ГЛАВА 23 Обработка событий

Настоящая глава посвящена важнейшему аспекту Java — событиям. Обработка событий является фундаментальной для всего программирования Java, поскольку это — основа аплетов и прочих типов программ с *графическим пользовательским интерфейсом* (Graphical User Interface — GUI). Как упоминалось в главе 22, аплеты представляют собой управляемые событиями программы, использующие графический интерфейс для взаимодействия с пользователем. Более того, любая программа с графическим пользовательским интерфейсом, такая как приложение Java, написанное для Windows, является управляемой событиями. Другими словами, вы не можете написать такую программу без четкого представления об обработке событий. События поддерживаются множеством пакетов, включая `java.util`, `java.awt` и `java.event`.

Большинство событий, на которые будет реагировать ваша программа, происходят при взаимодействии пользователя с программой на основе GUI. Именно события такого рода и рассматриваются в настоящей главе. События попадают в вашу программу множеством различных путей, каждый из которых зависит от конкретного события. Существует несколько типов событий, включая создаваемые мышью, клавиатурой и различными элементами управления GUI, такими как кнопка, полоса прокрутки или флажок.

Мы начнем эту главу с обзора механизма управления событиями Java. Затем рассмотрим основные классы и интерфейсы событий, используемые библиотекой AWT, и разработаем несколько примеров, демонстрирующих основы обработки событий. Эта глава также расскажет о том, как использовать классы адаптеров, вложенные классы и анонимные вложенные классы, чтобы упростить код обработки событий. Примеры, приведенные в остальной части книги, будут использовать эти приемы.

На заметку! Материал этой главы основан на событиях, имеющих отношение к программам на основе GUI. Но иногда события также используются в целях, не имеющих прямого отношения к программам подобного рода. Тем не менее во всех случаях применяются одни и те же приемы обработки событий.

Два механизма обработки событий

Прежде чем приступить к обсуждению обработки событий, сделаем одно важное замечание. Способ обработки событий существенно изменился от исходной версии Java (1.0) до более современных версий, начиная с 1.1. Метод обработки событий, принятый в версии 1.0, все еще поддерживается, хотя и не рекомендован для применения в новых программах. К тому же многие методы, поддерживающие старую модель событий 1.0, теперь объявлены устаревшими (deprecated). Современный подход состоит в том, что события должны обрабатываться всеми программами так, как описано в этой книге.

Java 7.indb 717 14.02.2012 13:01:59

Модель делегирования событий

Современный подход к обработке событий основан на *модели* делегирования *событий* (delegation event model), определяющей стандартные и согласованные механизмы для создания и обработки событий. Его концепция проста: *источник* извещает о событии одного или несколько *слушателей* (listener). В этой схеме слушатель просто ожидает до тех пор, пока не получит извещение о событии. Как только извещение о событии получено, слушатель обрабатывает его и возвращает управление. Преимущество такого дизайна в том, что логика приложения, обрабатывающего события, четко отделена от логики пользовательского интерфейса, извещающего об этом событии. Элемент пользовательского интерфейса может "делегировать" обработку события отдельному фрагменту кода.

В модели делегирования событий слушатели должны регистрироваться источником для того, чтобы получать извещения о событиях. Это обеспечивает важное преимущество: уведомления посылаются только тем слушателям, которые желают получать их. Это более эффективный способ обработки событий, нежели тот, что был принят в старом подходе Java 1.0. Раньше извещение о событии распространялось по всей иерархии вложенности до тех пор, пока не было обработано компонентом. Это вынуждало все компоненты получать извещения о событии, которые они могли и не обрабатывать, что приводило к значительным затратам времени. Модель делегирования событий исключила подобную расточительность.

В последующих разделах определим события и опишем роли источников и слушателей.

На заметку! Java также позволяет обрабатывать события без применения модели делегирования. Это может быть сделано при помощи расширения компонента библиотеки AWT. Такая техника обсуждается в конце главы 25. Однако модель делегирования событий более предпочтительна по описанным выше причинам.

События

В модели делегирования *событие* — это объект, описывающий изменение состояния источника. Он может быть создан в результате взаимодействия пользователя с элементом графического интерфейса. К событиям приводят такие действия, как щелчок на экранной кнопке, ввод символа с клавиатуры, выбор элемента в списке и щелчок кнопкой мыши. Многие другие пользовательские операции также могут служить подобными примерами.

События могут также происходить и не в результате прямого взаимодействия с пользовательским интерфейсом. Например, событие может произойти по истечении времени таймера в результате превышения счетчиком некоторого значения, программного или аппаратного сбоя либо завершения некоторой операции. Вы можете определять собственные события, отвечающие специфике вашего приложения.

Источники событий

Источник (source) — это объект, извещающий о событии. Событие происходит при изменении внутреннего состояния объекта некоторым образом. Источники могут извещать о событиях нескольких типов.

Источник должен регистрировать слушателей, чтобы они получали извещение о событиях определенного рода. Каждый тип события имеет собственный метод регистрации. Вот его общая форма.

```
public void addТипListener(ТипListener el)
```

Здесь *Тип* — имя события, а *el* — ссылка на слушателя событий. Например, метод, регистрирующий слушателя событий клавиатуры, называется addKeyListener(), а метод, регистрирующий слушателя движения мыши, — addMouseMotionListener(). Когда событие происходит, все зарегистрированные слушатели получают копию объекта события. Это называется *групповой рассылкой* (multicasting) события. Во всех случаях уведомления отправляются только тем слушателям, которые зарегистрированы на их получение.

Некоторые источники допускают регистрацию только одного слушателя. Общая форма такого метода показана ниже.

```
public void addТипListener(ТипListener el)
    throws java.util.TooManyListenersException
```

Здесь *Тип* — это имя объекта события, а *el* — ссылка на слушателя события. Когда такое событие происходит, зарегистрированный слушатель получает уведомление. Это называется *индивидуальной рассылкой* (unicasting) события.

Источник должен также предоставлять метод, позволяющий слушателю отменить регистрацию для определенного типа событий. Общая форма этого метода такова.

```
public void removeТипListener(ТипListener el)
```

И снова *Тип* — это имя объекта события, а *el* — ссылка на слушателя события. Например, чтобы удалить слушателя клавиатуры, следует вызвать метод removeKeyListener().

Метод, который добавляет или удаляет слушателей, предоставляется источником, извещающим о событии. Например, класс Component предлагает методы для добавления и удаления слушателей клавиатуры и мыши.

Слушатели событий

Слушатель (listener) — это объект, уведомляемый о событии. К нему предъявляются два основных требования. Во-первых, он должен быть зарегистрирован одним или более источниками событий, чтобы получать уведомления о событиях определенного рода. Во-вторых, он должен реализовать методы для получения и обработки таких уведомлений.

Методы, принимающие и обрабатывающие события, определены в наборе интерфейсов, находящихся в пакете java.awt.event. Например, интерфейс MouseMotionListener определяет два метода для получения уведомлений о перетаскивании объекта или перемещении мыши.

Любой объект может принимать и обрабатывать одно или оба эти события, если предоставляет реализацию этого интерфейса. В этой и последующих главах мы еще поговорим о многих других интерфейсах слушателей.

Классы событий

Классы, представляющие события, находятся в ядре механизма обработки событий Java. А потому дискуссия об обработке событий должна начинаться с классов событий. Важно понимать, однако, что Java определяет несколько типов событий и что не все классы событий будут упомянуты в настоящей главе. Наиболее широко используемыми событиями являются те, что определены библиотеками AWT и Swing. Здесь сосредоточимся на событиях библиотеки AWT. (Многие из них относятся также и к библиотеке Swing.) Несколько специфич-

ных для библиотеки Swing событий будут описаны в главе 30, когда речь пойдет о библиотеке Swing.

