ИНТЕРФЕЙС ПОЛЬЗОВАТЕЛЯ

Методические указания к лабораторной работе № 8

***Цель работы**:* ознакомиться с основными классами и интерфейсами пакета java.awt, научиться создавать пользовательский интерфейс, используя стандартные компоненты.

Теоретическая справка

AWT – Abstract Window Toolkit. Слово abstract в названии указывает, что все стандартные компоненты не являются самостоятельными, а работают в связке с соответствующими элементами операционной системы.

Дерево компонентов

Component

Абстрактный класс Component является базовым для всех компонент AWT и описывает их основные свойства. Визуальный компонент в AWT имеет прямоугольную форму, может быть отображен на экране и может взаимодействовать с пользователем.

Рассмотрим основные свойства этого класса.

Положение

Положение компонента описывается двумя целыми числами (тип int) x и y. В Java (как и во многих языках программирования) ось x проходит традиционно – горизонтально, направлена вправо, а ось y – вертикально, но направлена вниз, а не вверх, как принято в математике.

Для описания положения компонента предназначен специальный класс – Point (точка). В этом классе определено два public int поля x и y, а также множество конструкторов и вспомогательных методов для работы с ними. Класс Point применяется во многих типах AWT, где надо задать точку на плоскости.

Для компонента эта точка задает положение левого верхнего угла.

Установить положение компонента можно с помощью метода setLocation, который может принимать в качестве аргументов пару целых чисел, либо Point. Узнать текущее положение можно с помощью метода getLocation(), возвращающего Point, либо с помощью методов getX() и getY(), которые появились с версии Java 1.2.

Размер

Как было сказано, компонент AWT имеет прямоугольную форму, а потому его размер описывается также двумя целочисленными параметрами – width (ширина) и height

(высота). Для описания размера существует специальный класс Dimension (размер), в котором определено два public int поля width и height, а также вспомогательные методы.

Установить размер компонента можно с помощью метода setSize, который может принимать в качестве аргументов пару целых чисел, либо Dimension. Узнать текущее положение можно с помощью метода getSize(), возвращающего Dimension, либо с помощью методов getWidth() и getHeight(), которые появились с версии Java 1.2.

Совместно положение и размер компонента задают его границы. Область, занимаемую компонентом, можно описать либо четырьмя числами (x, y, width, height), либо экземплярами классов Point и Dimension, либо специальным классом Rectangle (прямоугольник). Как легко догадаться, в этом классе определено четыре public int поля, с которыми можно работать и в виде пары объектов Point и Dimension.

Задать границу объекта можно с помощью метода setBounds, который может принимать четыре числа, либо Rectangle. Узнать текущее значение можно с помощью метода getBounds(), возвращающего Rectangle.

Видимость

Существующий компонент может быть как виден пользователю, так и быть скрытым. Это свойство описывается булевским параметром visible. Методы для управления – setVisible, принимающий булевский параметр, и isVisible, возвращающий текущее значение.

Разумеется, невидимый компонент не может взаимодействовать с пользователем.

Доступность

Даже если компонент отображается на экране и виден пользователю, он может не взаимодействовать с ним. В результате события от клавиатуры, или мыши не будут получаться и обрабатываться компонентом. Такой компонент называется disabled. Если же компонент активен, его называют enabled. Как правило, компонент некоторым образом меняет свой внешний вид, когда становится недоступным (например, становится серым, менее заметным), но, вообще говоря, это необязательно (хотя очень удобно для пользователя).

Для изменения этого свойства применяется метод setEnabled, принимающий булевский параметр (true соответствует enabled, false – disabled), а для получения текущего значения – isEnabled.

Цвета

Разумеется, для построения современного графического интерфейса пользователя необходима работа с цветами.

Компонент обладает двумя свойствами, описывающими цвета, – foreground и background цвета. Первое свойство задает, каким цветом выводить надписи, рисовать линии и т.д. Второе – задает цвет фона, которым закрашивается вся область, занимаемая компонентом, перед тем, как прорисовывается внешний вид.

Для задания цвета в AWT используется специальный класс Color. Этот класс обладает довольно обширной функциональностью, поэтому рассмотрим основные характеристики.

Цвет задается 3 целочисленными характеристиками, соответствующими модели RGB, – красный, синий, зеленый. Каждая из них может иметь значение от 0 до 255 (тем не менее, их тип определен как int). В результате (0, 0, 0) соответствует черному, а (255, 255, 255) – белому.

Класс Color является неизменяемым, то есть, создав экземпляр, соответствующий какому-либо цвету, изменить параметры RGB уже невозможно. Это позволяет объявить в классе Color ряд констант, описывающих базовые цвета: белый, черный, красный, желтый и так далее. Например, вместо того, чтобы задавать синий цвет числовыми параметрами (0, 0, 255), можно воспользоваться константами Color.blue или Color.BLUE (второй вариант появился в более поздних версиях).

Для работы со свойством компонента foreground применяют методы setForeground и getForeground, а для background – setBackground и getBackground.

Шрифт

Раз изображение компонента может включать в себя надписи, необходимо свойство, описывающее шрифт для их прорисовки.

Для задания шрифта в AWT существует специальный класс Font, который включает в себя три параметра – имя шрифта, размер и стиль.

Имя шрифта задает внешний стиль отображения символов. Имена претерпели ряд изменений с развитием Java. В версии 1.0 требовалось, чтобы JVM поддерживала следующие шрифты: TimesRoman, Helvetica, Courier. Могут поддерживаться и другие семейства, это зависит от деталей реализации конкретной виртуальной машины. Чтобы узнать полный список во время исполнения программы, можно воспользоваться методом утилитного класса Toolkit. Экземпляры этого класса нельзя создать вручную, поэтому полностью такой запрос будет выглядеть следующим образом:

```
Toolkit.getDefaultToolkit().getFontList()
```

В результате будет возвращен массив строк-имен семейств поддерживаемых шрифтов.

