

# Лабораторная работа № 5

## Создание многопоточного приложения с элементами вывода графической информации

### Оглавление

Цель лабораторной работы.....	2
1 Порядок прорисовки содержимого компонентов графического пользовательского интерфейса на базе библиотеки Swing .....	2
2 Построение изображения компонентов с использованием библиотеки Java 2D .....	5
2.1 Создание изображения с использованием контекста отображения, представленного объектом класса <i>Graphics2D</i> .....	6
2.2 Создание и использование стиля линий. Интерфейс <i>Stroke</i> и класс <i>BasicStroke</i> .....	8
2.3 Создание и использование цветовых схем. Интерфейс <i>Paint</i> . .....	10
2.3.1 Создание однородной цветовой схемы. Классы <i>Color</i> и <i>SystemColor</i> . .....	11
2.3.2 Создание градиентных цветовых схем. ....	12
2.3.3 Создание цветовых схем на основе текстур. ....	16
2.4 Построение геометрических фигур.....	17
2.4.1 Создание сегментов прямых линий. Класс <i>Line2D</i> . ....	19
2.4.2 Создание сложных линий. Класс <i>GeneralPath</i> . ....	20
2.4.3 Создание дуг. Класс <i>Arc2D</i> . ....	22
2.4.4 Создание эллипсов. Класс <i>Ellipse2D</i> .....	24
2.4.5 Создание прямоугольников. Класс <i>Rectangle2D</i> .....	25
2.4.6 Создание сложных геометрических фигур. Класс <i>Area</i> .....	26
2.5 Рисование текста. ....	27
2.6 Преобразование изображений. Класс <i>AffineTransform</i> . ....	30
2.6.1 Преобразование масштабирования. ....	32
2.6.2 Преобразование вращения. ....	33
2.6.3 Преобразование смещения.....	34
2.6.4 Преобразование сдвига. ....	35
2.7 Использование фигур усечения и правил композиции. ....	36
3 Построение многопоточных приложений .....	37
3.1 Представление потоков выполнения в Java. Класс <i>Thread</i> .....	38
3.2 Способы создания потоков выполнения.....	40
3.2.1 Использование наследования от класса <i>Thread</i> .....	40
3.2.2 Использование класса, реализующего интерфейс <i>Runnable</i> , без наследования от класса <i>Thread</i> . ....	41
3.3 Организация взаимодействия потоков в многопоточном приложении.....	42
3.3.1 Организация прерывания выполнения потока. ....	42

3.3.2 Синхронизация потоков.....	44
4 Базовое приложение для лабораторной работы .....	47
4.1 Структура приложения.....	47
4.2 Реализация класса поля для движения фигур <i>Field</i> . ....	48
4.3 Реализация класса движущейся фигуры <i>MovingFigure</i> .....	50
4.4 Реализация главного окна приложения.....	54
.....	54
5 Задания.....	55
5.1 Вариант сложности А.....	55
5.2 Вариант сложности В.....	56
5.3 Вариант сложности С.....	57
Приложение 1. Исходный код базового приложения .....	60

## Цель лабораторной работы

Получить практические навыки разработки многопоточных приложений на Java, а также формирования графического изображения компонентов с использованием библиотеки Java 2D.

## 1 Порядок прорисовки содержимого компонентов графического пользовательского интерфейса на базе библиотеки Swing

Отличительной чертой большинства компонентов графического пользовательского интерфейса (Graphical user interface – GUI) библиотеки Swing является то, что все они для отображения на экране выполняют прорисовку своего содержимого в рамках контейнера, в котором они расположены. Такие компоненты GUI обычно называют “легковесными”. Процесс перерисовки содержимого контейнера верхнего уровня библиотеки Swing показан на рисунке 1.

В случае если операционная система, в рамках которой работает Java приложение, обнаружила, что окно верхнего уровня должно быть перерисовано либо полностью (например, при первом отображении на экране), либо частично (часть окна, ранее закрытая другим окном, стала видимой), она посылает контейнеру верхнего уровня сообщение о перерисовке, в результате обработки которого управление передается методу *paint*.