В корне иерархии классов событий Java лежит класс `EventObject`, расположенный в пакете `java.util`. Это — суперкласс для всех событий. Его единственный конструктор выглядит следующим образом.

```
EventObject(Object источник)
```

Параметр *источник* здесь представляет объект, извещающий о событии.

Класс `EventObject` содержит два метода: `getSource()` и `toString()`. Метод `getSource()` возвращает источник события. Его общая форма такова.

```
Object getSource()
```

Как можно ожидать, метод `toString()` возвращает строку — эквивалент события.

Класс `AWTEvent`, определенный в пакете `java.awt`, рассматривается в конце главы 25. А пока важно знать, что все классы, о которых пойдет речь в этом разделе, являются подклассами класса `AWTEvent`. Итак, подытожим.

- Класс `EventObject` — суперкласс для всех событий.
- Класс `AWTEvent` — суперкласс всех событий библиотеки AWT, обрабатываемых моделью делегирования событий.

Пакет `java.awt.event` определяет множество типов событий, происходящих во многих элементах пользовательского интерфейса. В табл. 23.1 показано несколько широко используемых классов событий и представлено краткое описание того, когда они происходят. Наиболее часто используемые конструкторы и методы каждого класса описаны в последующих разделах.

Таблица 23.1. Основные классы событий в пакете `java.awt.event`

Класс события	Описание
`ActionEvent`	Происходит при нажатии кнопки, двойном щелчке на элементе списка либо выборе пункта меню
`AdjustmentEvent`	Происходит при манипуляциях с полосой прокрутки
`ComponentEvent`	Происходит при сокрытии компонента, его перемещении, изменении его размера либо включении видимости
`ContainerEvent`	Происходит при добавлении или исключении компонента контейнера
`FocusEvent`	Происходит, когда компонент получает или теряет фокус клавиатурного ввода
`InputEvent`	Абстрактный суперкласс для всех классов ввода компонентов
`ItemEvent`	Происходит при щелчке на флажке или элементе списка, а также при выборе элемента списка и выборе или отмене выбора пункта меню
`KeyEvent`	Происходит при получении ввода с клавиатуры
`MouseEvent`	Происходит при перетаскивании, перемещении, щелчках, нажатии и отпускании кнопок мыши; также происходит, когда курсор мыши наводится на компонент либо покидает его
`MouseWheelEvent`	Происходит при прокрутке колесика мыши
`TextEvent`	Происходит при изменении значения текстовой области или текстового поля
`WindowEvent`	Происходит при активизации либо закрытии окна, а также при деактивизации, свертывании, развертывании окна или выходе из него

Класс ActionEvent

Объект класса ActionEvent создается при нажатии кнопки, двойном щелчке на элементе списка либо при выборе пункта меню. Класс ActionEvent определяет четыре целочисленные константы, которые могут быть использованы для идентификации любых модификаторов, ассоциированных с событием действия, — ALT_MASK, CTRL_MASK, META_MASK и SHIFT_MASK. Кроме того, имеется целочисленная константа ACTION_PERFORMED, которая может служить для идентификации событий действия.

Класс ActionEvent имеет три конструктора.

```
ActionEvent(Object источник, int тип, String команда)
ActionEvent(Object источник, int тип, String команда, int модификаторы)
ActionEvent(Object источник, int тип, String команда, long когда, int
модификаторы)
```

Здесь *источник* — это ссылка на объект, извещающий о событии. Тип события указан параметром *тип*, а его командная строка — параметром *команда*. Аргумент *модификаторы* указывает на нажатие модифицирующих клавиш (<Alt>, <Ctrl>, <Meta> и/или <Shift>) в момент события.

Вы можете получить имя команды для вызывающего объекта класса ActionEvent, используя метод getActionCommand(), показанный ниже.

```
String getActionCommand()
```

Например, при щелчке на кнопке происходит событие действия, которое имеет имя команды, соответствующее метке экранной кнопки.

Метод getModifiers() возвращает значение, указывающее на то, какие модифицирующие клавиши (<Alt>, <Ctrl>, <Meta> и/или <Shift>) были нажаты в момент события. Его форма такова.

```
int getModifiers()
```

Метод getWhen() возвращает время, когда произошло событие. Это называется *временной меткой* события. Метод getWhen() выглядит следующим образом.

```
long getWhen()
```

Класс AdjustmentEvent

Событие класса AdjustmentEvent передается полосой прокрутки. Существует пять типов событий настройки (adjustment). Класс AdjustmentEvent определяет целочисленные константы, которые могут использоваться для идентификации. Константы и их описания представлены в табл. 23.2.

Таблица 23.2. Константы, определенные классом AdjustmentEvent

Константа	Описание
BLOCK_DECREMENT	Пользователь щелкнул внутри полосы прокрутки для уменьшения ее значения
BLOCK_INCREMENT	Пользователь щелкнул внутри полосы прокрутки для увеличения ее значения
TRACK	Передвинут бегунок полосы
UNIT_DECREMENT	Был выполнен щелчок на кнопке в конце полосы для уменьшения ее значения
UNIT_INCREMENT	Был выполнен щелчок на кнопке в конце полосы для увеличения ее значения

Кроме того, имеется также целочисленная константа ADJUSTMENT_VALUE_CHANGED, которая указывает на то, что произошло изменение.

Вот единственный конструктор класса AdjustmentEvent.

```
AdjustmentEvent(Adjustable источник, int идентификатор, int тип, int
данные)
```

Здесь *источник* — ссылка на объект, извещающий о событии. Параметр *идентификатор* определяет событие. Тип настройки указан параметром *тип*, а ассоциированные с ней данные — параметром *данные*.

Метод getAdjustable() возвращает объект, извещающий о событии. Его форма представлена ниже.

```
Adjustable getAdjustable()
```

Тип события настройки может быть получен методом getAdjustmentType(). Он возвращает одну из констант, определенных классом AdjustmentEvent. Его общая форма такова.

```
int getAdjustmentType()
```

Величина настройки может быть получена методом getValue(), показанным ниже.

```
int getValue()
```

Например, когда выполняются манипуляции с полосой прокрутки, этот метод возвращает значение, представленное изменением.

Класс ComponentEvent

Объект класса ComponentEvent создается при изменении размера, положения или видимости компонента. Существует четыре типа событий компонентов. Для их идентификации класс ComponentEvent определяет целочисленные константы. Константы и их описания представлены в табл. 23.3.

Таблица 23.3. Константы, определенные классом ComponentEvent

Константа	Описание
COMPONENT_HIDDEN	Компонент скрыт
COMPONENT_MOVED	Компонент перемещен
COMPONENT_RESIZED	Изменен размер компонента
COMPONENT_SHOWN	Компонент стал видимым

Класс ComponentEvent имеет следующий конструктор.

```
ComponentEvent(Component источник, int тип)
```

Здесь *источник* — ссылка на объект, извещающий о событии. Тип события указан параметром *тип*.

Класс ComponentEvent — это прямой или непрямой суперкласс для классов ContainerEvent, FocusEvent, KeyEvent, MouseEvent, WindowEvent и др.

Метод getComponent() возвращает компонент, извещающий о событии. Выглядит он так.

```
Component getComponent()
```

Класс ContainerEvent

Объект класса ContainerEvent создается при добавлении компонента в контейнер или его удалении оттуда. Существует два типа событий контейнера. Класс

ContainerEvent определяет целочисленные константы, которые могут использоваться для его идентификации, — COMPONENT_ADDED и COMPONENT_REMOVED. Они определяют, добавлен ли компонент к контейнеру или же исключен из него.

Класс ContainerEvent — это подкласс класса ComponentEvent, имеющий следующий конструктор.

```
ContainerEvent(Component источник, int тип, Component компаратор)
```

Здесь *источник* — ссылка на контейнер, который известил о событии. Тип события указан параметром *тип*, а компонент, добавляемый в контейнер либо удаляемый из него, — параметром *компаратор*.