В Java 1.1 три обязательных имени были объявлены deprecated. Вместо них был введен новый список, который содержал более универсальные названия, не зависящие от конкретной операционной системы: Serif, SansSerif, Monospaced.

В Java 2 библиотека AWT была существенно пересмотрена и дополнена. Чтобы устранить неоднозначности с разной поддержкой шрифтов на разных платформах, было произведено разделение на логические и физические шрифты. Вторая группа определяется возможностями операционной системы, это те же шрифты, которые могут использовать другие программы, запущенные на этой платформе.

Первая группа состоит из 5 обязательных семейств (добавились Dialog и DialogInput). JVM устанавливает соответствие между ними и наиболее подходящими из доступных физических шрифтов.

Метод getFontList класса Toolkit был объявлен deprecated. Теперь получить список всех доступных физических шрифтов можно следующим образом:

```
GraphicsEnvironment.
  getLocalGraphicsEnvironment().
  getAvailableFontFamilyNames()
```

Класс Font является неизменяемым. После создания можно узнать заданное логическое имя (метод getName) и соответствующее ему физическое имя семейства (метод getFamily).

Вернемся к остальным параметрам, необходимым для создания экземпляра Font. Размер шрифта определяет, очевидно, величину символов. Однако конкретные значения измеряются не в пикселах, а в условных единицах (как и во многих текстовых редакторах). Для разных семейств шрифтов символы одинакового размера могут иметь различную ширину и высоту, измеренную в пикселах.

Как и в случае имени шрифта, программист может указать любое значение размера, а JVM поставит ему в соответствие максимально близкий из доступных.

Наконец, последний параметр – стиль. Этот параметр определяет, будет ли шрифт жирным, наклонным и т.д. Если никакие из этих свойств не требуются, указывается Font.PLAIN (параметр имеет тип int и в классе Font определен набор констант для удобства работы с ним). Значение Font.BOLD задает жирный шрифт, а Font.ITALIC – наклонный. Для сочетания этих свойств (жирный наклонный шрифт) необходимо произвести логическое сложение: Font.BOLD|Font.ITALIC.

Для работы с этим свойством класса Component предназначены методы setFont и getFont.

Итак, мы рассмотрели основные свойства класса Component. Как легко видеть, все они предназначены для описания графического представления компонента, то есть отображения на экране.

Существует еще одно важное свойство другого характера. Очевидно, что практически всегда пользовательский интерфейс состоит из более чем одного компонента. В больших приложениях их обычно гораздо больше. Для удобства организации работы с ними компоненты объединяются в контейнеры. В AWT существует класс, который так и называется – Container. Его рассмотрение – наша следующая тема. Важно отметить, что компонент может находиться лишь в одном контейнере – при попытке добавить его в другой он удаляется из первого. Рассматриваемое свойство как раз и отвечает за связь компонента с контейнером. Свойство называется parent. Благодаря нему компонент всегда "знает", в каком контейнере он находится.

Container

Контейнер описывается классом Container, который является наследником Component, а значит, обладает всеми свойствами графического компонента. Однако основная его задача – группировать другие компоненты. Для этого в нем объявлен целый ряд методов. Для добавления служит метод add, для удаления – remove и removeAll (последний удаляет все компоненты).

Добавляемые компоненты хранятся в упорядоченном списке, поэтому для удаления можно указать либо ссылку на компонент, который и будет удален, либо его порядковый номер в контейнере. Также определены методы для получения компонент, присутствующих в контейнере, – все они довольно очевидны, поэтому перечислим их с краткими пояснениями:

getComponent(int n) – возвращает компонент с указанным порядковым номером;

getComponents() – возвращает все компоненты в виде массива;

getComponentCount() – возвращает количество компонент;

getComponentAt(int x, int y) или (Point p) – возвращает компонент, который включает в себя указанную точку;

findComponentAt(int x, int y) или (Point p) – возвращает видимый компонент, включающий в себя указанную точку.

Мы уже знаем, что положение компонента (location) задается координатами левого верхнего угла. Важно, что эти значения отсчитываются от левого верхнего угла контейнера, который таким образом является центром системы координат для каждого находящегося в нем компонента. Если важно расположение компонента на экране безотносительно его контейнера, можно воспользоваться методом getLocationOnScreent.

Благодаря наследованию контейнер также имеет свойство size. Этот размер задается независимо от размера и положения вложенных компонент. Таким образом, компоненты могут располагаться частично или полностью за пределами своего контейнера (что это означает, будет рассмотрено ниже, но принципиально это допустимо).

Раз контейнер наследуется от Component, он сам является компонентом, а значит, может быть добавлен в другой, вышестоящий контейнер. В то же время компонент может находиться лишь в одном контейнере. Это означает, что все элементы сложного пользовательского интерфейса объединяются в иерархическое дерево. Такая организация не только облегчает операции над ними, но и задает основные свойства всей работы AWT. Одним из них является принцип отрисовки компонентов.

Алгоритм отрисовки

Начнем с отрисовки отдельного компонента – что определяет его внешний вид?

Для этой задачи предназначен метод paint. Этот метод вызывается каждый раз, когда необходимо отобразить компонент на экране. У него есть один аргумент, тип которого – абстрактный класс Graphics. В этом классе определено множество методов для отрисовки простейших графических элементов – линий, прямоугольников и многоугольников, окружностей и овалов, текста, картинок и т.д.