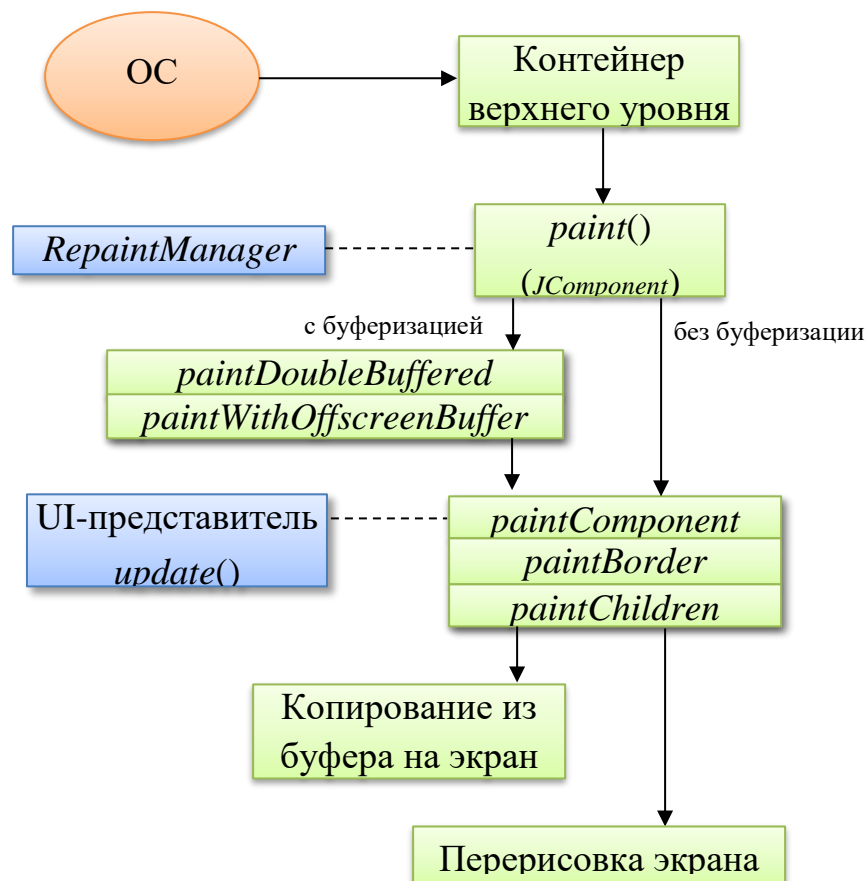


Рис 1. Схема перерисовки содержимого контейнера верхнего уровня в Swing

В большинстве случаев в компонентах библиотеки Swing используется реализация метода *paint* класса *JComponent*, который является суперклассом для всех компонентов данной библиотеки. Реализация *paint* данного класса максимально эффективно организует всю работу по перерисовке содержимого контейнера и поэтому обычно не переопределяется в подклассах.

Метод *paint* выполняет следующий набор действий:

1. Принимает решение об использовании двойной буферизации при выполнении двух условий:
  - метод *isDoubleBufferingEnabled* объекта менеджера перерисовки (типа *RepaintManager*) возвращает *true*, что указывает на необходимость использования двойной буферизации в приложении в целом (ссылка на объект менеджера перерисовки может быть получена при помощи вызова *RepaintManager.currentManager*);
  - метод *isDoubleBuffered* перерисовываемого компонента возвращает *true*, что указывает на необходимость использования двойной буферизации для данного компонента.
2. Если двойная буферизация используется, то вызывается метод *paintDoubleBuffered*, в котором настраивается внеэкранный буфер: получается изображение необходимого компоненту размера, для которого затем создается объект типа *Graphics* и вызывается метод *paintWithOffScreenBuffer*, инициирующий непосредственно прорисовку

компонента. В противном случае будет использован объект типа *Graphics*, позволяющий рисовать непосредственно на экране.

3. Вызывается метод *paintComponent*, выполняющий прорисовку изображения компонента используя переданный в него в качестве аргумента объект типа *Graphics*. Код прорисовки компонента GUI располагается:

- непосредственно в методе *paintComponent*, если компонент должен выглядеть всегда одинаково;
- в методах *paint* классов для создания UI-представителей, если требуется поддержание множественности представления на различных платформах.

