# 1. Работа с изображениями

В этой главе рассматривается AWT-класс image и пакет java.awt.image. Вместе они поддерживают работу с изображениями (отображение и манипуляции с графическими изображениями). Под изображением понимают прямоугольный графический объект. Изображения являются ключевым компонентом Web-дизайна. Включение тега <img> в браузер Mosaic NCSA (National Center for Supercomputer Applications, Национальный Центр Суперкомпьютерных Приложений) привело к началу взрывного роста Web в 1993 г. Этот тег был использован, чтобы встраивать изображение в поток гипертекста. Java расширяет данную базовую концепцию, допуская программное управление изображениями. Java обеспечивает интенсивную поддержку работы с изображениями.

Изображения — это объекты класса image, который является частью пакета java.awt. Для манипулирования изображениями используются классы пакета java.awt.image, который содержит большое количество классов и интерфейсов изображений. Здесь рассмотрим следующие классы java.awt.image:

- CropImageFilter
- FilteredImageSource
- ImageFilter
- MemoryImageSource
- PixelGrabber
- RGBImageFilter

Будут использоваться интерфейсы:

- ImageConsumer
- ImageObserver ,
- ImageProducer

Кроме того, рассматривается класс MediaTracker, который является частью пакета java.awt.

## Форматы графических файлов

Первоначально, Web-изображения могли быть только в формате GIF. Формат растровых изображений GIF (Graphics Interchange Format, формат обмена графическими данными) был создан в CompuServe Incorporation в 1987 г., для возможности просмотра встроенных изображений, что хорошо подходило для Internet. Каждое GIF-изображение может иметь не больше 256 цветов. Это ограничение заставило главных поставщиков браузеров в 1995 г. добавить поддержку изображений в формате JPEG. Формат JPEG (Joint Photographic Expert Group) был создан группой фотографических экспертов для хранения изображений с полным цветовым спектром и непрерывным тоном. Эти изображения, если они созданы должным образом, могут иметь намного более высокую точность

цветовоспроизведения и более высокую степень сжатия по сравнению с GIF-кодированием. В большинстве случаев вас не будет даже интересовать, какой формат вы используете в своих программах. В языке Java все различия в кодировании изображений скрыты за ясными и удобными интерфейсами их классов.

## 1.1. Создание, загрузка и просмотр изображений

Существует три общие операции, которые используются для работы с любыми изображениями: создание, загрузка и просмотр изображения на экране. Класс `image` языка Java имеет средства для создания нового `объекта изображения` и его загрузки, и средства, с помощью которых изображение можно отобразить на экране. Отметим также, что `Image` обслуживает как изображения, находящиеся в памяти, так и изображения, которые загружаются из внешних источников.

### Создание объекта изображения

Чтобы изображения стали видимыми, их нужно *рисовать* в окне.. Однако класс `Image` не имеет достаточной информации для того, чтобы создать надлежащий формат данных для экрана. Поэтому класс `Component` (из пакета `java.awt`) содержит специальный "производственный" (factory) метод с именем `createImage()`, который используется для создания `image`-объектов. (Напомним, что все AWT-компоненты являются подклассами `Component`, поэтому все они поддерживают данный метод.) Метод `createImage()` имеет две формы:

```
Image createImage (ImageProducer imgProd)
Image createImage(int width, int height)
```

Первая форма возвращает изображение, изготовленное параметром `imgProd`, который является объектом класса, реализующего интерфейс `ImageProducer` (производителей изображений мы рассмотрим позже). Вторая форма возвращает пустое изображение, которое имеет указанную ширину и высоту. Например:

```
Canvas c = new Canvas () ;
Image test = c.createImage (200, 100);
```

Здесь создается экземпляр (объект) класса `Canvas` и затем вызывается производственный метод `createImage()`, чтобы фактически построить объект типа `Image`. В этом случае изображение будет пустым. Позже мы рассмотрим, как записать в него данные.

### Загрузка изображения

Другой способ получения изображения — его загрузка. Для этого используется метод getImage(), определенный классом Applet. Он имеет следующие формы:

```
Image getImage(URL url)
Image getImage(URL url, String ImageName)
```

Первая версия возвращает image-объект, который инкапсулирует изображение, найденное по (универсальному) адресу, указанному в параметре url. Вторая версия возвращает image-объект, который инкапсулирует изображение, найденное по адресу, указанному в url, и имеющему имя, указанное в ImageName.

## 1.2. Просмотр изображений

Имея изображение, вы можете выводить его (на экран), используя метод drawImage(), который является членом класса Graphics. Он содержит несколько форм. Мы будем использовать метод в следующей форме:

```
boolean drawImage(Image imgObj, int left, int top, ImageObserver imgOb)
```

Он выводит изображение, переданное ему параметром imgObj, размещая его левый верхний угол с позиции, указанной в left и top. imgOb – ссылка на класс, который реализует интерфейс ImageObserver. Этот интерфейс реализуется всеми AWT-компонентами. Наблюдатель изображения (image observer) — это объект, который может контролировать изображение, пока оно загружается. Класс ImageObserver описан в следующем разделе.

С помощью getImage() и drawImage() действительно очень просто загружать и просматривать изображение. Ниже показан пример апплета, который загружает и выводит одиночное изображение. Загружается файл seattle.jpg, но вы можете заменить его любым файлом в формате GIF или JPG (только удостоверьтесь, что он находится в одном каталоге с HTML-файлом, который содержит апплет).

### Программа 132. Загрузка изображения

```
// Файл SimpleImageLoad.java
/*
 <applet code="SimpleImageLoad" width = 248 height = 146>
 <param name="img" value="Picture.jpg">
 </applet>
*/
import java.awt.*;
import java.applet.*;
public class SimpleImageLoad extends Applet {
```

```
    Image img;          // Ссылка на изображение
    public void init()   {
       img = getImage(getDocumentBase(), getParameter("img"));
    }
    public void paint(Graphics g) {
       g.drawImage(img, 0, 0, this);
    }
}
```

Для загрузки данного апплета следует подготовить html-файл следующего содержания:

```
<applet code="SimpleImageLoad" width = 248 height = 146>
 <param name="img" value="Picture.jpg">
 </applet>
```

Этог файл должен быть расположен в той же папке, что и откомпилированный class - файл апплета. Там же должен быть файл с картинкой `Picture.jpg`.

В методе `init()` переменной `img` назначается изображение, возвращенное методом `getImage()`. Метод `getImage()` использует строку, возвращенную методом `getParameter("img")`, как имя файла изображения. Это изображение загружается из URL-адреса, в который метод `getDocumentBase()` возвратил URL-адрес HTML-страницы с тегом данного апплета. Имя файла, возвращенное методом `getParameter("img")`, исходит из тега `<param name = "img" value = "seattle.jpg">` данного апплета. Этот тег является эквивалентом, правда немного более медленным, HTML-тега `<img src = "seattie.jpg" width = 248 height = 146>`. Результат выполнения этой программы показан на рис. 1.

Когда этот апплет выполняется, он начинает загрузку `img` в методе `init()`. На экране можно видеть изображение по мере его загрузки из сети, потому что реализация интерфейса `ImageObserver` в классе `Applet` вызывает метод `paint()` каждый раз, когда прибывает следующая порция данных изображения.

Наблюдение загрузки изображения довольно информативно, но было бы лучше, если бы вы использовали время загрузки изображения, чтобы что-то делать параллельно. Полностью сформированное изображение появляется на экране только в тот момент, когда оно целиком загружено. Для контроля загрузки изображения во время прорисовок экрана с другой информацией можно использовать интерфейс `ImageObserver`, описанный далее.

## Интерфейс ImageObserver

`ImageObserver` — это интерфейс, используемый для приема уведомлений о том, как генерируются изображения.

ImageObserver определяет только один метод: ImageUpdate(). Использование наблюдателя изображения позволяет выполнять (параллельно с загрузкой изображения) другие действия, такие как показ индикатора хода работы (progress-индикатора) или дополнительного экрана, которые информируют вас о ходе загрузки. Подобный вид уведомления очень полезен, когда изображение загружается по сети, где проектировщик содержимого редко принимает во внимание, что люди часто пробуют загружать апплеты через медленный модем.