Вы можете получить ссылку на контейнер, создавший это событие, из метода getContainer(), показанного ниже.

```
Container getContainer()
```

Метод getChild() возвращает ссылку на компонент, который был добавлен или удален из контейнера. Его общая форма такова.

```
Component getChild()
```

Класс FocusEvent

Объект класса FocusEvent создается при получении или потере компонентом фокуса. Эти события определяются целочисленными константами FOCUS_GAINED и FOCUS_LOST.

Класс FocusEvent — это подкласс класса ComponentEvent, имеющий следующие конструкторы.

```
FocusEvent(Component источник, int тип)
FocusEvent(Component источник, int тип, boolean временныйФлаг)
FocusEvent(Component источник, int тип, boolean временныйФлаг, Component
другое)
```

Здесь *источник* — это ссылка на компонент, извещающий о событии. Тип события указан параметром *тип*. Аргумент *временныйФлаг* установлен в состояние true, если событие фокуса является временным. В противном случае он устанавливается в состояние false. (Событие временного фокуса происходит в результате другой операции пользовательского интерфейса. Например, предположим, что фокус находится на текстовом поле. Если пользователь перемещает мышь, чтобы подвинуть полосу прокрутки, то фокус временно теряется.)

Другой компонент, участвующий в изменении фокуса и называемый противоположным компонентом, передается параметром *другое*. Таким образом, если происходит событие FOCUS_GAINED, то параметр *другое* будет ссылаться на компонент, теряющий фокус. В противоположность этому, если происходит событие FOCUS_LOST, то параметр *другое* ссылается на компонент, получающий фокус.

Вы можете определить другие компоненты вызовом метода getOppositeComponent(), показанного ниже.

```
Component getOppositeComponent()
```

Метод возвращает противоположный компонент.

Метод isTemporary() указывает на то, является ли изменение фокуса временным. Его форма такова.

```
boolean isTemporary()
```

Метод возвращает значение true, если изменение временно. В противном случае он возвращает значение false.

Класс InputEvent

Абстрактный класс InputEvent является подклассом класса ComponentEvent и суперклассом для событий ввода компонента. Его подклассы — KeyEvent и MouseEvent.

Класс InputEvent определяет несколько строковых констант, представляющих любые модификаторы, такие как признак нажатия управляющих клавиш, которые могут быть ассоциированы с событием. Изначально класс InputEvent определяет следующие восемь значений для представления модификаторов.

ALT_MASK	BUTTON2_MASK	META_MASK
ALT_GRAPH_MASK	BUTTON3_MASK	SHIFT_MASK
BUTTON1_MASK	CTRL_MASK	

Однако поскольку существует возможность конфликтов в используемых модификаторах между клавиатурными событиями, событиями мыши, а также другие проблемы, были добавлены следующие расширенные значения модификаторов.

ALT_DOWN_MASK	BUTTON2_DOWN_MASK	META_DOWN_MASK
ALT_GRAPH_DOWN_MASK	BUTTON3_DOWN_MASK	SHIFT_DOWN_MASK
BUTTON1_DOWN_MASK	CTRL_DOWN_MASK	

При написании нового кода рекомендуется использовать новые расширенные модификаторы вместо исходных.

Чтобы проверить, какая клавиша модификатора была нажата в момент события, используйте методы isAltDown(), isAltGraphDown(), isControlDown(), isMetaDown() и isShiftDown(). Формы этих методов показаны ниже.

```
boolean isAltDown()
boolean isAltGraphDown()
boolean isControlDown()
boolean isMetaDown()
boolean isShiftDown()
```

Вы можете получить значение, содержащее все флаги исходных модификаторов, вызовом метода getModifiers(). Он выглядит так.

```
int getModifiers()
```

Расширенные модификаторы можно получить методом getModifiersEx(), показанным ниже.

```
int getModifiersEx()
```

Класс ItemEvent

Объект класса ItemEvent создается при щелчке на флажке или элементе списка либо при выборе отмечаемого пункта меню. (Флажки и окна списков будут описаны далее.) Существует два типа событий элементов, которые идентифицируются целочисленными константами, описанными в табл. 23.4.

Таблица 23.4. Константы, определенные классом ItemEvent

Константа	Описание
DESELECTED	Пользователь отменил выбор элемента
SELECTED	Пользователь выбрал элемент

Кроме того, класс ItemEvent определяет одну целочисленную константу ITEM_STATE_CHANGED, которая сигнализирует об изменении состояния.

Класс ItemEvent имеет следующий конструктор.

```
ItemEvent(ItemSelectable источник, int тип, Object элемент, int состояние)
```

Здесь *источник* — ссылка на компонент, известивший о событии. Например, это может быть список или элемент выбора. Тип события указывается параметром *тип*. Конкретный элемент, приведший к событию, передается в параметре *элемент*. Текущее состояние элемента содержится в параметре *состояние*.

Метод getItem() может быть использован для получения ссылки на элемент, приведший к событию. Его сигнатура выглядит так.

```
Object getItem()
```

Метод getItemSelectable() может быть использован для получения ссылки на объект ItemSelectable, известивший о событии. Его общая форма показана ниже.

```
ItemSelectable getItemSelectable()
```

Списки и флажки являются примерами элементов пользовательского интерфейса, реализующих интерфейс ItemSelectable.

Метод getStateChanged() возвращает измененное состояние (т.е. значение SELECTED или DESELECTED) для события. Выглядит он так.

```
int getStateChanged()
```

Класс KeyEvent

Объект класса KeyEvent создается при клавиатурном вводе. Существует три типа клавиатурных событий, идентифицируемых целочисленными константами, — KEY_PRESSED, KEY_RELEASED и KEY_TYPED. События первых двух типов происходят при нажатии и отпускании клавиши. Последний же тип происходит при вводе символа. Помните, что нажатие не всех клавиш приводит к вводу символа. Например, нажатие клавиши <Shift> не вводит никакого символа.

Класс KeyEvent определяет множество других целочисленных констант. Например, константы VK_0–VK_9 и VK_A–VK_Z определяют эквиваленты ASCII чисел и букв. А вот еще несколько других констант.

VK_ALT	VK_DOWN	VK_LEFT	VK_RIGHT
VK_CANCEL	VK_ENTER	VK_PAGE_DOWN	VK_SHIFT
VK_CONTROL	VK_ESCAPE	VK_PAGE_UP	VK_UP

Константы VK задают *коды виртуальных клавиш*, не зависящие от любых модификаторов вроде <Control>, <Alt> или <Shift>.

Класс KeyEvent является подклассом класса InputEvent. Вот один из его конструкторов.

```
KeyEvent(Component источник, int тип, long когда, int модификаторы, int
код, char символ)
```

Здесь *источник* — ссылка на компонент, извещающий о событии. Тип события указывается параметром *тип*. Системное время, когда была нажата клавиша, передается в параметре *когда*. Аргумент *модификаторы* указывает, какие модификаторы были нажаты в момент события клавиши. Код виртуальной клавиши, такой как VK_UP, VK_A и так далее, передается в параметре *код*. Символьный эквивалент (если есть) передается в параметре *символ*. Если никакого корректного символа событие не содержит, то *символ* содержит значение CHAR_UNDEFINED. Для событий KEY_TYPED параметр *код* будет содержать значение VK_UNDEFINED.

Класс `KeyEvent` определяет несколько методов, но, вероятно, наиболее часто используемые из них — это метод `getKeyChar()`, возвращающий символ клавиши, и метод `getKeyCode()`, возвращающий код клавиши. Их общая форма представлена ниже.

```
char getKeyChar()
int getKeyCode()
```

Если никаких корректных символов нажатие клавиши не создает, то метод `getKeyChar()` возвращает значение `CHAR_UNDEFINED`. При событии `KEY_TYPED` метод `getKeyCode()` возвращает значение `VK_UNDEFINED`.

Класс MouseEvent

Существует восемь типов событий мыши. Класс `MouseEvent` определяет для их идентификации набор целочисленных констант, которые описаны в табл. 23.5.

