Соответственно, если контексты запроса и сессии содержат атрибут с одним и тем же именем, то будет использоваться атрибут из контекста запроса. Затем найденное значение (если оно было найдено) конвертируется в строку и выводится на страницу. Следует отметить, что мы не можем определить данные на странице, например, с помощью скриплета и затем вывести через EL:

<%

String name = "Tom";

%>

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>JSP Application</title>

</head>

<body>

<p>Name: ${name}</p>

</body>

</html>

Такой способ не будет работать. Если мы хотим определить данные непосредственно на страницы, то их затем необходимо включить в контекст страницы, который доступен через переменную pageContext:

<%

pageContext.setAttribute("name", "Tom");

%>

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>JSP Application</title>

</head>

<body>

<p>Name: ${name}</p>

</body>

</html>

Однако может сложиться ситуация, что сразу в нескольких контекстах одновременно содержатся данные с одним и тем же именем, например, "name". Соответственно EL будет получать данные в порядке просмотра контекстов. Но, возможно, нам захочется выводить данные из какого-то определенного контекста. В этом случае перед названием данных мы можем указать название контекста: pageScope, requestScope, sessionScope или applicationScope. Например:

<%

pageContext.setAttribute("name", "Bob");

%>

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>JSP Application</title>

</head>

<body>

<p>Name: ${requestScope.name}</p>

</body>

</html>

Передача сложных объектов

Подобным образом мы можем передать и более сложные данные - списки, массивы, сложные объекты. Допустим, в сервлете на страницу передается массив:

@WebServlet("/hello")

public class HelloServlet extends HttpServlet {

protected void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

String[] users = new String[] {"Tom", "Bob", "Sam"};

request.setAttribute("users", users);

getServletContext().getRequestDispatcher("/basic.jsp").forward(request, response);

}

}

И на странице basic.jsp получаем переданные данные:

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8">

<title>JSP Application</title>

</head>

<body>

<p>Second: ${users[1]}</p>

<p>Third: ${users[2]}</p>

</body>

</html>

Вместо массива в сервлете мы могли бы передать более гибкий объект - ArrayList:

ArrayList<String> users = new ArrayList<String>();

Collections.addAll(users, "Tom", "Bob", "Sam");

request.setAttribute("users", users);

В этом случае в jsp мы могли бы получить эти данные так же, как и из массива.

## Встроенные объекты Expression Language

По умолчанию Expression Language предоставляет ряд встроенных объектов, которые позволяют использовать различные аспекты запроса:

**param**: объект, который хранит все переданные странице параметры

**paramValues**: хранит массив значений для определенного параметра (если для параметра передается сразу несколько значений)

**header**: хранит все заголовки запроса

**headerValues**: предоставляет массив значений для определенного заголовка запроса

**cookie**: предоставляет доступ к отправленным в запросе кукам

**initParam**: возвращает значение для определенного параметра из элемента context-param из файла web.xml

**pageContext**: предоставляет доступ к объекту PageContext, который представляет контекст текущей страницы jsp

## JSP Standard Tag Library

JSP по умолчанию позволяет встраивать код на java в разметку html. Однако иногда использование стандартных способов для ряда операций, например, для вывод на страницу элементов из списка и т.д., может быть несколько громоздким. Чтобы упростить встраивание кода java в JSP была разработана специальная библиотека - **JSTL**. JSTL (JSP Standard Tag Library) предоставляет теги для базовых задач JSP (цикл, условные выражения и т.д.)

Эта библиотека не является частью инфраструктуры Java EE, поэтому ее необходимо добавлять в проект самостоятельно. Библиотеку можно найти по адресу <http://central.maven.org/maven2/javax/servlet/jstl/1.2/>. Из репозитория нам необходимо загрузить файл **jstl-1.2.jar**.