Наследники класса Component переопределяют метод paint и, пользуясь методами Graphics, задают алгоритм прорисовки своего внешнего вида:

```
public void paint(Graphics g) {
  g.drawLine(0, 0, getWidth(), getHeight());
  g.drawLine(0, getHeight(), getWidth(), 0);
}
```

Методы класса Graphics для отрисовки

Рассмотрим обзорно методы класса Graphics, предназначенные для отрисовки.

drawLine(x1, y1, x2, y2)

Этот метод отображает линию толщиной в 1 пиксел, проходящую из точки (x1, y1) в (x2, y2). Именно он использовался в предыдущем примере.

drawRect(int x, int y, int width, int height)

Этот метод отображает прямоугольник, чей левый верхний угол находится в точке (x, y), а ширина и высота равняются width и height соответственно. Правая сторона пройдет по линии x+width, а нижняя – y+height.

fillRect(int x, int y, int width, int height)

Этот метод закрашивает прямоугольник. Левая и правая стороны прямоугольника проходят по линиям x и x+width-1 соответственно, а верхняя и нижняя – y и y+height-1 соответственно. Таким образом, чтобы зарисовать все пикселы компонента, необходимо передать следующие аргументы:

```
g.fillRect(0, 0, getWidth(), getHeight());
```

drawOval(int x, int y, int width, int height)

Этот метод рисует овал, вписанный в прямоугольник, задаваемый указанными параметрами. Очевидно, что если прямоугольник имеет равные стороны (т.е. является квадратом), овал становится окружностью.

Снова для того, чтобы вписать овал в границы компонента, необходимо вычесть по единице из ширины и высоты:

```
g.drawRect(0, 0, getWidth()-1, getHeight()-1);
g.drawOval(0, 0, getWidth()-1, getHeight()-1);
```

fillOval(int x, int y, int width, int height)

Этот метод закрашивает указанный овал.

drawArc(int x, int y, int width, int height, int startAngle, int arcAngle)

Этот метод рисует дугу – часть овала, задаваемого первыми четырьмя параметрами. Дуга начинается с угла startAngle и имеет угловой размер arcAngle. Начальный угол

соответствует направлению часовой стрелки, указывающей на 3 часа. Угловой размер отсчитывается против часовой стрелки. Таким образом, размер в 90 градусов соответствует дуге в четверть овала (верхнюю правую). Углы "растянуты" в соответствии с размером прямоугольника. В результате, например, угловой размер в 45 градусов всегда задает границу дуги по линии, проходящей из центра прямоугольника в его правый верхний угол.

fillArc(int x, int y, int width, int height, int startAngle, int arcAngle)

Этот метод закрашивает сектор, ограниченный дугой, задаваемой параметрами.

drawString(String text, int x, int y)

Этот метод выводит на экран текст, задаваемый первым параметром. Точка (x, y) задает позицию самого левого символа.

Состояние Graphics

Экземпляр класса Graphics хранит параметры, необходимые для отрисовки. Рассмотрим их по порядку.

Цвет

Очевидно, что для отрисовки линий, овалов, текста и т.д. необходимо использовать тот или иной цвет. По умолчанию он задается свойством foreground компонента. В любой момент его можно изменить с помощью метода setColor.

Узнать текущее значение цвета для отрисовки можно с помощью метода getColor.

Шрифт

Метод drawString не имеет аргумента, задающего шрифт для вывода текста на экран. Этот параметр также является частью состояния Graphics. Его значение по умолчанию задается соответствующим свойством компонента, однако может быть изменено с помощью метода setFont. Для получения текущего значения служит метод getFont.

Clip (ограничитель)

Хотя методы класса Graphics могут принимать любые значения аргументов, задающих значения координат (в пределах типа int), существует дополнительный ограничитель – clip. Любые изменения вне этого ограничителя на экране появляться не будут. Например, если вызвать метод drawLine(-100, -100, 1000, 1000), то на компоненте отобразится лишь часть линии, которая помещается в его границы.

Размеры ограничителя можно изменять. Метод clipRect(int x, int y, int width, int height) вычисляет пересечение указанного прямоугольника и текущей области clip. Результат станет новым ограничителем. Таким образом, этот метод может только сужать область clip. Метод setClip(int x, int y, int width, int height) устанавливает ограничитель произвольно в форме прямоугольника. Метод getClipBounds возвращает текущее значение в виде объекта Rectangle.

При появлении приложения на экране каждый видимый компонент должен быть отрисован полностью. Поэтому при первом вызове метода paint, как правило, область clip совпадает с границами компонента. Однако при дальнейшей работе это не всегда так.

Методы repaint и update

Кроме paint в классе Component объявлены еще два метода, отвечающие за прорисовку компонента. Как было рассмотрено, вызов paint инициируется операционной системой, если возникает необходимость перерисовать окно приложения, или часть его. Однако может потребоваться обновить внешний вид, руководствуясь программной логикой. Например, отобразить результат операции вычисления, или работы с сетью. Можно изменить состояние компонента (значение его полей), но операционная система не отследит такое изменение и не инициирует перерисовку.

Для программной инициализации перерисовки компонента служит метод repaint. Конечно, у него нет аргумента типа Graphics, поскольку программист не должен создавать экземпляры этого класса (точнее, его наследников, ведь Graphics – абстрактный класс). Метод repaint можно вызывать без аргументов. В этом случае компонент будет перерисован максимально быстро. Можно указать аргумент типа long – количество миллисекунд. Система инициализирует перерисовку спустя указанное время. Можно указать четыре числа типа int (x, y, width, height), задавая прямоугольную область компонента, которая нуждается в перерисовке. Наконец, можно указать все 5 параметров – и задержку по времени, и область перерисовки.

Если перерисовка инициируется приложением, то система вызывает не метод paint, а метод update. У него уже есть аргумент типа Graphics и по умолчанию он лишь закрашивает всю область компонента фоновым цветом (свойство background), а затем вызывает метод paint. Зачем же было вводить этот дополнительный метод, если можно было сразу вызвать paint? Дело в том, что поскольку перерисовка инициируется приложением, для сложных компонентов становится возможной некоторая оптимизация обновления внешнего вида. Например, если изменение заключается в появлении нового графического элемента, то можно избежать повторной перерисовки остальных элементов – переопределить метод update и реализовать в нем отображение одного только нового элемента. Если же компонент имеет простую структуру, можно оставить метод update без изменений.