Таблица 23.5. Константы, определенные классом MouseEvent

Константа	Описание
MOUSE_CLICKED	Пользователь щелкнул кнопкой мыши
MOUSE_DRAGGED	Пользователь перетащил мышь при нажатой кнопке
MOUSE_ENTERED	Курсор мыши вошел в компонент
MOUSE_EXITED	Курсор мыши покинул компонент
MOUSE_MOVED	Мышь перемещена
MOUSE_PRESSED	Кнопка мыши нажата
MOUSE_RELEASED	Кнопка мыши отпущена
MOUSE_WHEEL	Выполнена прокрутка колесика мыши

Класс `MouseEvent` — это подкласс класса `InputEvent`. Вот один из его конструкторов.

```
MouseEvent(Component источник, int тип, long когда, int модификаторы,
int x, int y, int щелчки, boolean вызовМеню)
```

Здесь *источник* — ссылка на компонент, извещающий о событии. Тип события указан параметром *тип*. Системное время, когда произошло событие, передается в параметре *когда*. Аргумент *модификаторы* указывает, какие клавиши модификаторов были нажаты в момент события мыши. Координаты курсора мыши передаются параметрами *x* и *y*. Счетчик щелчков передается в параметре *щелчки*. Флаг *вызовМеню* указывает, должно ли данное событие вызывать всплывающее меню на данной платформе.

Два часто используемых метода этого класса — это методы `getX()` и `getY()`. Они возвращают координаты X и Y курсора мыши на момент события. Их форма такова.

```
int getX()
int getY()
```

В качестве альтернативы для получения координат курсора мыши вы можете использовать метод `getPoint()`. Он показан ниже.

```
Point getPoint()
```

Этот метод возвращает объект класса `Point`, содержащий координаты X, Y в своих целочисленных членах `x` и `y`.

Метод `translatePoint()` изменяет местоположение события. Его форма показана ниже.

```
void translatePoint(int x, int y)
```

Здесь аргументы *x* и *y* добавляются к первоначальным координатам события.

Метод `getClickCount()` получает количество щелчков мыши для данного события. Его сигнатура показана ниже.

```
int getClickCount()
```

Метод `isPopupTrigger()` проверяет, вызывает ли данное событие всплывающее меню на текущей платформе. Его форма такова.

```
boolean isPopupTrigger()
```

Также доступен метод `getButton()`, показанный ниже.

```
int getButton()
```

Он возвращает значение, представляющее кнопку, вызвавшую событие. Возвращаемое значение будет одной из констант, определенных в классе `MouseEvent`.

NOBUTTON	BUTTON1	BUTTON2	BUTTON3

Значение `NOBUTTON` указывает на то, что никакая кнопка не была нажата или отпущена. Доступны также три метода класса `MouseEvent`, которые получают координаты мыши относительно экрана, а не компонента. Они показаны ниже.

```
Point getLocationOnScreen()
int getXOnScreen()
int getYOnScreen()
```

Метод `getLocationOnScreen()` возвращает объект класса `Point`, содержащий обе координаты — X и Y. Другие два метода возвращают по одной координате соответственно.

Класс `MouseWheelEvent`

Этот класс инкапсулирует событие колесика мыши. Он является подклассом класса `MouseEvent`. Не все мыши оснащены колесиками. Если у мыши есть колесико, оно располагается между левой и правой кнопками. Эти колесики используются для прокрутки (изображения, текста, таблиц и т.п.). Класс `MouseWheelEvent` определяет две целочисленные константы, описанные в табл. 23.6.

Таблица 23.6. Константы, определенные классом `MouseWheelEvent`

Константа	Описание
WHEEL_BLOCK_SCROLL	Произошло событие прокрутки на страницу вверх или вниз
WHEEL_UNIT_SCROLL	Произошло событие прокрутки на строку вверх или вниз

Вот один из конструкторов, определенных в классе `MouseWheelEvent`.

```
MouseWheelEvent(Component источник, int тип, long когда,
                int модификаторы, int x, int y, int щелчки,
                boolean вызовМеню, int какПрокрутить,
                int сколько, int количество)
```

Здесь *источник* — ссылка на компонент, извещающий о событии. Тип события указан параметром *тип*. Системное время события передается в параметре *когда*. Аргумент *модификаторы* указывает, какие клавиши модификаторов были нажаты в момент события мыши. Координаты курсора мыши передаются параметрами *x* и *y*.

Счетчик щелчков передается в параметре *щелчки*. Флаг *вызовМеню* указывает на то, должно ли данное событие вызывать всплывающее меню на текущей платформе. Значением параметра *какПрокрутить* должно быть либо WHEEL_UNIT_SCROLL, либо WHEEL_BLOCK_SCROLL. Количество единиц прокрутки передается в параметре *сколько*. Параметр *количество* указывает количество единиц прокрутки, на которое повернуто колесико.

Класс MouseWheelEvent определяет методы, предоставляющие доступ к событию колесика. Чтобы получить количество единиц прокрутки, вызовите метод getWheelRotation(), показанный ниже.

```
int getWheelRotation()
```

Он возвращает количество единиц прокрутки. Если значение положительно, значит, колесико повернуто против часовой стрелки, если же отрицательно — то по часовой стрелке. В комплекте JDK 7 представлен метод getPreciseWheel Rotation(), поддерживающий колесико с высокой разрешающей способностью. Он работает, как и метод getWheelRotation(), но возвращает значение типа double.

Чтобы получить тип прокрутки, вызовите метод getScrollType().

```
int getScrollType()
```

Он возвращает либо значение WHEEL_UNIT_SCROLL, либо значение WHEEL_BLOCK_SCROLL. Если типом прокрутки является WHEEL_UNIT_SCROLL, то вы получите количество единиц прокрутки непосредственно методом getScroll Amount(). Этот метод выглядит следующим образом.

```
int getScrollAmount()
```

Класс TextEvent

Экземпляры этого класса описывают текстовые события. Они создаются текстовыми полями и областями, когда в них осуществляется ввод пользователем или программой. Класс TextEvent определяет целочисленную константу TEXT_VALUE_CHANGED.

Единственный конструктор этого класса показан ниже.

```
TextEvent(Object источник, int тип)
```

Здесь *источник* — ссылка на объект, извещающий о событии. Тип события указывается параметром *тип*.

Объект класса TextEvent не включает символов, находящихся в данный момент в текстовом компоненте, известившем о событии. Вместо этого ваша программа должна использовать другие методы, ассоциированные с текстовым компонентом, для извлечения информации. Эта операция отличается от других объектов событий, о которых речь пойдет в следующем разделе. По этой причине никакие методы класса TextEvent мы пока не обсуждаем. Уведомления о текстовом событии следует считать сигналом слушателю о том, что последний должен извлечь информацию из указанного текстового компонента.

Класс WindowEvent

Существует десять типов событий окон. Класс WindowEvent определяет целочисленные константы, которые могут использоваться для их идентификации. Эти константы вместе с их описаниями перечислены в табл. 23.7.

Таблица 23.7. Константы, определенные классом WindowEvent

Константа	Описание
WINDOW_ACTIVATED	Окно было активизировано
WINDOW_CLOSED	Окно было закрыто
WINDOW_CLOSING	Пользователь запросил закрытие окна
WINDOW_DEACTIVATED	Окно деактивизировано
WINDOW_DEICONIFIED	Окно развернуто
WINDOW_GAINED_FOCUS	Окно получило фокус ввода
WINDOW_ICONIFIED	Окно свернуто
WINDOW_LOST_FOCUS	Окно утратило фокус ввода
WINDOW_OPENED	Окно было открыто
WINDOW_STATE_CHANGED	Состояние окна изменилось

Класс `WindowEvent` — это подкласс класса `ComponentEvent`. Он определяет несколько конструкторов. Рассмотрим первый.

```
WindowEvent(Window источник, int тип)
```

Здесь *источник* — ссылка на объект, известивший о событии. Тип события передается в параметре *тип*. Следующие три конструктора обеспечивают более подробный контроль.

```
WindowEvent(Window источник, int тип, Window другое)
WindowEvent(Window источник, int тип, int изСостояния, int вСостояние)
WindowEvent(Window источник, int тип, Window другое, int изСостояния,
int вСостояние)
```

Здесь параметр *другое* определяет противоположное окно при событии фокуса или активизации. Параметр *изСостояния* определяет предыдущее состояние окна, а параметр *вСостояние* — новое состояние, которое окно получает при смене состояния.