Метод ImageUpdate() имеет следующую общую форму:

```
boolean imageUpdate(Image imgObj, int flags, int left, int top, int width,
int height)
```

Здесь imgObj — загружаемое изображение, а flags – целое число, которое сообщает состояние отчета обновления. Четыре целых параметра left, top, width и height представляют прямоугольник, который содержит различные значения в зависимости от передаваемых в flags-значений. imageUpdate() должен возвратить false, если он завершил загрузку, и true, если еще имеется остаток изображения для обработки.

Параметр flags содержит один или несколько разрядных флажков, определенных как статические переменные внутри интерфейса ImageObserver. Эти флажки и информация, которую они обеспечивают, перечислены в табл. 16.1.

**Таблица 16.1. Разрядные флажки параметра flags метода imageUpdate()**

| Флажок | Значение |
|---|---|
| WIDTH | Параметр width правилен и содержит ширину изображения |
| HEIGHT | Параметр height правилен и содержит высоту изображения |
| PROPERTIES | Свойства, связанные с изображением могут теперь быть получены через imgObj.getProperty() |
| SOMEBITS | Получена следующая порция пикселов, необходимых для вывода изображения. Параметры left, top, width, и height определяют прямоугольник, содержащий новые пикселы |
| FRAMEBITS | Получен полный фрейм, являющийся частью многофреймового изображения, которое было предварительно нарисовано. Данный фрейм может быть отображен. Параметры left, top, width и height не используются |
| ALLBITS | Изображение выведено целиком. Параметры left, top, width и height не используются |
| ERROR | Произошла ошибка с изображением, которое |

| | |
|---|---|
| | прослеживалось асинхронно. Изображение неполно и не может быть отображено. Никакая дальнейшая видеоинформация не будет получена. Для удобства будет также установлен флажок ABORT, чтобы указать, что производство изображения было прервано |
| ABORT | Изображение, которое прослеживалось асинхронно, было прервано прежде, чем оно было закончено. Однако, если ошибка не произошла, доступ к любой части данных изображения перезапустит |

Класс Applet имеет реализацию метода imageUpdate() интерфейса ImageObserver, который используется для перерисовки изображений во время их загрузки. Его можно переопределить в вашем классе, чтобы изменить поведение метода.

Простой пример метода imageUpdate():

```
public boolean imageUpdate (Image img, int flags,
                            int x, int y, int w, int h) {
   if ((flags & ALLBITS) == 0) {
      System.out.println("Загрузка изображения...");
      return true;
   }
   else {
      Applet imageupdate ImageObserver, imageupdate ():
      System.out.println("Загрузка изображения завершена.");
      return false;
   }
}
```

На рис. 1 показана работа апплета с отображенной картинкой.

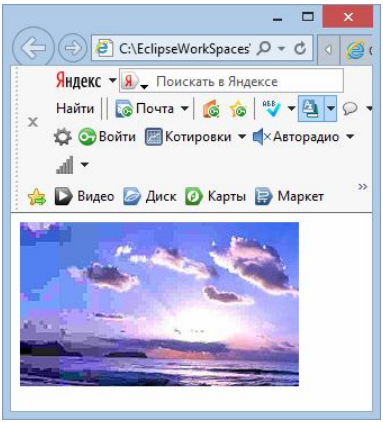


Рис. 1. Отображение рисунка в окне апплета

## Пример с ImageObserver

Теперь рассмотрим пример, который переопределяет imageUpdate() так, чтобы избавить версию апплета SimpleImageLoad от частого мерцания изображения. Умалчиваемая реализация imageUpdate() в классе Applet имеет несколько проблем. Во-первых, она перерисовывает полное изображение каждый раз, когда прибывают какие-нибудь новые данные. Это вызывает вспышки между цветом фона и изображением. Во-вторых, он использует свойство Applet.repaint(), заставляющее систему перерисовывать изображение только каждую десятую долю секунды или около этого. Это вызывает чувство толчкообразного, негладкого движения изображения при его рисовании. Наконец, заданная по умолчанию реализация не знает ничего относительно изображений, которые могут быть не в состоянии загружаться должным образом. Многие начинающие программисты Java недовольны тем, что метод getImage() преуспевает даже тогда, когда указанное изображение не существует. Вы не узнаете об отсутствии изображения, пока не начнет работу imageUpdate(). Если вы используете умалчиваемую реализацию imageUpdate(), то никогда и не узнаете, что случилось. Ваш метод paint() просто не будет ничего делать, когда вы вызываете g.drawImage().

Следующий ниже пример фиксирует все три этих проблемы в десяти строках кода. Во-первых, он устраняет мерцание с помощью двух небольших изменений. Он переопределяет метод update() так, чтобы тот вызвал paint() без первоначальной прорисовки цвета фона. Фон устанавливается через метод setBackground() в init(), так что начальный фон рисуется только однажды. Он также использует версию метода repaint(), которая указывает прямоугольник для рисования. Система установит область отсечения такой, чтобы ничего вне данного прямоугольника не рисовалось. Это устраняет мерцание перерисовки и улучшает исполнение.

Во-вторых, это избавит от толчкообразного, негладкого показа входящего изображения путем перерисовки каждый раз, когда repaint() принимает обновление. Эти обновления происходят на базе построчного сканирования, так что изображение, которое имеет 100 пикселов в высоту, будет "перерисовываться" 100 раз за время загрузки. Обратите внимание, что это не самый быстрый способ отображать изображение, а лишь самый сглаженный.

Наконец, он обрабатывает ошибку, вызванную тем, что нужный файл не найден, исследуя ABORT-разряд параметра flags. Если он установлен, переменная экземпляра error принимает значение true, и затем вызывается repaint (). Метод paint() модифицируется так, что,

если переменная error равна true, то сообщение об ошибках печатается на ярко-красном фоне.

## Программа 133. Наблюдение загрузки изображения

```
// Файл ObservedImageLoad.java
/*
<applet code = "ObservedImageLoad" width = 248 height = 146>
   <param name = "img" value = "seattle.jpg">
</applet>
*/
import java.awt.*;
import java.applet.*;
public class ObservedImageLoad extends Applet {
   Image img;
   boolean error = false;
   String imgname;
   public void init() {
      setBackground(Color.blue);
      imgname = getParameter("img");
      img = getImage(getDocumentBase(), imgname);
   }
   public void paint(Graphics g) {
      if (error) {
         Dimension d = getSize();
         g.setColor(Color.red);
         g.fillRect(0, 0, d.width, d.height);
         g.setColor(Color.black);
         g.drawString("Image not found: " + imgname, 10, d.height/2);
      }
      else {
         g.drawImage (img, 0, 0, this);
      }
   }
   public void update(Graphics g) {
      paint(g);
   }
   public boolean imageUpdate(Image img, int flags,
                              int x, int y, int w, int h) {
      if ((flags & SOMEBITS) != 0) {
      // Новые частичные данные
         repaint(x,y, w, h); // Рисует новые пикселы
      }
      else if ((flags & ABORT) != 0) {
         error = true;            // Файл не найден
         repaint();               // Рисовать весь апплет
      }
      return (flags & (ALLBITS | ABORT)) == 0;
   }
}
```

Рис. 2 показывает экраны двух отдельных прогонов этого апплета. Левый экран показывает загруженное изображение, а правый —

отображает сообщение о том, что в теге апплета имя файла было напечатано с ошибкой.