Прорисовка контейнера

Теперь, когда известно, как работает прорисовка компонента, перейдем к рассмотрению контейнера. Для его корректного отображения необходимо выполнить два действия. Во-первых, нарисовать сам контейнер, ведь он является наследником компоненты, а значит, имеет метод paint, который может быть переопределен для задания особенного внешнего вида такого контейнера. Во-вторых, инициировать отрисовку всех компонентов, вложенных в него.

Первый шаг ничем не отличается от прорисовки обычного компонента. Как правило, контейнер не содержит никаких особых элементов отображения, ведь основную его площадь занимают вложенные компоненты. Поэтому перейдет ко второму шагу.

Если контейнер не пустой, значит, в нем есть одна или несколько компонент. Они будут отрисованы последовательно в том порядке, в каком были добавлены. Однако недостаточно просто в цикле вызвать метод paint для каждого компонента.

Во-первых, если компонента невидима (свойство visible выставлено в false), то, очевидно, метод paint у нее вызываться не должен.

Во-вторых, центр координат компонента находится в левом верхнем углу его контейнера, а у контейнера – в левом верхнем углу его контейнера. Таким образом, при переходе от отрисовки контейнера к отрисовке лежащего в нем компонента необходимо изменить (перенести) центр системы координат.

Затем необходимо установить clip в соответствии с размером очередного компонента. Необходимо выставить значения по умолчанию для цвета и шрифта, тем более что предыдущий компонент мог изменить их непредсказуемым образом.

В итоге получается более удобным создать новый экземпляр Graphics для каждого компонента. Для этого существует метод create, который порождает копию Graphics, причем ему можно передать аргументы (int x, int y, int width, int height). В результате у нового Graphics будет смещен центр координат в точку (x, y), а clip-область будет получена пересечением существующего ограничителя с прямоугольником (0, 0, width, height) (в новых координатах). Метод create создает копию без изменения этих параметров.

Такие копии бывает удобно порождать и в рамках одного вызова метода paint, если в нем описан слишком сложный алгоритм. После использования такого объекта Graphics его необходимо особым образом освобождать – вызовом метода dispose(). Если необходимо только сместить точку отсчета координат, можно использовать метод translate (int x, int y).

Таким образом, контейнер своим методом paint отрисовывает себя и все вложенные в него компоненты. Если какие-то из них, в свою очередь, являются контейнерами, то процесс иерархически продолжается вглубь. В итоге весь AWT интерфейс, каким бы сложным он ни был, состоит из дерева контейнеров и компонент, отрисовка которых начинается с самого верхнего контейнера и по ветвям развивается вглубь до каждого видимого компонента.

Отдельный интерес представляет этот самый верхний контейнер. Как правило, это окно операционной системы, одновременно являющееся контейнером для Java-компонент. Именно операционная система инициализирует процесс отрисовки, отвечает за сворачивание и разворачивание окна, изменение его размера и так далее. Со стороны Java для работы с окном используется класс Window, который является наследником Container и рассматривается ниже.

Наследники класса Component

Теперь, когда рассмотрены основные принципы работы классов Component и Container, рассмотрим их наследников, с помощью которых и строится функциональный пользовательский интерфейс.

Начнем с наследников класса Component.

Класс Canvas

Класс Canvas является простейшим наследником Component. Он не добавляет никакой новой функциональности, но именно его нужно использовать в качестве суперкласса для создания пользовательского компонента с некоторым нестандартным внешним видом.

Класс Label

Как понятно из названия, этот компонент отображает надпись. Соответственно, и его основной конструктор принимает один аргумент типа String – текст надписи. С помощью стандартных свойств класса Component – шрифт, цвет, фоновый цвет – можно менять вид надписи. Текст можно сменить и после создания Label с помощью метода setText.

Обратите внимание, что при этом компонент сам обновляет свой вид на экране. Такой особенностью обладают все стандартные компоненты AWT.

Класс Button

Этот компонент позволяет добавить в интерфейс стандартные кнопки. Основной конструктор принимает в качестве аргумента String – надпись на кнопке. Как обрабатывать нажатие на кнопку и другие пользовательские события, рассматривается ниже.

Классы Checkbox и CheckboxGroup

Компонент Checkbox имеет два способа применения.

Когда он используется сам по себе, он представляет checkbox – элемент, который может быть выделен или нет (например, нужна доставка для оформляемой покупки или нет). В этом случае в конструктор передается лишь текст – подпись к checkbox.

Ниже приведён пример, в котором в теле контейнера добавляется два checkbox:

```
Checkbox payment = new
        Checkbox("Оплата в кредит");
payment.setBounds(10, 10, 120, 20);
add(payment);
Checkbox delivery = new Checkbox("Доставка");
delivery.setBounds(10, 30, 120, 20);
add(delivery);
```

Обратите внимание, что размер Checkbox должен быть достаточным для размещения не только поля для "галочки", но и для подписи.

Второй способ применения компонент Checkbox предназначен для организации "переключателей" (radio buttons). В этом случае несколько экземпляров объединяются в группу, причем лишь один из переключателей может быть выбран. В роли такой группы выступает класс CheckboxGroup. Он не является визуальным, то есть никак не отображается на экране. Его задача – логически объединить несколько Checkbox. Группу, к которой принадлежит переключатель, можно указывать в конструкторе:

```
CheckboxGroup delivery = new CheckboxGroup();
Checkbox fast = new Checkbox(
      "Срочная (1 день)", delivery, false);
fast.setBounds(10, 10, 150, 20);
add(fast);
Checkbox normal = new Checkbox(
      "Обычная (1 неделя)", delivery, true);
normal.setBounds(10, 30, 150, 20);
add(normal);
Checkbox postal = new Checkbox(
      "По почте (до 1 месяца)",
       delivery, false);
postal.setBounds(10, 50, 150, 20);
add(postal);
```

В примере при вызове конструктора класса Checkbox помимо текста подписи и группы, указывается состояние переключателя (булевский параметр).