При прорисовке компонента важную роль играет его **непрозрачность** (величина типа *boolean*, для получения которой используется метод *isOpaque*, а для изменения метод *setOpaque*). Если ее значение равно *true*, то для корректного отображения, код перерисовки компонента, обязан закрашивать все пиксели в рамках соответствующей ему области. В противном случае в не закрашенных при прорисовке областях компонента системой рисования Swing будут отображаться части других компонентов, расположенных под данным компонентом. Это может использоваться на практике для создания компонентов непрямоугольной формы, однако обеспечение прозрачности компонентов приводит к увеличению затрат на их прорисовку.

4. Вызывается метод *paintBorder*, который прорисовывает рамку вокруг компонента, если она (ссылка на объект типа *Border*) для него установлена. Поскольку рамка рисуется поверх изображения, сформированного методом *paintComponent*, то при ее наличии стоит оставлять для нее место. Для определения прямоугольника, соответствующего месту внутри рамки можно воспользоваться статическим методом *getInteriorRectangle* класса *AbstractBorder*.

5. Вызывается метод *paintChildren*, выполняющий прорисовку компонентов, вложенных в данный компонент.

После завершения прорисовки содержимого компонента в случае, если использовалась двойная буферизация, изображение из буфера копируется на экран.

Для инициирования перерисовки содержимого компонента из кода программы используется метод *repaint*, который должен вызываться для перерисовываемого компонента.

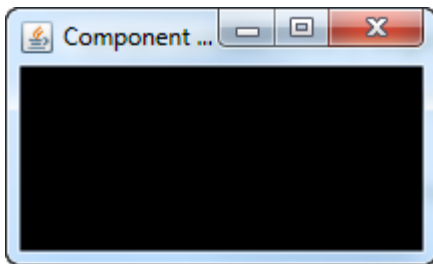
## 2 Построение изображения компонентов с использованием библиотеки Java 2D

Для разработки собственного компонента GUI в первую очередь необходимо выбрать один классов библиотеки Swing, от которого будет порожден класс нового компонента. В качестве такого класса может выступать базовый класс для всех компонентов Swing *JComponent*, однако его использование для вышеуказанной цели сопряжено со следующими неудобствами:

1. данный класс не заливает при перерисовке принадлежащую ему область установленным для него фоновым цветом;
2. по умолчанию он является прозрачным (*isOpaque* возвращает false), что не всегда необходимо и делает процесс прорисовки несколько менее эффективным.

Данные неудобства отсутствуют у класса *JPanel*, который напрямую порожден от *JComponent*, поэтому при создании собственных компонентов именно *JPanel* часто используется в качестве базового класса.

Простейший пример класса собственного компонента *JMyComponent*, порожденного от *JPanel* и всего лишь устанавливающего по умолчанию черный цвет в качестве фонового приведен на рисунке 2.



```
public class JMyComponent
    extends JPanel {
    public JMyComponent() {
        setBackground(Color.black);
    }

    protected void paintComponent(
        Graphics g) {
        super.paintComponent(g);
    }
}
...
JMyComponent comp = new JMyComponent();
getContentPane().add(comp);
...
```

Рис 2. Класс собственного компонента, унаследованный от *JPanel*, и его использование

Как видно из кода класса *JMyComponent* переопределенный в нем метод *paintComponent* получает ссылку на объект класса *Graphics* библиотеки AWT, который используется как виртуальный холст для рисования содержимого компонента.

Для обеспечения доступа к возможностям библиотеки Java 2D необходимо использовать класс *Graphics2D*, который является подклассом *Graphics* и расширяет его возможности. Тип объекта, передаваемого в качестве аргумента в метод *paintComponent*, может быть приведен к классу *Graphics2D*, как показано в представленном ниже фрагменте кода.

```

public void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2d = (Graphics2D) g;
    // Дальнейшие действия с объектом g2d
    // ...
}

```

Полученный в результате такого приведения типов объект класса *Graphics2D* используется в дальнейшем для создания изображения компонента GUI.