Часто используемый метод этого класса — `getWindow()`. Он возвращает объект класса `Window`, известивший о событии. Вот его общая форма.

```
Window getWindow()
```

Класс `WindowEvent` также определяет методы, возвращающие противоположное окно (при событиях фокуса или активизации), предыдущее состояние окна, а также его текущее состояние. Эти методы показаны ниже.

```
Window getOppositeWindow()
int getOldState()
int getNewState()
```

Источники событий

В табл. 23.8 перечислены некоторые компоненты пользовательского интерфейса, способные извещать о событиях, описанных в предыдущем разделе. В дополнение к элементам графического пользовательского интерфейса, любой класс, унаследованный от класса `Component`, такой как класс `Applet`, способен извещать о событиях. Например, вы можете получать события клавиатуры и мыши от аплета. (Также вы можете создавать собственные компоненты, которые извещают о событиях.) В этой главе поговорим только о событиях клавиатуры и мыши, а в следующих главах речь пойдет об обработке событий от источников, перечисленных в табл. 23.8.

Таблица 23.8. Примеры источников событий

Источник событий	Описание
Кнопка	Извещает о событии действия при нажатии кнопки
Флажок	Извещает о событиях элемента при установке и сбросе флажка
Переключатель	Извещает о событиях элемента при изменении выбора
Список	Извещает о событиях действия при двойном щелчке на элементе; извещает о событии элемента при выделении или снятии выделения с элемента
Пункт меню	Извещает о событиях действия при выборе пункта меню; извещает о событии элемента при установке и сбросе флажка с пункта меню
Полоса прокрутки	Извещает о событиях настройки при манипуляциях с полосой прокрутки
Текстовые компоненты	Извещает о текстовых событиях, когда пользователь вводит символ
Окно	Извещает о событиях окна при активизации, закрытии, деактивизации, развертывании, свертывании, открытии окна или выходе из окна

Интерфейсы слушателей событий

Как уже объяснялось, модель делегирования событий состоит из двух частей: источников и слушателей. Слушатели создаются при реализации одного или нескольких интерфейсов, определенных в пакете `java.awt.event`. Когда происходит событие, его источник вызывает соответствующий метод, определенный в слушателе, и передает объект события в качестве аргумента. В табл. 23.9 перечислены часто используемые интерфейсы слушателей и представлено краткое описание методов, определяемых ими. В следующих разделах рассматриваются специфические методы, содержащиеся в каждом интерфейсе.

Таблица 23.9. Часто используемые интерфейсы слушателей событий

Интерфейс	Описание
`ActionListener`	Определяет один метод для принятия событий действия
`AdjustmentListener`	Определяет один метод для принятия событий настройки
`ComponentListener`	Определяет четыре метода для распознавания сокрытия, перемещения, изменения размера или отображения компонента
`ContainerListener`	Определяет два метода для распознавания того, когда компонент добавляется в контейнер либо исключается из него
`FocusListener`	Определяет два метода для распознавания получения или утраты компонентом фокуса клавиатурного ввода
`ItemListener`	Определяет один метод, указывающий момент изменения состояния элемента
`KeyListener`	Определяет три метода, распознающих нажатие, отпускание клавиши либо ввод символа

Окончание табл. 23.9

Интерфейс	Описание
MouseListener	Определяет пять методов, распознающих щелчок мыши, момент входа курсора мыши на поле компонента, его выхода за пределы компонента, нажатия и отпускания кнопок мыши
MouseMotionListener	Определяет два метода для распознавания перетаскивания или движения мыши
MouseWheelListener	Определяет один метод, распознающий движение колесика мыши
TextListener	Определяет один метод, распознающий изменение текстового значения
WindowFocusListener	Определяет два метода для распознавания получения или утраты окном фокуса ввода
WindowListener	Определяет семь методов, распознающих активизацию окна, закрытие, деактивизацию, развертывание, свертывание, открытие или выход

Интерфейс ActionListener

Этот интерфейс определяет метод `actionPerformed()`, вызываемый при событии действия. Его общая форма показана ниже.

```
void actionPerformed(ActionEvent ae)
```

Интерфейс AdjustmentListener

Этот интерфейс определяет метод `adjustmentValueChanged()`, вызываемый при событии настройки. Его общая форма такова.

```
void adjustmentValueChanged(AdjustmentEvent ae)
```

Интерфейс ComponentListener

Этот интерфейс определяет четыре метода, вызываемых при изменении размера компонента, его перемещении, отображении или сокрытии. Их общая форма представлена ниже.

```
void componentResized(ComponentEvent ce)
void componentMoved(ComponentEvent ce)
void componentShown(ComponentEvent ce)
void componentHidden(ComponentEvent ce)
```

Интерфейс ContainerListener

Этот интерфейс содержит два метода. Когда компонент добавляется к контейнеру, вызывается метод `componentAdded()`. Когда же компонент удаляется из контейнера, вызывается метод `componentRemoved()`. Их общие формы выглядят следующим образом.

```
void componentAdded(ContainerEvent ce)
void componentRemoved(ContainerEvent ce)
```

Интерфейс FocusListener

Этот интерфейс определяет два метода. Когда компонент получает фокус клавиатурного ввода, вызывается метод focusGained(). Когда компонент теряет фокус ввода, вызывается метод focusLost(). Обобщенная форма этих методов показана ниже.

```
void focusGained(FocusEvent fe)
void focusLost(FocusEvent fe)
```

Интерфейс ItemListener

Этот интерфейс определяет метод itemStateChanged(), вызываемый при изменении состояния элемента. Его общая форма такова.

```
void itemStateChanged(ItemEvent ie)
```

Интерфейс KeyListener

Этот интерфейс определяет три метода. Методы keyPressed() и keyReleased() вызываются, соответственно, при нажатии и отпускании клавиш. Метод keyTyped() вызывается при вводе символа.

Например, если пользователь нажимает и отпускает клавишу <A>, последовательно происходит три события: нажатие клавиши, ввод символа, отпускание клавиши. Если пользователь нажимает и отпускает клавишу <Home>, происходит последовательно только два события: нажатие клавиши и ее отпускание.

Общие формы этих методов выглядят следующим образом.

```
void keyPressed(KeyEvent ke)
void keyReleased(KeyEvent ke)
void keyTyped(KeyEvent ke)
```

Интерфейс MouseListener

Этот интерфейс определяет пять методов. Если кнопка мыши нажата и отпущена в одной и той же точке, вызывается метод mouseClicked(). Когда курсор мыши входит на поле компонента, вызывается метод mouseEntered(). Когда же он покидает поле компонента, вызывается метод mouseExited(). Методы mousePressed() и mouseReleased() вызываются, соответственно, когда кнопка мыши нажимается и отпускается. Общие формы этих методов представлены ниже.

```
void mouseClicked(MouseEvent me)
void mouseEntered(MouseEvent me)
void mouseExited(MouseEvent me)
void mousePressed(MouseEvent me)
void mouseReleased(MouseEvent me)
```

Интерфейс MouseMotionListener

Этот интерфейс определяет два метода. Метод mouseDragged() вызывается множество раз при перетаскивании объекта мышью. Метод mouseMoved() вызывается множество раз при перемещении мыши. Их общая форма такова.

```
void mouseDragged(MouseEvent me)
void mouseMoved(MouseEvent me)
```

Интерфейс MouseWheelListener

Этот интерфейс определяет метод mouseWheelMoved(), вызываемый при прокрутке колесика мыши. Его общая форма показана ниже.

```
void mouseWheelMoved(MouseWheelEvent mwe)
```

Интерфейс TextListener

Этот интерфейс определяет метод textChanged(), вызываемый при изменении содержимого текстовой области или текстового поля. Его общая форма выглядит следующим образом.

```
void textChanged(TextEvent te)
```

Интерфейс WindowFocusListener

Этот интерфейс определяет два метода — windowGainedFocus() и windowLostFocus(). Они вызываются, когда окно получает и утрачивает фокус ввода. Их общая форма показана ниже.

```
void windowGainedFocus(WindowEvent we)
void windowLostFocus(WindowEvent we)
```

Интерфейс WindowListener

Этот интерфейс определяет семь методов. Методы windowActivated() и windowDeactivated() вызываются, когда окно, соответственно, активизируется и деактивизируется.