Классы Choice и List

Компонент Choice служит для выбора пользователем одного из нескольких возможных вариантов (выпадающий список). Рассмотрим пример:

```
Choice color = new Choice();
color.add("Белый");
color.add("Зеленый");
color.add("Синий");
color.add("Черный");
add(color);
```

В обычном состоянии компонент отображает только выбранный вариант. В процессе выбора отображается весь набор вариантов.

Обратите внимание, что для компонента Choice всегда есть выбранный элемент.

Компонент List, подобно Choice, предоставляет пользователю возможность выбирать варианты из списка предложенных. Отличие заключается в том, что List отображает сразу несколько вариантов. Количество задается в конструкторе:

```
List accessories = new List(3);
accessories.add("Чехол");
accessories.add("Наушники");
accessories.add("Аккумулятор");
accessories.add("Блок питания");
add(accessories);
```

В списке находится 4 варианта. Однако в конструктор был передан параметр 3, поэтому только 3 из них будут видны на экране. С помощью полосы прокрутки можно выбрать остальные варианты.

Еще одно свойство List – возможность выбрать сразу несколько из предложенных вариантов. Для этого надо либо в конструкторе вторым параметром передать булевское значение true (false соответствует выбору только одного элемента), либо воспользоваться методом setMultipleMode.

Классы TextComponent, TextField, TextArea

Класс TextComponent является наследником Component и базовым классом для компонент, работающих с текстом,– TextField и TextArea.

TextField позволяет вводить и редактировать одну строку текста. Различные методы позволяют управлять содержимым этого поля ввода:

```
TextField tf = new TextField();
tf.setText("Enter your name");
tf.selectAll();
add(tf);
```

В коде вторая строка устанавливает значение текста в поле ввода (метод getText позволяет получить текущее значение). Затем весь текст выделяется (есть методы, позволяющие выделить часть текста).

Для любой текстовой компоненты можно задать особый режим. В базовом классе Component определено свойство enabled, которое, если выставлено в false, блокирует все пользовательские события. Для текстовой компоненты вводится новое свойство – editable (можно редактировать), методы для работы с ним – isEditable и setEditable. Если текст нельзя редактировать, но компонент доступен, то пользователь может выделить часть, или весь текст, и, например, скопировать его в буфер.

TextField обладает еще одним свойством. Все хорошо знакомы с полем ввода для пароля – вводимые символы не отображаются, вместо них появляется один и тот же символ. Для TextField его можно установить с помощью метода setEchoChar (например, setEchoChar(‘*’)).

TextArea позволяет вводить и просматривать многострочный текст. В конструктор передается количество строк и столбцов, которые определяют размер компонента (вычисляется на основе средней ширины символа). Эти параметры не ограничивают длину вводимого текста – при необходимости появляются полосы прокрутки:

Класс Scrollbar

Класс Scrollbar позволяет работать с полосами прокрутки, которые используются для перемещения внутренней области от начальной до конечной позиции. Полоса может быть расположена горизонтально или вертикально. Стрелки на каждом из ее концов служат для перемещения "на один шаг" в соответствующем направлении. "Взявшись" курсором мыши за бегунок, можно переместить его в любую позицию. С помощью кликов мыши по полосе прокрутки, но вне положения бегунка, можно делать перемещение "на страницу" вверх или вниз. Все эти действия хорошо знакомы по многим пользовательским интерфейсам, например, Windows. Они полностью поддерживаются компонентом Scrollbar.

Конструктор позволяет задавать ориентацию полосы прокрутки — для этого предусмотрены константы VERTICAL и HORIZONTAL. Кроме того, с помощью конструктора можно задать начальное положение бегунка, размер "страницы", а также минимальное и максимальное значения, в пределах которых линейка прокрутки может изменять параметр. Для получения и установки текущего состояния полосы прокрутки используются методы getValue и setValue. Ниже приведен пример, в котором создается и вертикальный, и горизонтальный Scrollbar.

```
int height = getHeight(), width = getWidth();
int thickness = 16;
Scrollbar hs = new Scrollbar(
            Scrollbar.HORIZONTAL, 50,
            width/10, 0, 100);
Scrollbar vs = new Scrollbar(
            Scrollbar.VERTICAL, 50,
            height/2, 0, 100);
add(hs); add(vs);
hs.setBounds(0, height - thickness,
          width - thickness, thickness);
vs.setBounds(width - thickness, 0, thickness,
          height - thickness);
```

Наследники класса Container

Теперь перейдем к рассмотрению стандартных контейнеров AWT.

Класс Panel

Подобно тому, как Canvas служит базовым классом для создания своих компонент с особым внешним видом, класс Panel является суперклассом для новых контейнеров с особой работой с вложенными компонентами. Впрочем, поскольку Panel, в отличие от Container, класс не абстрактный, его можно использовать для иерархической организации сложного пользовательского интерфейса, группируя компоненты в такие простейшие контейнеры.

Класс ScrollPane

Выше был рассмотрен компонент Scrollbar, предназначенный для полосы прокрутки. Если стоит задача, например, показать пользователю график некоторой функции с возможностью просмотра для изучения различных областей, необходимо создать две полосы прокрутки, правильно их установить и в дальнейшем обрабатывать все действия пользователя, вычислять новое положение видимой области, перерисовывать график и т.д.

В большинстве случаев все эти задачи может взять на себя контейнер ScrollPane. Этот контейнер обладает рядом особенностей. Во-первых, в него можно поместить лишь одну компоненту – при добавлении новой старая удаляется. Во-вторых, отличается работа с вложенным компонентом, чьи границы выходят за границы самого контейнера. Как мы рассматривали раньше, "выступающие" области никогда не будут отображены на экране. В контейнере ScrollPane в этом случае появляются полосы прокрутки (горизонтальная или

вертикальная), с помощью которых можно промотать видимую область и таким образом увидеть весь компонент полностью. При этом не нужно предпринимать никаких дополнительных действий – надо лишь добавить компонент в ScrollPane.