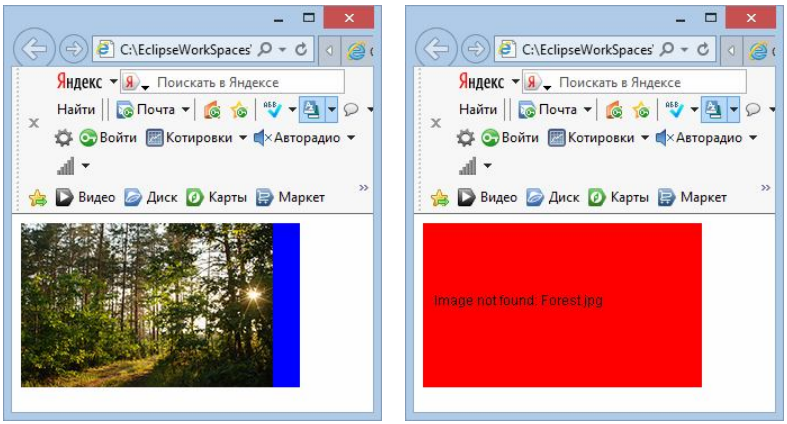


Рис. 2. Удачная загрузка рисунка и файл с рисунком не найден

Ниже показан интересный вариант `imageUpdate()`, который вы могли бы попробовать испытать. Он ждет, пока изображение не будет полностью загружено перед моментальной вставкой его в экран единственной перерисовкой.

```
public boolean imageUpdate(Image img,  int flags,
                           int x,  int y,  int w,  int h) {
   if ((flags & ALLBITS) != 0) {
      repaint();
   }
   else if ((flags &  (ABORT | ERROR)) != 0) {
      error = true;         // file not found
      repaint() ;
   }
   return (flags & (ALLBITS | ABORT | ERROR)) == 0;
}
```

## Двойная буферизация

Мало того, что изображения полезны для хранения картинок, но вы можете также использовать их для рисования поверхностей вне экрана. Это позволяет вам передавать любое изображение, включая текст и графику, во внеэкранный буфер, который вы можете отображать в более позднее время. Преимущество данного механизма состоит в том, что изображение становится видимым только тогда, когда оно уже окончательно построено. Рисование сложного изображения может занимать несколько миллисекунд или больше, что может наблюдаться пользователем как мигание или мерцание. Это мигание отвлекает

пользователя и заставляет его чувствовать вашу визуализацию как более медленную, чем она есть в действительности. Использование внеэкранного изображения для ослабления мерцания называется двойной буферизацией, из-за того что экран рассматривается как буфер для пикселов, а внеэкранное изображение — это второй буфер, где можно готовить пикселы для показа на экране.

Ранее было показано, как можно создать пустой image-объект. Теперь покажем, как можно рисовать на данном изображении, а не на экране. Для этого необходим объект типа Graphics, чтобы использовать любой из его визуализующих методов. Удобно также, что Graphics-объект, который можно использовать для рисования на изображении, доступен через метод getGraphics(). Ниже представлен фрагмент кода, который создает новое изображение, получает его графический контекст, и заполняет полное изображение красными пикселами:

```
Canvas c = new Canvas ();
Image test = c.createImage(200/ 100);
Graphics gc = test.getGraphics();
gc.setColor(Color.red);
gc.fillRect(0, 0, 200,  100);
```

Как только создано и заполнено внеэкранное изображение, оно еще не будет видимым. Для его окончательного отображения вызывается drawImage(). Чтобы продемонстрировать влияние двойной буферизации на время рисования, ниже приведен пример, рисующий изображение со значительным временем прорисовки:

**Программа 134. Двойная буферизация**

```
/*
 <applet code = DoubleBuffer width = 250 height = 250>
 </applet>
*/
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
public class DoubleBuffer extends Applet {
   int gap = 3;
   int mx, my;
   boolean flicker = true;
   Image buffer = null;
   int w, h;
   public void init() {
      Dimension d = getSize();
      w = d.width;
      h = d.height;
      buffer = createImage(w, h);
      addMouseMotionListener(new MouseMotionAdapter() {
         public void mouseDragged(MouseEvent me) {
            mx = me.getX();
```

```
            my = me.getY();
            flicker = false;
            repaint();
         }
         public void mouseMoved(MouseEvent me) {
            mx = me.getX();
            my = me.getY();
            flicker = true;
            repaint();
         }
      });
   }
   public void paint(Graphics g) {
      Graphics screengc = null;
      if (!flicker) {
         screengc = g;
         g = buffer.getGraphics();
      }
      g.setColor(Color.blue);
      g.fillRect(0, 0, w, h);
      g.setColor(Color.red);
      for (int i = 0; i < w; i += gap)
         g.drawLine(i, 0, w - i, h);
      for (int i = 0; i < h; i += gap)
         g.drawLine(0, i, w, h - i);
      g.setColor(Color.black);
      g.drawString("Press mouse button to double buffer", 10, h/2);
      g.setColor(Color.yellow);
      g.fillOval(mx - gap, my - gap, gap * 2 + 1, gap * 2 + 1);
      if (!flicker) {
         screengc.drawImage(buffer, 0, 0, null);
      }
   }
   public void update(Graphics g) {
      paint (g);
   }
}
```

Предложенный простой апплет имеет усложненный метод `paint()`. Он заполняет фон синим и затем рисует сверху красный муаровый образец. В верхней его части выводится некоторый черный текст, и затем рисуется желтый круг с центром в (`mx`, `my`)-координатах. Методы `mouseMoved()` и `mouseDragged()` переопределены так, чтобы отслеживать позицию мыши.

Эти методы идентичны, за исключением установок булевой переменной `flicker`. `mouseMoved()` устанавливает `flicker` в `true`, а `mouseDragged()` — в `false`. Это приводит к вызову `repaint()` с установкой `flicker` в `true`, когда мышь передвигается (но никакая кнопка не нажата), и установкой ее в `false`, когда мышь перетаскивается с любой нажатой кнопкой.

Когда вызывается paint() с переменной flicker, установленной в true, мы видим на экране выполнение каждой операции рисования. В случае, когда кнопка мыши нажата, и paint() вызывается с flicker, установленной в false, мы видим совершенно иную картину. Метод paint() обменивает Graphics-ссылку g с графическим контекстом, который отсылает к внеэкранному полотну buffer, который мы создали в init(). Тогда все операции рисования становятся невидимыми. В конце paint(), мы просто вызываем drawImage(), чтобы показать результаты этих методов рисования все сразу.

Обратите внимание, что удобно передавать в drawImage() в качестве четвертого параметра null. Этот параметр используется для передачи объекта типа ImageObserver, который принимает уведомление об image-событиях. Так как это изображение не производится из сетевого потока, нам не нужно уведомление.

Левый снимок на рис. 3 показывает, как выглядит апплет с ненажатыми кнопками мыши. Снимок был сделан, когда изображение было где-то в середине перерисовки. Правый снимок показывает, что, когда кнопка мыши нажимается, изображение всегда имеет законченный, четкий вид (благодаря двойной буферизации).

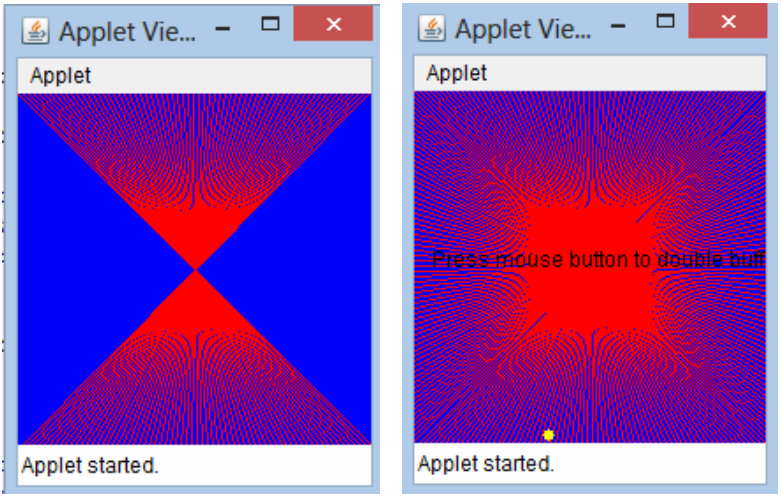


Рис. 3. Рисунок без буферизации (слева) и с двойной буферизацией (справа)

## Класс MediaTracker

Многие из ранних разработчиков Java находили интерфейс ImageObserver слишком сложным для понимания и управления загрузкой множественных изображений. Сообщество разработчиков просило

более простое решение, которое позволило бы программистам загружать все их изображения синхронно, не беспокоясь относительно imageUpdate(). В ответ на это Sun Microsystems в последующем выпуске JDK добавила к java.awt класс с именем MediaTracker. MediaTracker создает объект, который будет параллельно проверять состояние произвольного числа изображений.

Для использования MediaTracker вы создаете его новый экземпляр и применяете его метод addImage(), чтобы прослеживать состояние загрузки изображения. Общий формат addimage():

```
void addImage (Image imgObj, int imgID)
void addImage(Image imgObj, int imgID, int width, int height)
```

Здесь imgObj — прослеживаемое изображение. Его идентификационный номер передается в imgID. ID (идентификатор) номера не должны быть уникальными. Вы можете использовать тот же номер с несколькими изображениями, как средство их идентификации в качестве части группы. Во второй форме параметры width и height определяют размеры отображаемого объекта.