Если окно сворачивается в пиктограмму, вызывается метод windowIconified(). Когда окно разворачивается, вызывается метод windowDeIconified(). Когда окно открывается и закрывается, вызываются методы windowOpened() и windowClosed(). Метод windowClosing() вызывается при закрытии окна. Общие формы этих методов выглядят следующим образом.

```
void windowActivated(WindowEvent we)
void windowClosed(WindowEvent we)
void windowClosing(WindowEvent we)
void windowDeactivated(WindowEvent we)
void windowDeiconified(WindowEvent we)
void windowIconified(WindowEvent we)
void windowOpened(WindowEvent we)
```

Использование модели делегирования событий

Теперь, когда вы ознакомились с теорией, лежащей в основе модели делегирования событий, и получили представление о различных ее компонентах, наступило время обратиться к практике. Использовать модель делегирования событий довольно легко. Нужно просто выполнить следующие два действия.

1. Реализовать соответствующий интерфейс в слушателе, чтобы он мог принимать события нужного типа.
2. Реализовать код регистрации и отмены регистрации (при необходимости) слушателя как получателя уведомлений о событии.

Следует помнить, что источник может извещать о нескольких типах событий. Каждое событие должно быть зарегистрировано отдельно. К тому же объект может подписаться на получение нескольких типов событий и должен реализовать все интерфейсы, необходимые для получения этих событий.

Чтобы увидеть, как модель делегирования событий работает на практике, мы разберем примеры, обрабатывающие два часто используемых генератора событий — событий мыши и клавиатуры.

Обработка событий мыши

Чтобы обработать события мыши, следует реализовать интерфейсы MouseListener и MouseMotionListener. (Вы можете также реализовать интерфейс MouseWheelListener, но мы не станем делать этого здесь.) Следующий аплет демонстрирует весь процесс. Он отображает текущие координаты мыши в строке состояния. Всякий раз, когда нажимается кнопка, отображается слово "Down" в точке, где находится курсор мыши. При каждом отпускании кнопки отображается слово "Up". При щелчке на кнопке, в левом верхнем углу отображаемой области аплета выводится сообщение "Mouse clicked".

При входе и выходе курсора мыши из поля окна аплета в левом верхнем углу отображаемой области аплета также выводится соответствующее сообщение. При перетаскивании мышью отображается символ "*", сопровождающий курсор мыши. Обратите внимание: две переменные — mouseX и mouseY — сохраняют местоположение курсора мыши, когда происходят события нажатия и отпускания кнопки, а также события перетаскивания. Эти координаты затем используются методом paint() для отображения вывода в точке возникновения события.

```
// Использование обработчиков событий мыши.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="MouseEvents" width=300 height=100>
</applet>
*/

public class MouseEvents extends Applet
implements MouseListener, MouseMotionListener {

    String msg = "";
    int mouseX = 0, mouseY = 0; // координаты курсора мыши

    public void init() {
        addMouseListener(this);
        addMouseMotionListener(this);
    }

    // Обработка щелчка мыши.
    public void mouseClicked(MouseEvent me) {
        // сохранить координаты
        mouseX = 0;
        mouseY = 10;
        msg = "Mouse clicked.";
     // msg = "Щелчок кнопкой мыши.";
        repaint();
    }

    // Обработка входа курсора мыши.
    public void mouseEntered(MouseEvent me) {
```

```
        // сохранить координаты
        mouseX = 0;
        mouseY = 10;
        msg = "Mouse entered.";
     // msg = "Курсор мыши вошел.";
        repaint();
    }

    // Обработка выхода курсора мыши.
    public void mouseExited(MouseEvent me) {
        // сохранить координаты
        mouseX = 0;
        mouseY = 10;
        msg = "Mouse exited.";
     // msg = "Курсор мыши вышел.";
        repaint();
    }

    // Обработка нажатия кнопки.
    public void mousePressed(MouseEvent me) {
        // сохранить координаты
        mouseX = me.getX();
        mouseY = me.getY();
        msg = "Down";
     // msg = "Вниз";
        repaint();
    }

    // Обработка отпускания кнопки.
    public void mouseReleased(MouseEvent me) {
        // сохранить координаты
        mouseX = me.getX();
        mouseY = me.getY();
        msg = "Up";
     // msg = "Вверх";
        repaint();
    }

    // Обработка перетаскивания мышью.
    public void mouseDragged(MouseEvent me) {
        // сохранить координаты
        mouseX = me.getX();
        mouseY = me.getY();
        msg = "*";
        showStatus("Dragging mouse at " + mouseX + ", " + mouseY);
     // showStatus("Перетаскивание мыши в " + mouseX + ", " + mouseY);
        repaint();
    }

    // Обработка движения мыши.
    public void mouseMoved(MouseEvent me) {
        // Показать состояние
        showStatus("Moving mouse at " + me.getX() + ", " + me.getY());
     // showStatus("Перемещение мыши в " + me.getX() + ", " + me.getY());
    }

    // Отобразить msg в окне аплета в текущей позиции X,Y.
    public void paint(Graphics g) {
        g.drawString(msg, mouseX, mouseY);
    }
}
```

Пример работы этого аплета показан на рис. 23.1.

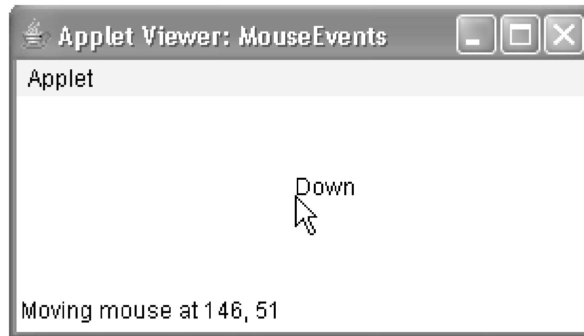

Рис. 23.1. Пример работы аплета, использующего обработчики событий мыши

Рассмотрим этот пример внимательнее. Класс MouseEvents расширяет класс Applet и реализует интерфейсы MouseListener и MouseMotionListener. Эти два интерфейса содержат методы, принимающие и обрабатывающие различные типы событий мыши. Обратите внимание на то, что аплет одновременно является и источником, и слушателем этих событий. Это работает, потому что класс Component, применяющий методы addMouseListener() и addMouseMotionListener(), является суперклассом класса Applet. Использование одного и того же объекта в качестве источника и слушателя событий типично для аплетов.

Внутри метода init() аплет регистрирует самого себя в качестве слушателя событий мыши. Это осуществляется методами addMouseListener() и addMouseMotionListener(), которые, как уже упоминалось, являются членами класса Component. Эти методы показаны ниже.

```
void addMouseListener(MouseListener ml)
void addMouseMotionListener(MouseMotionListener mml)
```

Здесь *ml* — ссылка на объект, принимающий события мыши, а *mml* — ссылка на объект, принимающий события перемещения мыши. В данной программе в обоих методах используется один и тот же объект.