Может вызвать удивление, почему разрешается добавление лишь одного компонента. А если нужно проматывать более сложную конструкцию? Здесь и проявляется польза класса Panel. Все элементы собираются в этот простейший контейнер, который, в свою очередь, добавляется в ScrollPane.

Конструктор этого класса может принимать параметр, задающий логику появления полос прокрутки – они могут быть видимы всегда, появляться по мере необходимости, либо не появляться никогда.

Класс Window

Из опыта работы с оконными графическими интерфейсами современных операционных систем мы привыкли к тому, что каждое приложение обладает одним или несколькими окнами. Класс Window служит базовым классом для всех окон, порождаемых из Java. Разумеется, он также является интерфейсом к соответствующему окну операционной системы, которая обслуживает окна всех приложений.

Как правило, используется один из двух наследников Window – классы Frame и Dialog, которые будут рассмотрены следующими. Однако экземпляры Window не обладают ни рамкой, ни кнопками закрытия или минимизации окна, а потому зачастую используются как заставки (так называемые splash screen).

Конструктор Window требует в качестве аргумента ссылку на Window или Frame. Другими словами, базовые окна не являются самостоятельными, они привязываются к другим окнам.

Классы Frame и Dialog

Класс Frame предназначен для создания полнофункциональных окон приложений – с полосой заголовка, рамкой, кнопками закрытия, минимизации и максимизации окна. Поскольку Frame, как правило, является главным окном приложения, он создается невидимым, чтобы можно было настроить все его параметры, добавить все вложенные контейнеры и компоненты и лишь затем отобразить его в подготовленном виде. Конструктор принимает текстовый параметр – заголовок фрейма.

Рассмотрим пример организации работы с фреймом, который отображает компонент:

```
public class TestCanvas extends Canvas {
  public void paint(Graphics g) {
    g.drawLine(0, 0, getWidth(), getHeight());
    g.drawLine(0, getHeight(),getWidth(), 0);
  }

  public static void main(String arg[]) {
    Frame f = new Frame("Test frame");
    f.setSize(400, 300);
    f.add(new TestCanvas());
```

```
    f.setVisible(true);
  }
}
```

Если класс Frame предназначен для создания основного окна приложения, то экземпляры класса Dialog позволяют открывать дополнительные окна для взаимодействия с пользователем. Это может потребоваться, например, для вывода критического сообщения, для ввода параметров и т.д.. Окно диалога обладает стандартным оформлением – полоса заголовка, рамка. В правой части полосы заголовка присутствует лишь одна кнопка – закрытия окна.

Поскольку Dialog является несамостоятельным окном, в конструктор необходимо передать ссылку на родительский фрейм или окно другого диалога. Также можно задать заголовок окна. Как и Frame, диалоговое окно создается изначально невидимым.

Важным свойством диалогового окна является модальность. Если диалог модальный, то при его появлении на экране блокируются все пользовательские события, приходящие в родительское окно такого диалога.

Класс FileDialog

Класс FileDialog является модальным диалогом (наследником Dialog) и позволяет легко организовать работу с файлами. Этот класс предназначен и для открытия файла (open file), и для сохранения (save file). Окно диалога имеет внешний вид, принятый для текущей операционной системы.

Конструктор принимает в качестве параметров ссылку на родительский фрейм, заголовок окна и режим работы. Для задания режима в классе определены две константы – LOAD и SAVE.

После создания диалога FileDialog его необходимо сделать видимым. Затем пользователь делает свой выбор. После закрытия диалога результат можно узнать с помощью методов getDirectory (для получения полного имени каталога) и getFile (для получения имени файла). Если пользователь нажал кнопку "Отмена" ("Cancel"), то будут возвращены значения null.

Обработка пользовательских событий

Весь предыдущий раздел "Дерево компонентов" был посвящен заданию внешнего вида пользовательского интерфейса. Однако до сих пор он был статическим. Перейдем теперь к рассмотрению правил обработки различных событий, которые могут возникать как результат действий пользователя, и не только.

Модель обработки событий построена на основе стандартного шаблона проектирования ООП Observer/Observable. В качестве наблюдаемого объекта выступает тот или иной компонент AWT. Для него можно задать один или несколько классов-наблюдателей. В AWT они называются слушателями (listener) и описываются специальными интерфейсами, название которых оканчивается на слово Listener. Когда с наблюдаемым объектом что-то происходит, создается объект "событие" (event), который

"посылается" всем слушателям. Так слушатель узнает, например, о действии пользователя и может на него отреагировать.

Каждое событие является подклассом класса java.util.EventObject. События пакета AWT, которые и рассматриваются в данной лекции, являются подклассами java.awt.AWTEvent. Для удобства классы различных событий и интерфейсы слушателей помещены в отдельный пакет java.awt.event.

Прежде, чем углубляться в особенности событий, рассмотрим, как они применяются на практике, на примере простейшего события – ActionEvent.

Событие ActionEvent

Рассмотрим появление события ActionEvent на примере нажатия на кнопку.

Предположим, в нашем приложении создается кнопка сохранения файла:

```
Button save = new Button("Save");
add(save);
```

Теперь, когда окно приложения с этой кнопкой появится на экране, пользователь сможет нажать ее. В результате AWT сгенерирует ActionEvent. Чтобы получить и обработать его, необходимо зарегистрировать слушателя. Название нужного интерфейса прямо следует из названия события – ActionListener. В нем всего один метод (в некоторых слушателях их несколько), который имеет один аргумент – ActionEvent.