Несмотря на то, что JSTL часто называется библиотекой, на самом деле она содержит ряд библиотек:

**Core**: содержит основные теги для наиболее распространенных задач.

Использует префикс "c" и uri "http://java.sun.com/jsp/jstl/core"

**Formatting**: предоставляет теги для форматирования чисел, дат, времени.

Использует префикс "fmt" и uri "http://java.sun.com/jsp/jstl/fmt"

**SQL**: предоставляет теги для sql-запросами и источниками данных.

Использует префикс "sql" и uri "http://java.sun.com/jsp/jstl/sql"

**XML**: предоставляет теги для работы с xml.

Использует префикс "x" и uri "http://java.sun.com/jsp/jstl/xml"

**Functions**: предоставляет функции для работы со строками.

Использует префикс "fn" и uri "http://java.sun.com/jsp/jstl/functions"

Для подключения функционала этих библиотек на страницу jsp применяется директива **taglib**. Например, подключения основной библиотеки:

<%@ taglib uri="<http://java.sun.com/jsp/jstl/core>" prefix="c" %>

Рассмотрим некоторые основные возможности JSTL.

Цикл

Вывод в цикле элементов массива или коллекции:

<c:forEach var="user" items="${users}">

<p>${user}</p>

</c:forEach>

В данном случае параметр "items" указывает на коллекцию, элементы которой выводятся. А параметр "var" задает переменную, через которую доступен текущий перебираемый элемент.

Например, в сервлет передается массив строк:

public class HelloServlet extends HttpServlet {

protected void doGet(HttpServletRequest request, HttpServletResponse response)

throws ServletException, IOException {

String[] users = new String[]{"Tom", "Bob", "Sam"};

request.setAttribute("users", users);

getServletContext().getRequestDispatcher("/index.jsp").forward(request, response);

}

}

В jsp мы можем получить и вывести элементы массива следующим образом:

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title>User Info</title>

</head>

<body>

<ul>

<c:forEach var="user" items="${users}">

<li><c:out value="${user}" /></li>

</c:forEach>

</ul>

</body>

</html>

Цикл в JSTL в JSP

Еще один пример - вывод в цикле всех cookie:

<ul>

<c:forEach var="cook" items="${cookie}">

<li>

<p><c:out value="${cook.value.name}" /></p>

<p><c:out value="${cook.value.value}" /></p>

</li>

</c:forEach>

</ul>

Условные выражения

Выражение if:

<c:if test="${isVisible == true}">

<p>Visible</p>

</c:if>

В данном случае если атрибут isVisible равен true, то выводится код, который расположен между тегами <c:if> и </c:if>

Если надо задать альтернативную логику, то можно добавить тег c:if, который проверяет противоположное условие:

<c:if test="${isVisible == true}">

<p>Visible</p>

</c:if>

<c:if test="${isVisible == false}">

<p>Invisible</p>

</c:if>

Тег choose

Тег c:choose подобно конструкции switch...case в Java проверяет объект на соответствие одному из значений. Например

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title>JSTL in JSP</title>

</head>

<body>

<c:choose>

<c:when test="${val == 1}">

<p>Equals to 1</p>

</c:when>

<c:when test="${val == 2}">

<p>Equals to 2</p>

</c:when>

<c:otherwise>

<p>Undefined</p>

</c:otherwise>

</c:choose>

</body>

</html>

В данном случае тег c:choose проверяет значение атрибута "val". Для проверки применяются вложенные теги c:when, которые аналогичны блокам case в конструкции switch..case. С помощью их параметра test значение атрибута сравнивается с определенным значением. И если выражения сравнения истинно, то выводится код, который размещен внутри данного элемента c:when. Таким образом мы можем определить несколько блоков c:when. Дополнительный тег <c:otherwise> выполняется, если условия проверки значения во всех тегах c:when ложно.

Тег url

Тег <c:url> позволяет создать адрес относительно корня приложения. Этот тег может применяться, например, при создании ссылок.

Например,

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<!DOCTYPE html>

<html>

<head>

<meta charset="UTF-8" />

<title>JSTL in JSP</title>

</head>

<body>

<a href='<c:url value="/edit" />'><c:url value="/edit" /></a>

</body>

</html>

Параметр value содержит часть адреса, которая добавляется к корню приложения.

Редирект

С помощью тега redirect можно установить редирект на другой адрес. Например, в случае если атрибут val не определен, то делаем редирект на страницу "notfound.jsp":

<c:if test="${val == null}">

<c:redirect url="/notfound.jsp" />

</c:if>