Как только вы зарегистрировали изображение, можете проверить, загружено ли оно, или можете ждать, когда оно полностью загрузится. Для проверки состояния изображения вызывают метод checkID(), одна из версий, которого, имеет формат:

```
boolean checkID(int imgID)
```

Здесь imgID определяет ID изображения, которое вы хотите проверить. Метод возвращает true, если все изображения, которые имеют указанный идентификатор, были загружены (или, когда загрузка закончилась с ошибкой или пользователь выполнил аварийное завершение работы). Иначе он возвращает false. Чтобы увидеть, были ли все прослеживаемые изображения загружены, можно использовать метод checkAll().

MediaTracker следует применять при загрузке группы изображений. Пока все изображения не разгружены, вы можете развлечь пользователя, отобразив на экране что-нибудь еще (пока прибывают загружаемые изображения).

Далее следует пример, который загружает слайд-показ с семью изображениями и отображает прогресс-полоску процесса загрузки:

**Программа 135. Показ слайдов**

```
/*
<applet code = "TrackedImageLoad" width = 300 height = 400>
<param name="img"
   value = "vincent + leonardo + matisse + picasso
             + renoir + seurat + vermeer">
```

```
</applet>
*/
import java.util.*;
import java.applet.*;
import java.awt.*;
public class TrackedImageLoad extends Applet
 implements Runnable {
   MediaTracker tracker;
   int tracked;
   int frame_rate = 5;
   int current_img = 0;
   Thread motor;
   static final int MAXIMAGES = 10;
   Image img[] = new Image[MAXIMAGES];
   String name[] = new String[MAXIMAGES];
   boolean stopFlag;
   public void init() {
      tracker = new MediaTracker(this);
      StringTokenizer st =
          new StringTokenizer(getParameter("img"), "+");
      while (st.hasMoreTokens() && tracked <= MAXIMAGES) {
         name[tracked] = st.nextToken();
         img[tracked] = getImage(getDocumentBase(),
                                       name[tracked] + ".jpg");
         tracker.addImage(img[tracked], tracked);
         tracked++;
      }
   }
   public void paint(Graphics g) {
      String loaded = "";
      int donecount = 0;
      for(int i = 0; i < tracked; i++) {
         if (tracker.checkID(i, true)) {
            donecount++;
            loaded += name[i] + " ";
         }
      }
      Dimension d = getSize();
      int w = d.width;
      int h = d.height;
      if (donecount == tracked) {
         frame_rate = 1;
         Image i = img[current_img++];
         int iw = i.getWidth(null);
         int ih = i.getHeight(null);
         g.drawImage(i, (w - iw)/2, (h - ih)/2, null);
         if (current_img >= tracked)
            current_img = 0;
      } else {
         int x = w * donecount;        // отслежено
         g.setColor(Color.black);
         g.fillRect(0, h/3, x, 16);
         g.setColor(Color.white);
         g.fillRect(x, h/3, w - x, 16);
         g.setColor(Color.black);
```

```
            g.drawString(loaded, 10, h/2);
        }
    }
    public void start() {
        motor = new Thread(this);
        stopFlag = false;
        motor.start() ;
    }
    public void stop() {
        stopFlag = true;
    }
    public void run() {
        motor.setPriority(Thread.MIN_PRIORITY);
        while (true) {
            repaint();
            try {
                Thread.sleep(1000 / frame_rate);
            }
            catch (InterruptedException e)
            { }
            if(stopFlag)
                return;
        }
    }
}
```

Для запуска данного апплета создадим следующий html-файл:

```
<applet code = "TrackedImageLoad" width = 300 height = 400>
<param name = "img"
 value = "vincent+leonardo+matisse+picasso+renoir+seurat+vermeer">
</applet>
```

В папке с class - файлом должны быть файлы с картинками, имене которых указаны в html-файле: `vincent.jpg`, `leonardo.jpg`, `matisse.jpg`, `picasso.jpg`, `renoir.jpg`, `seurat.jpg`, `vermeer.jpg`.

Представленный пример создает новый `MediaTracker` в методе `init()` и затем добавляет каждое из именованных прослеживаемых изображений с помощью метода `addImage()`. В методе `paint()` на очередном из прослеживаемых изображений вызывается `checkID()`. Если все изображения загружены, они отображаются. Если нет, показывается простая прогресс-полоска числа загруженных изображений с именами полностью загруженных (отображаемых ниже этой полоски). На рис.4 показано окно апплета с одним из портретов.

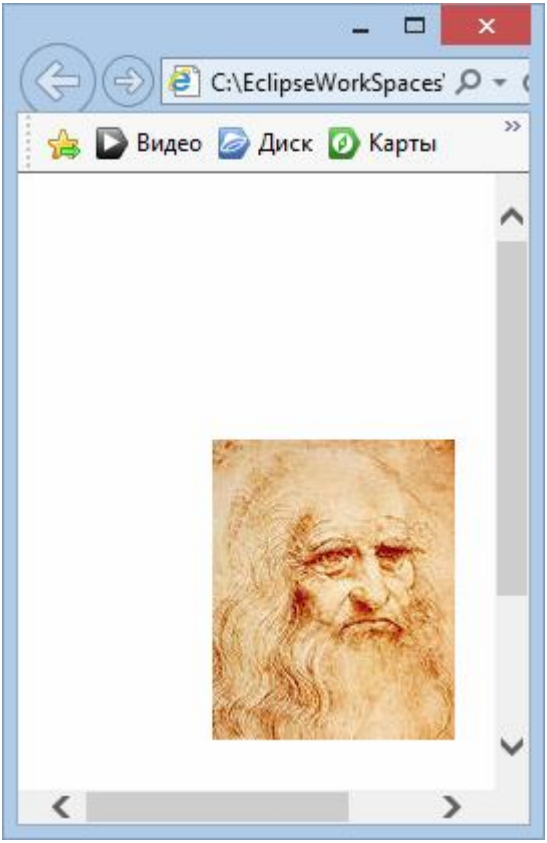


Рис. 4. Один из слайдов, показываемых апплетом

## Интерфейс ImageProducer

`ImageProducer` — это интерфейс для объектов, которые хотят производить данные для изображений. Объект, реализуюший интерфейс `ImageProducer`, поставляет целочисленные или байтовые массивы, которые представляют данные изображения и производят `Image`-объекты. Как вы видели ранее, одна из форм метода `createImage()` имеет объект `ImageProducer` в качестве своего параметра. Существуют два *производителя изображений* (image producers), содержащихся в `java.awt.image`: `MemoryImageSource` и `FilteredImageSource`. Здесь мы рассмотрим `MemoryImageSource` и создадим новый `Image`- объект из данных, генерируемых внутри апплета.

## Производитель изображений MemoryImageSource

MemorylmageSource — это класс, который создает новый Image-объект ИЗ массива данных. Он определяет несколько конструкторов. Тот, который мы будем использовать, имеет следующую сигнатуру:

```
MemorylmageSource (int width, int height, int pixel [ ], int offset, int
scanLineWidth) ,
```

Объект MemoryImageSource создается из массива целых чисел (в формате умалчиваемой цветовой модели RGB), указанного в параметре pixel (он-то и содержит данные для воспроизведения Image-объекта). В умалчиваемой цветовой модели пиксел — это целое число формата OxAARRGGBB, где A - Alpha, R - Red, G - Green, и B - Blue.

Значение Alpha представляет степень прозрачности пиксела (0 — полностью прозрачный, 255 — полностью непрозрачный). Ширина и высота результирующего изображения передается в параметрах width и height. Исходную точку для начала чтения данных в массиве пикселов задает параметр offset. Ширина строки сканирования (которая часто совпадает с шириной изображения) задает параметр scanLinewidth.

Следующий короткий пример генерирует MemoryImageSource-объект, используя разновидность простого алгоритма (поразрядное исключающее ИЛИ (x, y)-координат каждого пиксела).