Объявим класс, который реализует этот интерфейс:

```
class SaveButtonListener
    implements ActionListener {
  private Frame parent;
  public SaveButtonListener(Frame parentFrame)
  {
    parent = parentFrame;
  }
  public void actionPerformed(ActionEvent e)
  {
    FileDialog fd = new FileDialog(parent,
            "Save file", FileDialog.SAVE);
    fd.setVisible(true);
    System.out.println(fd.getDirectory()+"/"+
                    fd.getFile());
  }
}
```

Конструктор класса требует в качестве параметра ссылку на родительский фрейм, без которого не удастся создать FileDialog. В методе actionPerformed класса ActionListener описываются действия, которые необходимо предпринять по нажатию пользователем на кнопку. А именно, открывается файловый диалог, с помощью которого определяется путь сохранения файла. Для нашего примера достаточно вывести этот путь на консоль.

Следующий шаг – регистрация слушателя. Название соответствующего метода снова прямо следует из названия интерфейса – addActionListener.

```
save.addActionListener(
    new SaveButtonListener(frame));
```

Все необходимое для обработки нажатия пользователем на кнопку сделано. Ниже приведен полный листинг программы:

```
import java.awt.*;
import java.awt.event.*;

public class Test {
  public static void main(String args[]) {
    Frame frame = new Frame("Test Action");
    frame.setSize(400, 300);
    Panel p = new Panel();
    frame.add(p);
    Button save = new Button("Save");
    save.addActionListener(
        new SaveButtonListener(frame));
    p.add(save);

    frame.setVisible(true);
   }
}

class SaveButtonListener
    implements ActionListener {
  private Frame parent;
  public SaveButtonListener(Frame parentFrame)
  {
    parent = parentFrame;
  }
  public void actionPerformed(ActionEvent e)
  {
    FileDialog fd = new FileDialog(parent,
      "Save file", FileDialog.SAVE);
    fd.setVisible(true);
    System.out.println(fd.getDirectory()+
                    fd.getFile());
  }
}
```

После запуска программы появится фрейм с одной кнопкой "Save". Если нажать на нее, откроется файловый диалог. После выбора файла на консоли отображается полный путь к нему.

События AWT

Итак, для каждого события AWT определен класс XXEvent, интерфейс XXListener, а в компоненте-источнике событий – метод для регистрации слушателя addXXListener.

Совсем не обязательно, чтобы одно событие могло порождаться лишь одним компонентом как результат какого-то одного действия пользователя. Например, рассмотренный ActionEvent генерируется после нажатия на кнопку (Button), после нажатия клавиши Enter в поле ввода текста (TextField), при двойном щелчке мыши по элементу списка (List) и т.д. Узнать, какие события генерирует тот или иной компонент, можно по наличию методов addXXListener.

Многие слушатели, в отличие от ActionListener, имеют более одного метода для различных видов событий. Например, MouseMotionListener наблюдает за движением мыши и имеет два метода – mouseMoved (обычное движение) и mouseDragged (перемещение с нажатой кнопкой мыши). Иногда бывает необходимо работать лишь с одним методом, остальные приходится объявлять и оставлять пустыми. Чтобы избежать этой бесполезной работы, в пакете java.awt.event объявлены вспомогательные классы-адаптеры, например, MouseMotionAdapter (название прямо следует из названия слушателя). Эти классы наследуются от Object и реализуют соответствующий интерфейс. Адаптер – абстрактный класс, но абстрактных методов в нем нет, они все объявлены пустыми. От такого класса можно наследоваться и переопределить только те методы, которые нужны для приложения.

Классы сообщений (event) содержат вспомогательную информацию для обработки события. Метод getSource() возвращает объект-источник события. Конкретные наследники AWTEvent могут иметь дополнительные методы. Например, MouseEvent сообщает о нажатии кнопки мыши, а его методы getX и getY возвращают координаты точки, где это событие произошло.

Наряду с методом addXXListener важную роль играет removeXXListener. Поскольку в Java ненужные объекты удаляются из памяти автоматическим сборщиком мусора, который подсчитывает ссылки на объекты, важно следить за тем, чтобы не оставалось ссылок на ненужные объекты. Если слушатель уже выполнил свою роль и более не нужен, то явно в программе может не остаться ссылок на него, однако компонент будет хранить его в своем списке слушателей. Чтобы дать сработать garbage collector, необходимо воспользоваться методом removeXXListener.

Рассмотрим обзорно все события AWT и соответствующих им слушателей, определенных в Java начиная с версии 1.1.

MouseMotionListener и MouseEvent

Это событие рассматривалось выше в примере. Оно отвечает за перемещение курсора мыши. Соответствующий слушатель имеет два метода – mouseMoved для обычного перемещения и mouseDragged для перемещения с нажатой кнопкой мыши. Обратите внимание, что этот слушатель работает не с событием MouseMotionEvent (такого класса нет), а с MouseEvent, как и MouseListener.

MouseListener и MouseEvent

Этот слушатель имеет методы mouseEntered и mouseExited. Первый вызывается, когда курсор мыши появляется над компонентом, а второй – когда выходит из его границ.

Для обработки нажатий кнопки мыши служат три метода: mousePressed, mouseReleased и mouseClicked. Если пользователь нажал, а затем отпустил кнопку, то слушатель получит все три события в указанном порядке. Если щелчков было несколько, то метод getClickCount класса MouseEvent вернет количество. Как уже указывалось, методы getX и getY возвращают координаты точки, где произошло событие. Чтобы определить, какая кнопка мыши была нажата, нужно воспользоваться методом getModifiers и сравнить результат с константами:

```
(event.getModifiers() &
  MouseEvent.BUTTON1_MASK)!=0
```

Как правило, первая кнопка соответствует левой кнопке мыши.

KeyListener и KeyEvent

Этот слушатель отслеживает нажатие клавиш клавиатуры и имеет три метода: keyTyped, keyPressed, keyReleased. Первый отвечает за ввод очередного Unicode-символа с клавиатуры. Метод keyPressed сигнализирует о нажатии, а keyReleased – об отпускании некоторой клавиши. Взаимосвязь между этими событиями может быть нетривиальной. Например, если пользователь нажмет и будет удерживать клавишу Shift и в это время нажмет клавишу "A", произойдет одно событие типа keyTyped и несколько keyPressed/Released. Если пользователь нажмет и будет удерживать, например, пробел, то после первого keyPressed будет многократно вызван метод keyTyped, а после отпускания – keyReleased.