Затем аплет реализует все методы, определенные в интерфейсах MouseListener и MouseMotionListener. Таким образом, существуют обработчики для всех событий мыши. Каждый из них обрабатывает собственное событие, а затем возвращает управление.

Обработка событий клавиатуры

Для обработки событий клавиатуры, используется та же общая архитектура, что и в примере с событиями мыши, приведенном в предыдущем разделе. Отличие, конечно, в том, что здесь вы будете реализовывать интерфейс KeyListener.

Прежде чем обращаться к примеру, полезно еще раз рассмотреть процесс извещения о клавиатурных событиях. При нажатии клавиши происходит событие KEY_PRESSED. Это приводит к вызову обработчика событий keyPressed(). При отпускании клавиши происходит событие KEY_RELEASED и выполняется обработчик keyReleased(). Если нажатие клавиши создает символ, то также происходит событие KEY_TYPED и выполняется обработчик keyTyped(). Таким образом, всякий раз, когда пользователь нажимает клавишу, происходит как минимум два, а то и три события. Если вас интересуют действительные символы, введенные с клавиатуры, то можете игнорировать информацию о нажатии и отпускании клавиш. Но если ваша программа должна обрабатывать специальные клавиши вроде клавиш

со стрелками или функциональных клавиш, то их следует отслеживать в обработчике keyPressed().

В следующем аплете демонстрируется клавиатурный ввод. Он отображает нажатые клавиши в окне аплета и демонстрирует в строке состояния, нажата клавиша или отпущена.

```
// Использование обработчиков событий клавиатуры.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="SimpleKey" width=300 height=100>
</applet>
*/

public class SimpleKey extends Applet
implements KeyListener {

    String msg = "";
    int X = 10, Y = 20; // координаты вывода

    public void init() {
        addKeyListener(this);
    }

    public void keyPressed(KeyEvent ke) {
        showStatus("Key Down");
     // showStatus("Клавиша нажата");
    }

    public void keyReleased(KeyEvent ke) {
        showStatus("Key Up");
     // showStatus("Клавиша отпущена");
    }

    public void keyTyped(KeyEvent ke) {
        msg += ke.getKeyChar();
        repaint();
    }
    // Отображение нажатых клавиш.
    public void paint(Graphics g) {
        g.drawString(msg, X, Y);
    }
}
```

Пример работы этого аплета показан на рис. 23.2.

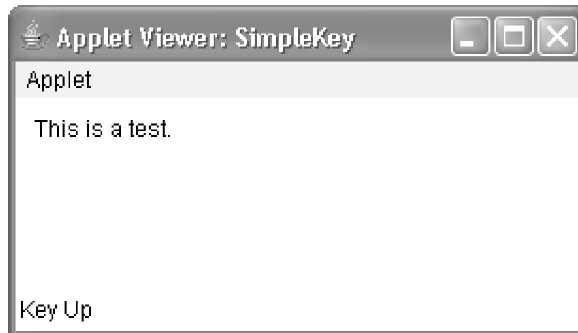

Рис. 23.2. Пример работы аплета, использующего обработчики событий клавиатуры

Если хотите обрабатывать специальные клавиши, такие как клавиши со стрелками и функциональные клавиши, то следует реагировать на них в обработчике `keyPressed()`. Такие клавиши в обработчике `keyTyped()` не доступны. Чтобы идентифицировать эти клавиши, можно использовать коды виртуальных клавиш. Например, следующий аплет выводит имена некоторых специальных клавиш.

```
// Использование некоторых кодов виртуальных клавиш.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="KeyEvents" width=300 height=100>
</applet>
*/

public class KeyEvents extends Applet
implements KeyListener {

    String msg = "";
    int X = 10, Y = 20; // координаты вывода

    public void init() {
    addKeyListener(this);
}

public void keyPressed(KeyEvent ke) {
    showStatus("Key Down");
 // showStatus("Клавиша нажата");
    int key = ke.getKeyCode();

    switch(key) {
        case KeyEvent.VK_F1:
            msg += "<F1>";
            break;
        case KeyEvent.VK_F2:
            msg += "<F2>";
            break;
        case KeyEvent.VK_F3:
            msg += "<F3>";
            break;
        case KeyEvent.VK_PAGE_DOWN:
            msg += "<PgDn>";
            break;
        case KeyEvent.VK_PAGE_UP:
            msg += "<PgUp>";
            break;
        case KeyEvent.VK_LEFT:
            msg += "<Left Arrow>";
         // msg += "<Стрелка влево>";
            break;
        case KeyEvent.VK_RIGHT:
            msg += "<Right Arrow>";
         // msg += "<Стрелка вправо>";
            break;
    }
    repaint();
}

public void keyReleased(KeyEvent ke) {
    showStatus("Key Up");
    showStatus("Клавиша отпущена");
}
```

```
public void keyTyped(KeyEvent ke) {
    msg += ke.getKeyChar();
    repaint();
}

// Отобразить нажатые клавиши.
public void paint(Graphics g) {
    g.drawString(msg, X, Y);
}
```

Пример работы этого аплета показан на рис. 23.3.

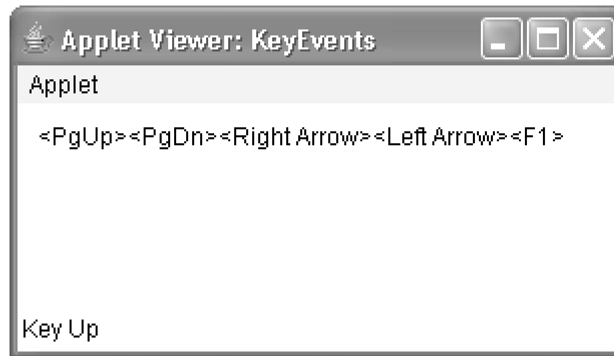

Рис. 23.3. Пример работы аплета, использующего некоторые коды виртуальных клавиш

Процедуры, показанные в приведенных примерах обработки событий мыши и клавиатуры, могут быть обобщены для обработки событий любого типа, включая события элементов управления. В последующих главах вы увидите множество примеров обработки событий других типов, но все они следуют одной и той же базовой структуре, что и только что описанные программы.

Классы адаптеров

В Java имеется специальное средство, называемое *классом адаптера*, который в некоторых ситуациях упрощает реализацию обработчиков событий. Класс адаптера предлагает пустую реализацию всех методов интерфейса слушателя событий. Классы адаптеров удобны, когда вы хотите принимать и обрабатывать только некоторые события, обрабатываемые определенным интерфейсом слушателя. Вы можете определить новый класс для использования в качестве слушателя событий, расширив один из классов адаптеров и реализовав только те события, в которых вы заинтересованы.

Например, класс `MouseMotionAdapter` имеет два метода `mouseDragged()` и `mouseMoved()`, которые определены в интерфейсе `MouseMotionListener`. Если вы заинтересованы только в событиях перетаскивания мыши, можете просто расширить класс `MouseMotionAdapter` и переопределить метод `mouseDragged()`. Пустая реализация метода `mouseMoved()` обработает события перемещения мыши за вас.

В табл. 23.10 перечислены часто используемые классы адаптеров пакета `java.awt.event` и отмечены интерфейсы, реализуемые каждым из них.

Таблица 23.10. Часто используемые интерфейсы слушателей, реализуемые классами адаптеров

Класс адаптера	Интерфейс слушателя
ComponentAdapter	ComponentListener
ContainerAdapter	ContainerListener
FocusAdapter	FocusListener
KeyAdapter	KeyListener
MouseAdapter	MouseListener
MouseMotionAdapter	MouseMotionListener
WindowAdapter	WindowListener

В следующем примере демонстрируется применение адаптера. Он отображает сообщение в строке состояния средства просмотра аплетов, когда выполняется щелчок кнопкой мыши или перетаскивание. Однако все прочие события мыши игнорируются. Программа состоит из трех классов. Класс AdapterDemo расширяет класс Applet. Его метод init() создает экземпляр класса MyMouseAdapter и регистрирует этот объект для получения уведомлений о событиях мыши. Также он создает экземпляр класса MyMouseMotionAdapter и регистрирует его для получения уведомлений о событиях перемещения мыши. Оба конструктора принимают ссылку на аплет в качестве аргумента.