**Программа 136. Преобразование целого массива в изображение**

```
// Файл MemoryImageGenerator.java
/*
<applet code = "MemoryImageGenerator" width = 256 height = 256>
</applet>
*/
import java.applet.*;
import java.awt.*;
import java.awt.image.*;
public class MemoryImageGenerator extends Applet {
   Image img;
   public void init() {
      Dimension d = getSize();
      int w = d.width;
      int h = d.height;
      int pixels[] = new int[w * h];
      int i = 0;
      for(int y = 0; y < h; y++) {
         for(int x = 0; x < w; x++) {
            int r = (x ^ y) & 0xff;
            int g = (x * 2 ^ y * 2) & 0xff;
            int b = (x * 4 ^ y * 4) & 0xff;
            pixels[i++] = (255 << 24) | (r << 16) | (g << 8) | b;
         }
      }
```

```
        img = createImage(new MemoryImageSource(w, h, pixels, 0, w) );
    }
    public void paint(Graphics g) {
        g.drawImage(img, 0, 0, this);
    }
}
```

Данные для нового `MemoryImageSource` создаются в методе `init()`. Массив целых предназначен для хранения пикселных значений; данные генерируются во вложенных `for`-циклах, где значения `r`, `g` и `b` организуют сдвиги в пикселах массива `pixels`. Наконец, вызывается метод `createImage()` с новым экземпляром `MemoryImageSource`, созданным из необработанных пикселных данных, в качестве последнего аргумента. Рис. 5 показывает изображение после выполнения апплета.

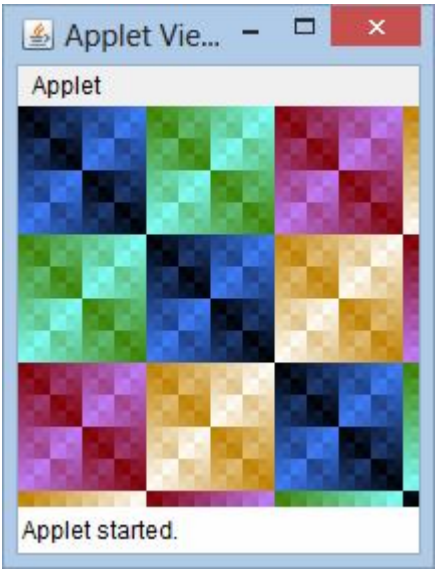


Рис. 5. Изображение, созданное апплетом

## Интерфейс ImageConsumer

`imageConsumer` — это абстрактный интерфейс для объектов, которые хотят получать пикселные данные изображений (скажем, от производителя) и поставлять их (скажем, на экран) уже как другой вид данных. Очевидно, что этот интерфейс является противоположностью интерфейса `ImageProducer`.

Объект, который реализует интерфейс `ImageConsumer`, собирается создавать массивы `int` или `byte`, которые представляют пикселы `Image-`

объекта. Рассмотрим класс PixelGrabber, который является простой реализацией интерфейса ImageConsumer.

## Класс PixelGrabber

Класс PixelGrabber определен в java.lang.image. Это инверсия класса MemoryImageSource. Вместо построения изображения из массива пикселных значений, он берет существующее изображение и строит из него массив пикселов. Для использования PixelGrabber вы сначала организуете int-массив достаточно большой, чтобы содержать данные пикселов, и затем создаете экземпляр PixelGrabber, передавая ему прямоугольную область, которую вы хотите преобразовать в пикселное представление. Наконец, вы вызываете метод grabPixels() этого экземпляра.

Конструктор PixelGrabber, который используется в этой главе, имеет следующую форму:

```
PixelGrabber (Image imgObj, int left, int top, int width, int height,
              int pixel[ ], int offset, int scanbineWxdth)
```

Здесь imgObj – объект, чьи пикселы преобразуются. Значения left и top определяют левый верхний угол прямоугольника, а width и height — его размеры, из которого будут получены пикселы. Пикселы будут сохранены в массиве pixel, со смещением offset. Ширину строки сканирования (которая часто такая же, как ширина изображения) задают в scanLineWidth.

Метод grabPixels() определяется с такими сигнатурами:

```
boolean grabPixels() throws InterruptedException
boolean grabPixels(long milliseconds) throws InterruptedException
```

Оба метода возвращают true, если завершаются успешно, и false — в противном случае. Во второй форме параметр milliseconds определяет, как долго метод будет ожидать пикселы.

Далее показан пример, который "захватывает" пикселы изображения и затем создает гистограмму их яркости. Под *гистограммой яркости* здесь понимается распределение количества пикселов с определенной яркостью по всем значениям шкалы яркости (от 0 до 255). После того как апплет рисует изображение, он выводит (поверх этого изображения) гистограмму его яркости (рис. 23.6).

**Программа 137. Гистограмма яркости изображения**

```
// Файл HistoGrab.java
/*
<applet code = HistoGrab.class width = 341 height = 400>
<param name = img value = Gerl.jpg>
</applet>
```

```
*/
import java.applet.*;
import java.awt.*;
import java.awt.image. *;
public class HistoGrab extends Applet {
   Dimension d;
   Image img;
   int iw, ih;
   int pixels[];
   int w, h;
   int hist[] = new int [256];
   int max_hist = 0;
   public void init() {
      d = getSize();
      w = d.width;
      h = d.height;
      try {
         img = getImage(getDocumentBase(), getParameter ("img"));
         MediaTracker t = new MediaTracker(this);
         t.addImage(img, 0);
         t.waitForID(0);
         iw = img.getWidth(null);
         ih = img.getHeight(null);
         pixels = new int[iw * ih];
         PixelGrabber pg = new PixelGrabber(img, 0, 0, iw, ih,
                                            pixels, 0, iw);
         pg.grabPixels();
      }
      catch (InterruptedException e) { };
      for (int i = 0; i < iw * ih; i++) {
         int p = pixels[i];
         int r = 0xff & (p >> 16);
         int g = 0xff & (p >> 8);
         int b = 0xff & (p);
         int y = (int) (.33 * r + .56 * g + .11 * b);
         hist[y]++;
      }
      for (int i = 0; i < 256; i++) {
         if (hist[i] > max_hist)
            max_hist = hist[i];
      }
   }
   public void update()  {}
   public void paint(Graphics g) {
      g.drawImage(img, 0, 0, null);
      int x = (w - 256) / 2;
      int lasty = h - h * hist[0] / max_hist;
      for (int i = 0; i < 256; i++, x++) {
         int y = h - h * hist[i] / max_hist;
         g.setColor(new Color(i, i, i));
         g.fillRect(x, y, 1, h);
         g.setColor(Color.red);
         g.drawLine (x - 1, lasty, x, y);
         lasty = y;
      }
```

```
    }
}
```

На рис. 6 показан пример гистограммы яркости изображения.