В классе KeyEvent определено множество констант, которые позволяют точно идентифицировать, какая клавиша была нажата и в каком состоянии находились служебные клавиши (Ctrl, Alt, Shift и так далее).

FocusListener и FocusEvent

В каждом приложении один из компонентов обладает фокусом и может получать события от клавиатуры. Фокус можно переместить, например, щелкнув мышкой по другому компоненту, либо нажав клавишу Tab.

Интерфейс FocusListener содержит два метода – focusGained и focusLost (получен/потерян).

TextListener и TextEvent

Компоненты-наследники TextComponent отвечают за ввод текста и порождают TextEvent. Слушатель имеет один метод textValueChanged. С его помощью можно отслеживать каждое изменение текста, чтобы, например, выдавать пользователю подсказку, основываясь на первых введенных символах.

ItemListener и ItemEvent

Это событие могут генерировать такие классы, как Checkbox, Choice, List. Слушатель имеет один метод itemStateChanged, который сигнализирует об изменении состояния элементов.

AdjustmentListener и AdjustmentEvent

Это событие генерируется компонентом ScrollBar. Слушатель имеет один метод adjustmentValueChanged, сигнализирующий об изменении состояния полосы прокрутки.

WindowListener и WindowEvent

Это событие сигнализирует об изменении состояния окна (класс Window и его наследники).

Рассмотрим особо один из методов слушателя – windowClosing. Этот метод вызывается, когда пользователь предпринимает попытку закрыть окно, например, нажимая на соответствующую кнопку в заголовке окна. Мы видели из примеров ранее, что в Java окна при этом не закрываются. Дело в том, что AWT лишь посылает WindowEvent в ответ на такое действие, а инициировать закрытие окна должен программист:

```
public class WindowClosingAdapter
      extends WindowAdapter {
   public void windowClosing(WindowEvent e)
   {
     ((Window)e.getSource()).dispose();
   }
}
```

Объявленный адаптер в методе windowClosing получает ссылку на окно, от которого пришло событие. Обычно мы пользовались методом setVisible(false), чтобы сделать компонент невидимым. Но поскольку Window автоматически порождает окно операционной системы, существует специальный метод dispose, который освобождает все системные ресурсы, связанные с этим окном.

Когда окно будет закрыто, у слушателя вызывается еще один метод – windowClosed.

ComponentListener и ComponentEvent

Это событие отражает изменение основных параметров компонента – положение, размер, свойство visible.

ContainerListener и ContainerEvent

Это событие позволяет отслеживать изменение списка содержащихся в этом контейнере компонент.

С развитием Java в AWT появляются и другие события, например, позволяющие поддерживать колесико мыши. Однако все они работают по точно такой же схеме, а потому их можно легко освоить самостоятельно.

Обработка событий с помощью внутренних классов

В теле класса можно объявлять внутренние классы. До сих пор такая возможность не была востребована в наших примерах, однако обработка событий AWT – как раз удобный случай рассмотреть такие классы на примере анонимных классов.

Предположим, в приложение добавляется кнопка, которой следует добавить слушателя. Зачастую бывает удобно описать логику действий в отдельном методе того же класса. Если вводить слушателя, как делалось раньше – в отдельном классе, то это сразу порождает ряд неудобств: появляется новый, малосодержательный класс, которому к тому же необходимо передать ссылку на исходный класс и так далее.

Гораздо удобнее поступить следующим образом:

```
Button b = new Button();
b.addActionListener(new ActionListener())
{
  public void actionPerformed(ActionEvent e)
  {
    processButton();
  }
};
```

Рассмотрим подробно, что происходит в этом примере. Сначала создается кнопка, у которой затем вызывается метод addActionListener. Обратим внимание на аргумент этого метода. Может сложится впечатление, что производится попытка создать экземпляр интерфейса (new ActionListener()), однако это невозможно. Дело меняет фигурная скобка, которая указывает, что порождается экземпляр нового класса, объявление которого последует за этой скобкой. Класс наследуется от Object и реализует интерфейс ActionListener. Ему необходимо реализовать метод actionPeroformed, что и делается. Обратите внимание на еще одну важную деталь – в этом методе вызывается processButton. Это метод, который мы планировали разместить во внешнем классе. Таким образом, внутренний класс может напрямую обращаться к методам внешнего класса.

Такой класс называется анонимным, он не имеет своего имени. Однако правило, согласно которому компилятор всегда создает .class-файл для каждого класса Java, действует и здесь. Если внешний класс называется Test, то после компиляции появится файл Test$1.class.

Задание на работу

Создать класс Starter в пакете rootPackage. В методе main класса Starter создать объект класса Frame, на нём разместить контейнер (нечётные варианты работают с типом Panel, чётные – со ScrollPane). В контейнер поместить объект типа Label и один компонент в соответствии с номером варианта (номер варианта – остаток от деления Вашего номера в списке на 7).

0 Button
1 Checkbox
2 Radiogroup (Checkbox and CheckboxGroup)

3 Choice
4 List
5 TextArea
6 TextField

Объявить отдельный класс для обработки события, относящегося к этому компоненту. Обработчик события должен записывать в текстовое поле метки состояние компонента: кнопка нажата/не нажата, или кнопка нажата n раз; пункт выбран/не выбран, или имя выбранного (выбранных) пунктов (для списковых компонентов); текст, введённый в компонент (для наследников TextComponent).

Реализовать выход из программы по нажатию кнопки закрытия главного окна. Завершить программу можно с помощью метода exit класса System: System.exit(0);

Отчёт по работе

Отчёт по работе должен содержать название работы, цель работы, структурированный текст программы, вывод по работе.