Класс MyMouseAdapter расширяет класс MouseAdapter и переопределяет метод mouseClicked(). Все прочие события мыши игнорируются кодом, унаследованным от класса MouseAdapter. Класс MyMouseMotionAdapter расширяет класс MouseMotionAdapter и переопределяет метод mouseDragged(). Другое событие перемещения мыши игнорируется кодом, унаследованным от класса MouseMotionAdapter.

Обратите внимание на то, что оба класса слушателей событий сохраняют ссылку на аплет. Эта информация предоставляется в виде аргумента и используется позднее для вызова метода showStatus().

```
// Демонстрация применения адаптера.
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
/*
<applet code="AdapterDemo" width=300 height=100>
</applet>
*/

public class AdapterDemo extends Applet {
    public void init() {
        addMouseListener(new MyMouseAdapter(this));
        addMouseMotionListener(new MyMouseMotionAdapter(this));
    }
}

class MyMouseAdapter extends MouseAdapter {

    AdapterDemo adapterDemo;
    public MyMouseAdapter(AdapterDemo adapterDemo) {
        this.adapterDemo = adapterDemo;
    }
    // Обработка щелчка мыши.
    public void mouseClicked(MouseEvent me) {
```

```
        adapterDemo.showStatus("Mouse clicked");
     // adapterDemo.showStatus("Щелчок кнопкой мыши");
    }
}

class MyMouseMotionAdapter extends MouseMotionAdapter {
    AdapterDemo adapterDemo;
    public MyMouseMotionAdapter(AdapterDemo adapterDemo) {
        this.adapterDemo = adapterDemo;
    }

    // Обработка перетаскивания мыши.
    public void mouseDragged(MouseEvent me) {
        adapterDemo.showStatus("Mouse dragged");
        // adapterDemo.showStatus("Перетаскивание мышью");
    }
}
```

Как видите, отсутствие необходимости реализовать все методы, определенные интерфейсами `MouseMotionListener` и `MouseListener`, позволяет сэкономить значительное количество усилий и избавляет ваш код от перегрузки пустыми методами. В качестве упражнения можете попробовать переписать один из приведенных ранее примеров, обрабатывающих клавиатурный ввод, с использованием класса `KeyAdapter`.

Вложенные классы

Основы вложенных классов были описаны в главе 7. Сейчас вы убедитесь, насколько они важны. Напомним, что *вложенный класс* — это класс, определенный внутри другого класса или даже внутри выражения. В настоящем разделе проиллюстрировано использование вложенных классов для упрощения кода в случае классов адаптеров событий.

Чтобы понять выгоду, которую обеспечивают вложенные классы, рассмотрим аплет, приведенный в следующем листинге. В нем *не используются* вложенные классы. Его назначение — отобразить строку `"Mouse Pressed"` в строке состояния средства просмотра аплетов или браузера, когда нажата кнопка мыши. В этой программе присутствует еще два класса верхнего уровня. Класс `MousePressedDemo` расширяет класс `Applet`, а класс `MyMouseAdapter` — класс `MouseAdapter`. Метод `init()` в классе `MousePressedDemo` создает экземпляр класса `MyMouseAdapter` и предоставляет этот объект в качестве аргумента методу `addMouseListener()`.

Обратите внимание на то, что ссылка на аплет выступает в качестве аргумента конструктора класса `MyMouseAdapter`. Эта ссылка сохраняется в переменной экземпляра для последующего использования методом `mousePressed()`. Когда нажимается кнопка мыши, вызывается метод `showStatus()` аплета через сохраненную ссылку на аплет. Другими словами, метод `showStatus()` вызывается относительно ссылки на аплет, сохраненной в объекте класса `MyMouseAdapter`.

```
// В этом аплете НЕ используются вложенные классы.
import java.applet.*;
import java.awt.event.*;
/*
<applet code="MousePressedDemo" width=200 height=100>
</applet>
*/

public class MousePressedDemo extends Applet {
```

```
    public void init() {
        addMouseListener(new MyMouseAdapter(this));
    }
}

class MyMouseAdapter extends MouseAdapter {
    MousePressedDemo mousePressedDemo;
    public MyMouseAdapter(MousePressedDemo mousePressedDemo) {
        this.mousePressedDemo = mousePressedDemo;
    }
    public void mousePressed(MouseEvent me) {
        mousePressedDemo.showStatus("Mouse Pressed");
     // mousePressedDemo.showStatus("Кнопка мыши нажата");
    }
}
```

В следующем листинге показано, как можно усовершенствовать предыдущую программу, используя вложенный класс. Здесь класс `InnerClassDemo` — это класс верхнего уровня, расширяющий класс `Applet`. Класс `MyMouseAdapter` — это вложенный класс, расширяющий класс `MouseAdapter`. Поскольку класс `MyMouseAdapter` определен внутри области видимости класса `InnerClassDemo`, он имеет доступ ко всем переменным и методам, находящимся в контексте этого класса. Таким образом, метод `mousePressed()` может вызывать метод `showStatus()` непосредственно. Больше нет необходимости делать это через сохраненную ссылку на аплет. А потому не нужно и передавать конструктору `MyMouseAdapter()` ссылку на вызывающий объект.

```
// Демонстрация применения вложенного класса.
import java.applet.*;
import java.awt.event.*;
/*
<applet code="InnerClassDemo" width=200 height=100>
</applet>
*/

public class InnerClassDemo extends Applet {
    public void init() {
        addMouseListener(new MyMouseAdapter());
    }
    class MyMouseAdapter extends MouseAdapter {
        public void mousePressed(MouseEvent me) {
            showStatus("Mouse Pressed");
         // showStatus("Кнопка мыши нажата");
        }
    }
}
```

Анонимные вложенные классы

Анонимный вложенный класс — это класс, которому не назначено имя. В этом разделе проиллюстрируем, как анонимный вложенный класс может облегчить написание обработчиков событий. Рассмотрим аплет, показанный в следующем листинге. Как и раньше, его назначение — отобразить строку `"Mouse Pressed"` в строке состояния средства просмотра аплетов или браузера, когда нажата кнопка мыши.

```
// Демонстрация применения анонимного вложенного класса.
import java.applet.*;
import java.awt.event.*;
/*
```

```
<applet code="AnonymousInnerClassDemo" width=200 height=100>
</applet>
*/

public class AnonymousInnerClassDemo extends Applet {
    public void init() {
        addMouseListener(new MouseAdapter() {
            public void mousePressed(MouseEvent me) {
                showStatus("Mouse Pressed");
              // showStatus("Кнопка мыши нажата");
            }
        });
    }
}
```

В этой программе присутствует только один класс верхнего уровня — AnonymousInnerClassDemo. Метод init() вызывает метод addMouseListener(). Его аргументом служит выражение, определяющее и создающее экземпляр анонимного вложенного класса. Давайте тщательно проанализируем это выражение.

Синтаксис new MouseAdapter(){...} указывает компилятору, что код между фигурными скобками определяет анонимный вложенный класс. Более того, этот класс расширяет класс MouseAdapter. Этот новый класс не имеет имени, но его экземпляр автоматически создается при выполнении данного выражения.

Поскольку анонимный вложенный класс определен внутри контекста класса AnonymousInnerClassDemo, он имеет доступ ко всем переменным и методам, находящимся в контексте данного класса. Поэтому он может вызывать метод showStatus() непосредственно.

Как видите, именованные и анонимные вложенные классы решают некоторые досадные проблемы простым и эффективным способом. Они также позволяют вам создавать более эффективный код.