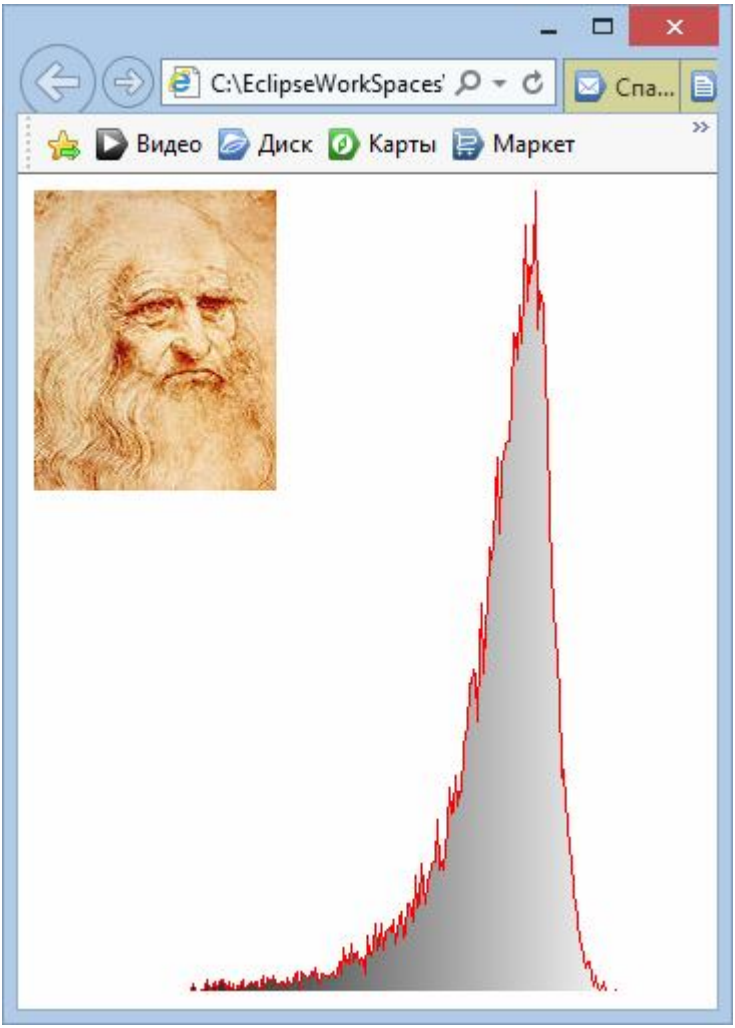


Рис. 6. Гистограмма яркости изображения

## Класс ImageFilter

На базе интерфейсов ImageProducer и ImageConsumer (и их конкретных классов MemoryImageSource И PixelGrabber), можно создать произвольный набор преобразующих фильтров, каждый из которых берет источник пикселов, модифицирует сгенерированные ими пикселы и передает произвольному потребителю. Этот механизм аналогичен способу, с помощью которого создаются конкретные абстрактные классы ввода/вывода InputStream, OutputStream, Reader и Writer. Такая *поточная модель изображений* завершается введением класса ImageFilter. Класс ImageFilter пакета java.awt.Image имеет несколько подклассов — AreaAveragingScaleFilter, CropImageFilter, ReplicateScaleFilter и RGBImageFilter. Кроме того, в пакете имеется специальная реализация интерфейса ImageProducer с именем FilteredImageSource. Объекты класса FilteredImageSource производят фильтрованные пикселы изображений, генерируемых объектами класса ImageProducer. Для фильтрации в них используются объекты одного из указанных выше подклассов ImageFilter. Экземпляр класса FilteredImageSource передается (в качестве аргумента) в вызов метода CreateImage(), который используется в качестве производителя фильтрованных (преобразованных) изображений. Мы рассмотрим два специальных фильтра: CropImageFilter и RGBImageFilter.

### Фильтр CropImageFilter

Фильтр CropImageFilter просто вырезает из исходного изображения небольшую прямоугольную область. Его удобно использовать, например, когда нужно работать не с целым большим изображением, а с более мелкими его частями. Если каждое подизображение имеет один и тот же размер, то можно легко извлекать их, используя для разборки блока изображения фильтр CropImageFilter. Далее показан пример, в котором одиночное изображение делится фильтром на шестнадцать одинаковых прямоугольных частей и затем эти неперекрывающиеся изображения скрэмблируются (перемешиваются случайным образом) путем 32-кратного обмена пар, выбираемых случайно из шестнадцати ырезанных изображений (на рис. 7 — картина Кузьмы Сергеевича етрова-Водкина (1878-1939) "Купание красного коня", скрэмблированная апплетом TileImage).

**Программа 138. Разбиение изображения на части**

```
// Файл TileImage.java
/*
<applet code = TileImage.class width = 288 height = 399>
   <param name = img value = picasso.jpg>
```

```
</applet>
*/
import java.applet.*; import java.awt.*;
import java.awt.image.*;
public class TileImage extends Applet {
   Image img;
   Image cell[] = new Image[4 * 4];
   int iw, ih;
   int tw, th;
   public void init() {
      try {
         img = getImage(getDocumentBase(), getParameter("img"));
         MediaTracker t = new MediaTracker(this);
         t.addImage(img, 0);
         t.waitForID(0);
         iw = img.getWidth(null);
         ih = img.getHeight(null);
         tw = iw / 4;
         th = ih / 4;
         CropImageFilter f;
         FilteredImageSource fis;
         t = new MediaTracker(this);
         for (int y = 0; y < 4; y++) {
            for (int x = 0; x < 4; x++) {
               f = new CropImageFilter(tw * x, th * y, tw, th);
               fis = new FilteredImageSource(img.getSource(), f);
               int i = y * 4 + x;
               cell[i] = createImage(fis);
               t.addImage(cell[i], i);
            }
         }
         t.waitForAll();
         for (int i = 0; i < 32; i++) {
            int si = (int)(Math.random() * 16);
            int di = (int)(Math.random() * 16);
            Image tmp = cell[si];
            cell[si] = cell[di];
            cell[di] = tmp;
         }
      }
      catch (InterruptedException e) { };
   }
   public void update(Graphics g) {
      paint(g);
   }
   public void paint(Graphics g) {
      for (int y = 0; y < 4; y++) {
         for (int x = 0; x < 4; x++) {
            g.drawImage(cell[y * 4 + x], x * tw, y * th, null);
         }
      }
   }
}
```

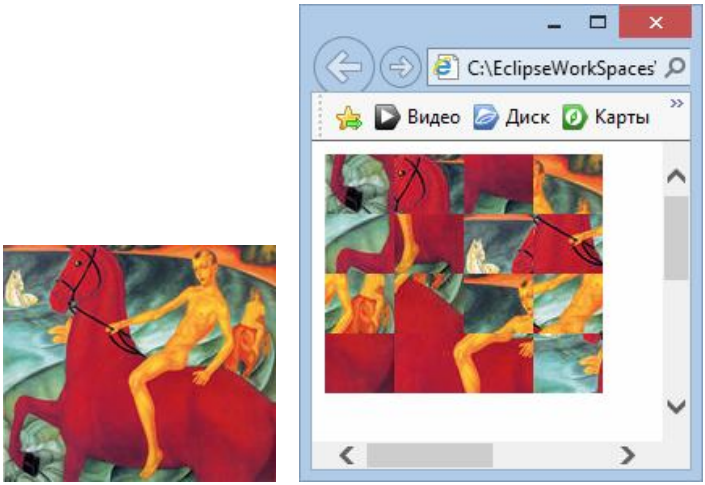


Рис. 7. Разбиение изображения на 16 частей и их перемешивание

## Фильтр RGBImageFilter

Фильтр `RGBImageFilter` используется для попикселного преобразования одного изображения в другое, трансформируя цвет пикселов. Данный фильтр можно использовать для прояснения изображения, увеличения его контраста, или даже для преобразования цветного изображения к полутоновому.

Пример, демонстрирующий `RGBImageFilter`, использует динамическую встроенную стратегию для обрабатывающих изображения фильтров. Программа включает интерфейс для обобщенной фильтрации изображения так, чтобы апплет мог просто загружать эти фильтры, основываясь на HTML-тегах `<param>`, без необходимости знать подробности обо всех фильтрах изображений. Этот пример состоит из главного апплет-класса с именем `ImageFilterDemo`, интерфейса с именем `PlugInFilter` и сервисного класса с именем `LoadedImage`, который инкапсулирует некоторые из уже использовавшихся методов класса `MediaTracker`. Кроме того, в программу включены три фильтра — `Grayscale`, `Invert` и `Contrast`, которые просто манипулируют цветовым пространством исходного изображения, используя `RGBImageFilters`, и еще два класса — `Blur` и `Sharpen`, которые применяют более сложные фильтры "свертывания", изменяющие данные пиксела, основываясь на пикселах, окружающих каждый пиксел исходных данных. `Blur` и `Sharpen` — это подклассы абстрактного класса-помощника называемого `Convolver`. Рассмотрим каждую часть нашего примера.

## ImageFilterDemo.java

Класс ImageFilterDemo является каркасом апплета для других фильтров изображений. Он использует простой менеджер компоновки BorderLayout, с панелью в позиции South, содержащей кнопки, которые будут представлять каждый фильтр. Объект Label занимает слот North для информационных сообщений о ходе работы фильтра. Изображение (которое инкапсулировано в Canvas подклассе LoadedImage, описанном позже) размещается в слоте Center. Мы анализируем кнопки фильтров вне параметра filters тега <param> (где они отделены значками +) — с помощью класса StringTokenizer.

Метод ActionPerformed() интересен тем, что он использует метку кнопки как имя класса фильтра, который он пробует загрузить с помощью метода newInstance():

```
pif = (PlugInFilter) Class.forName(a).newInstance();
```

Данный метод устойчив и выбирает адекватное действие, даже если кнопка не соответствует подходящему классу, реализующему PluglnFilter.

**Программа 139. Фильтрация изображения**

```
// Файл ImageFilterDemo.java
/*
<applet code = ImageFilterDemo width = 350 height = 450>
   <param name = img value = "brullov.jpg">
   <param name = filters value = "Grayscale+Invert+Contrast+Blur+Sharpen">
</applet>
*/
import java.applet.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
public class ImageFilterDemo extends Applet implements ActionListener {
   Image img;
   PlugInFilter pif;
   Image fimg;
   Image curImg;
   LoadedImage lim;
   Label lab;
   Button reset;
   public void init() {
      setLayout(new BorderLayout ());
      Panel p = new Panel();
      add(p, BorderLayout.SOUTH);
      reset = new Button("Reset");
      reset.addActionListener(this);
      p.add(reset);
      StringTokenizer st =
         new StringTokenizer(getParameter("filters"), "+");
      while(st.hasMoreTokens()) { // Разборка строки с названиями кнопок
```

```
            Button b = new Button(st.nextToken());   // Создание кнопки
            b.addActionListener(this);               // Регистрация кнопки
            p.add(b);                                // Добавление кнопки
         }
         lab = new Label("");
         add(lab, BorderLayout.NORTH);
         img = getImage(getDocumentBase(), getParameter("img"));
         lim = new LoadedImage (img);
         add(lim, BorderLayout.CENTER);
      }
      public void actionPerformed(ActionEvent ae) {
         String a = "";
         try {
            a = (String)ae.getActionCommand();
            if (a.equals("Reset")) {
               lim.set (img);
               lab.setText("Normal");
            }
            else {
               pif = (PlugInFilter) Class.forName(a).newInstance();
               fimg = pif.filter(this, img);
               lim.set(fimg);
               lab.setText("Filtered:  " + a);
            }
            repaint();
         }
         catch (ClassNotFoundException e) {
            lab.setText(a + " not found");
            lim.set(img);
            repaint();
         }
         catch  (InstantiationException e) {
            lab.setText("could't new " + a);
         }
         catch (IllegalAccessException e) {
            lab.setText("no access:  " + a);
         }
      }
   }
```

Как видно из кода апплета, в нем использованы классы `PlugInFilter`, `LoadedImage`, которые должны быть добавлены в программу. Состав проекта в среде Eclipse приведен на рис.8.

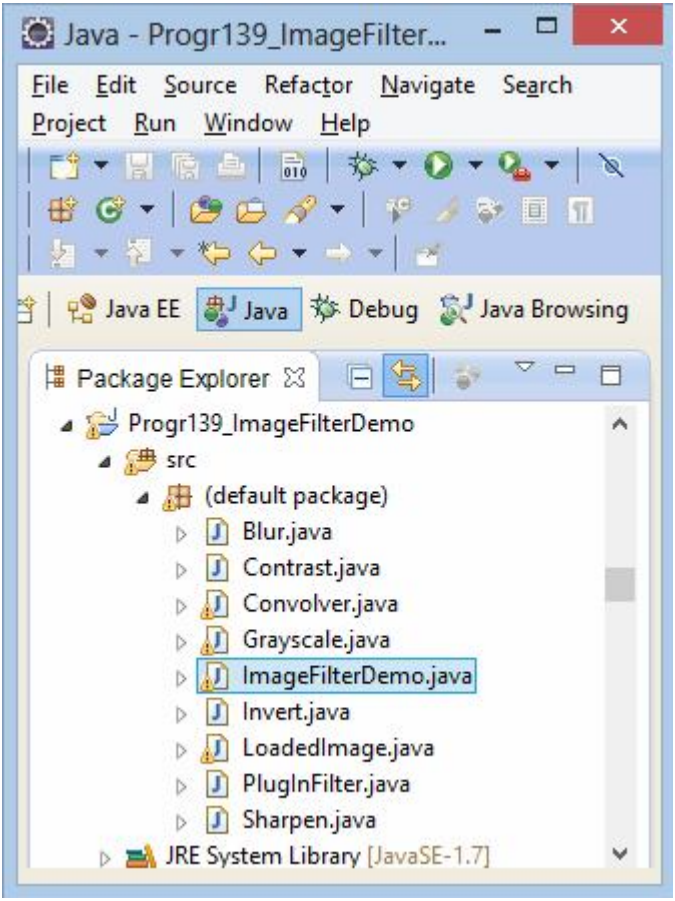


Рис. 8. Состав проекта

После полной разработки проекта для запуска апплета надо подготовить html-файл, в котором задать два параметра, один - с указанием файла изображения и второй - с названиями фильтров, разделенных знаком "+". Далее приведен пример такого html-файла:

```
<applet code = ImageFilterDemo width = 550 height = 550>
   <param name = img value = "brullov.jpg">
   <param name = filters value = "Grayscale+Invert+Contrast+Blur+Sharpen">
</applet>
```

При запуске апплета выводится изображение из указанного файла, рис.9.


Рис. 9. Исходное изображение

## PlugInFilter.java

`PlugInFilter` — это простой интерфейс, используемый для абстрактной фильтрации изображения. Он имеет только один метод `filter()`, который берет апплет и исходное изображение и возвращает новое изображение, которое было отфильтровано некоторым способом.

```
// Файл PlugInFilter.java
```

```
interface PlugInFilter {
   java.awt.Image filter(java.applet.Applet a, java.awt.Image in);
}
```

### LoadedImage.java

Loadedimage — это удобный Canvas-подкласс, который берет изображение во время конструирования и синхронно загружает его, используя загрузчик MediaTracker. LoadedImage ведет себя должным образом под управлением менеджера компоновки LayoutManager потому, что он переопределяет методы getPreferredSize() и getMinimumSize(). Он также определяет специальный установочный метод set(). С его помощью можно добиться такого вида нового изображения (Image-объекта), чтобы оно могло отображаться в данном (пустом) окне (Canvas-объекте). Именно так фильтрованное изображение отображается после того, как подключение заканчивается.

```
// Файл LoadedImage.java
import java.awt.*;
public class LoadedImage extends Canvas {
   Image img;
   public LoadedImage(Image i) {
      set(i);
   }
   void set(Image i) {
      MediaTracker mt = new MediaTracker(this);
      mt.addImage(i, 0);
      try {
         mt.waitForAll ();
      }
      catch (InterruptedException e) { };
      img = i;
      repaint();
   }
   public void paint(Graphics g) {
      if (img == null) {
         g.drawString("no image", 10, 30);
      }
      else {
         g.drawImage(img, 0, 0, this);
      }
   }
   public Dimension getPreferredSize() {
      return new Dimension(img.getWidth(this), img.getHeight(this));
   }
   public Dimension getMinimumSize() {
      return getPreferredSize();
   }
}
```

## Grayscale.java

Фильтр Grayscale – это подкласс фильтра RGBImageFilter. Данная формулировка означает, что Grayscale может использоваться как ImageFilter-параметр для конструктора FilteredImageSource. Тогда все, что нужно сделать, это переопределить метод fiiterRGB() так, чтобы изменить входящие значения цветов. Он берет красные, зеленые и синие значения и вычисляет яркость пиксела, используя NTSC фактор преобразования "цвет-яркость" (NTSC - National Television Standards Committee, — Национальный Телевизионный Комитет Стандартов). Затем он просто возвращает серый пиксел, который имеет ту же яркость, что и цветной источник.

```
// Файл Grayscale.java
import java.applet.*;
import java.awt.*;
import java.awt.image.*;
class Grayscale extends RGBImageFilter implements PlugInFilter {
   public Image filter(Applet a, Image in) {
      return a.createImage(new FilteredImageSource(in.getSource(), this));
   }
   public int filterRGB(int x, int y, int rgb) {
      int r = (rgb >> 16) & 0xff;
      int g = (rgb >> 8)  & 0xff;
      int b = rgb & 0xff;
      int k = (int) (.56 * g + .33 * r + .11 * b);
      return (0xff000000 | k << 16 | k << 8  |  k);
   }
}
```

## Invert.java

Фильтр Invert также весьма прост. Он разбирает пиксел, отбирая красные, зеленые и синие каналы, и затем инвертирует их, вычитая их из 255. Эти инвертированные значения упаковываются обратно в значение пиксела.

```
// Файл Invert.java
import java.applet.*;
import java.awt.*;
import java.awt.image.*;
class Invert extends RGBImageFilter implements PlugInFilter {
   public Image filter(Applet a, Image in) {
      return a.createImage (new FilteredImageSource(in.getSource(), this));
   }
   public int filterRGB(int x, int y, int rgb) {
      int r = 0xff - (rgb >> 16) & 0xff;
      int g = 0xff - (rgb >> 8) & 0xff;
      int b = 0xff - rgb & 0xff;
      return (0xff000000 | r << 16 | g << 8 | b);
   }
```

```
}
```

## Contrast.java

Фильтр `Contrast` очень похож на `Grayscale`, за исключением того, что его переопределение метода `fiiterRGB()` немного сложнее. Алгоритм, используемый им для улучшения контраста, берет красные, зеленые и синие значения отдельно и умножает их на `1.2`, если они ярче, чем `128`. Если их яркость ниже `128`, то они делятся на `1.2`. С помощью метода `multclamp()` обработанные таким способом значения, если нужно, ограничиваются величиной `255`.

```
// Файл Contrast.java
import java.applet.*;
import java.awt.*;
import java.awt.image.*;
public class Contrast extends RGBImageFilter implements PlugInFilter {
   public Image filter(Applet a, Image in) {
      return a.createImage(new FilteredImageSource(in.getSource(), this));
   }
   private int multclamp(int in, double factor) {
      in = (int) (in * factor);
      return in > 255 ? 255 : in;
   }
   double gain = 1.2;
   private int cont(int in) {
      return (in < 128) ? (int)(in/gain) : multclamp(in, gain);
   }
   public int filterRGB (int x, int y, int rgb) {
      int r = cont((rgb >> 16) & 0xff);
      int g = cont((rgb >>  8) & 0xff);
      int b = cont(rgb & 0xff);
      return (0xff000000 | r << 16 | g << 8  | b);
   }
}
```

## Convolver.java

Абстрактный класс `Convolver` выполняет базовую обработку фильтра свертывания, реализуя интерфейс потребителя `imageConsumer` для перемещения исходных пикселов в массив с именем `imgpixels`. Он также создает второй массив с именем `newimgpixels` для фильтрованных данных. Фильтры свертывания производят выборку маленького прямоугольника пикселов вокруг каждого пиксела в изображении, называемого *ядром свертывания.* Эта область (3x3 пиксела в данной демонстрации) используется для принятия рещения, как следует изменить центральный пиксел в этой области. Два конкретных подкласса, показанные в следующем разделе, просто реализуют метод

convolve(), используя imgpixels для исходных данных и newimgpixels для сохранения результата.

Причина того, что фильтр не может изменять массив imgpixels на месте, заключается в том, что следующий пиксел на строке сканирования пробовал бы использовать первоначальное значение для предыдущего пиксела, который был только что отфильтрован.

```
// Файл Convolver.java
import java.applet.*;
import java.awt.*;
import java.awt.image.*;
abstract class Convolver implements ImageConsumer, PlugInFilter {
   int width, height;
   int imgpixels[], newimgpixels[];
   abstract void convolve();          // Здесь идет фильтр
   public Image filter(Applet a, Image in) {
      in.getSource().startProduction(this);
      waitForImage();
      newimgpixels = new int[width * height];
      try {
         convolve();
      }
      catch (Exception e) {
         System.out.println("Convolver failed: " + e);
         e.printStackTrace();
      }
      return a.createImage(
            new MemoryImageSource(width, height, newimgpixels, 0, width));
   }
   synchronized void waitForImage() {
      try {
         wait();
      }
      catch (Exception e) { }
   }
   public void setProperties(java.util.Hashtable dummy) { }
   public void setColorModel(ColorModel dummy)   { }
   public void setHints(int dummy) { }
   public synchronized void imageComplete(int dummy) {
      notifyAll();
   }
   public void setDimensions(int x, int y) {
      width = x;
      height = y;
      imgpixels = new int[x * y];
   }
   public void setPixels(int x1, int y1, int w, int h,
            ColorModel model, byte pixels[], int off, int scansize) {
      int pix, x, y, x2, y2, sx, sy;
      x2 = x1 + w;
      y2 = y1 + h;
      sy = off;
      for(y = y1; y < y2; y++) {
```

```
            sx = sy;
            for(x = x1; x < x2; x++) {
               pix = model.getRGB(pixels[sx++]);
               if((pix & 0xff000000) == 0)
                  pix = 0x00ffffff;
               imgpixels[y * width + x] = pix;
            }
            sy += scansize;
         }
      }
      public void setPixels(int x1, int y1, int w, int h,
         ColorModel model, int pixels[], int off, int scansize) {
         int pix, x, y, x2, y2, sx, sy;
         x2 = x1 + w;
         y2 = y1 + h;
         sy = off;
         for(y = y1; y < y2; y++) {
            sx = sy;
            for(x = x1; x < x2; x++) {
               pix = model.getRGB(pixels[sx++]);
               if((pix & 0xff000000) == 0)
                  pix = 0x00ffffff;
               imgpixels[y * width + x] = pix;
            }
            sy += scansize;
         }
      }
   }
```

## Blur.java

Фильтр `Blur` — это подкласс `Convolver`. Он просто пробегает через каждый пиксел в исходном массиве изображения `imgpixels` и вычисляет среднее значение по всей окружающей его области 3x3. Результирующий пиксел помещается в соответствующую позицию массива `newimgpixels`. (Рис. 23.11 показывает апплет после `Blur`-фильтра.)

```
// Файл Blur.java
public class Blur extends Convolver {
   public void convolve() {
      for(int y = 1; y < height - 1; y++) {
         for(int x = 1; x < width - 1; x++) {
            int rs = 0;
            int gs = 0;
            int bs = 0;
            for(int k = -1; k <= 1; k++) {
               for(int j = -1; j <= 1; j++){
                  int rgb = imgpixels[(y + k) * width + x + j];
                  int r = (rgb >> 16) & 0xff;
                  int g = (rgb >>  8) & 0xff;
                  int b = rgb & 0xff;
                  rs += r;
```

```
                gs += g;
                bs += b;
            }
        }
        rs /= 9;
        gs /= 9;
        bs /= 9;
        newimgpixels[y * width + x] =
                    (0xff000000 | rs << 16 | gs << 8 | bs) ;
    }
  }
 }
}
```

## Sharpen.java

Фильтр `Sharpen` — также подкласс `Convolver` и является инверсией `Blur`. Он пробегает через каждый пиксел в исходном массиве изображения `imgpixels` и вычисляет среднее значение по окружающему `3x3` блоку пикселов, не считая центральный. Соответствующие выводные пикселы в `newimgpixels` различаются между собой (в зависимости от окружения центрального пиксела) добавляемыми в них усреднениями. Грубо говоря, если пиксел в тридцать раз ярче, чем его окружение, подобная фильтрация делает примерно тридцать окружающих пикселов ярче. Если, однако, он в десять раз темнее, то примерно десять окружающих пикселов станут более темными. Это приводит к подчеркиванию граней изображения, оставляя гладкие области неизменными.

```
// Файл Sharpen.java
public class Sharpen extends Convolver {
   private final int clamp(int c) {
      return (c > 255 ? 255 : (c < 0 ? 0 : c) );
   }
   public void convolve() {
      int r0 = 0, g0 = 0, b0 = 0;
      for(int y = 1; y < height - 1; y++) {
         for(int x = 1; x < width - 1; x++) {
            int rs = 0;
            int gs = 0;
            int bs = 0;
            for(int k = -1; k <= 1; k++) {
               for(int j = -1; j <= 4; j++) {
                  int rgb = imgpixels[(y + k) * width + x + j];
                  int r = (rgb >> 16) & 0xff;
                  int g = (rgb >> 8) & 0xff;
                  int b = rgb & 0xff;
                  if (j == 0 && k == 0) {
                     r0 = r;
                     g0 = g;
                     b0 = b;
```

```
                    }
                    else {
                        rs += r;
                        gs += g;
                        bs += b;
                    }
                }
            }
            rs >>= 3;
            gs >>= 3;
            bs >>= 3;
            newimgpixels [y * width + x] = (0xff000000 |
                                            clamp(r0 + r0 - rs) << 16 |
                                            clamp(g0 + g0 - gs) <<8 |
                                            clamp(b0 + b0 - bs));
        }
    }
  }
}
```