## 2.1 Создание изображения с использованием контекста отображения, представленного объектом класса *Graphics2D*.

Для рисования графики при помощи объекта типа *Graphics2D* используется следующий набор шагов:

1. задать требуемые настройки холста:
  - задать текущую штриховку (метод *setStroke*);
  - задать текущую цветовую схему (методы *setPaint* или *setColor*);
  - задать текущий шрифт (метод *setFont*);
  - задать преобразование координат (метод *setTransform*);
  - задать область усечения (метод *setClip*);
  - задать правила композиции (метод *setComposite*);
2. нарисовать необходимые элементы:
  - создать объект графического элемента с использованием подходящего класса или набора классов геометрических фигур;
  - нарисовать контуры элемента (метод *draw*);
  - закрасить внутреннюю часть фигуры (метод *fill*).

При рисовании в каждом конкретном случае можно использовать только те из приведенных выше действий, которые необходимы для создания требуемого изображения. Для создания более сложных изображений поведенные выше действия могут повторяться несколько раз.

Любая геометрическая фигура, участвующая в формировании необходимого изображения, рисуется не сразу после создания соответствующего объекта, а только при вызове методов *draw* и/или *fill*. Алгоритм рисования геометрической фигуры представлен на рисунке 3.



Рис 3. Алгоритм рисования геометрической фигуры при использовании объекта класса *Graphics2D*

Как видно из рисунка 3 при прорисовке фигуры последовательно выполняются следующие этапы:

- прорисовывается контур фигуры;
- применяются заданные преобразования (сдвига, поворота и т.п.);
- фигура усекается (если фигура и область усечения не пересекаются, то процесс визуализации прекращается);
- оставшаяся после усечения часть фигуры закрашивается;
- точки закрашенной фигуры совмещаются с точками ранее созданного изображения с использованием определенных правил композиции.

При использовании объекта класса *Graphics2D* для рисования в качестве контекста отображения, можно настраивать его параметры задавая ряд специальных объектов. Методы, используемые для задания этих объектов приведены в таблице 1.

Таблица 1 Методы для настройки свойств контекста отображения, представленного объектом класса *Graphics2D*

Название метода	Описание
<i>Stroke</i> <b>getStroke()</b> <i>void setStroke(Stroke s)</i>	Позволяют получить/установить текущий тип штриховки, для рисования линий
<i>Color</i> <b>getColor()</b> <i>Paint</i> <b>getPaint()</b> <i>void setColor(Color c)</i> <i>void setPaint (Paint p)</i>	Позволяют получить/установить текущую цветовую схему для рисования линий и закрашивания внутренней области фигур. Поскольку класс <i>Color</i> реализует интерфейс <i>Paint</i> , то вызов <i>setColor(c)</i> , эквивалентен вызову <i>setPaint(c)</i> . Неоднородные заливки задаются только через <i>setPaint</i>
<i>Font</i> <b>getFont()</b> <i>void setFont(Font font)</i>	Позволяют получить/установить текущий шрифт для холста
<i>AffineTransform</i> <b>getTransform()</b> <i>void setTransform(AffineTransform Tx)</i>	Позволяют получить/установить преобразование, которое будет применяться к рисуемым геометрическим фигурам
<i>Rectangle</i> <b>getClipBounds()</b> <i>void setClip(int x, int y, int width, int height)</i> <i>void setClip(Shape clip)</i>	Позволяют получить/установить текущую область отсечения для холста, вне которой результаты рисования отображаться не будут
<i>Composite</i> <b>getComposite()</b> <i>void setComposite(Composite comp)</i>	Позволяют получить/установить правила для объединения новой фигуры с ранее созданным изображением

Как видно из таблицы 1, часть настроек контекста отображения, представленного объектом типа *Graphics2D*, используется при начертании границ рисуемого графического элемента изображения и заполнении его внутреннего пространства. Для задания этих настроек задействуются объекты двух типов: *Stroke* и *Paint*.



## 2.2 Создание и использование стиля линий. Интерфейс *Stroke* и класс *BasicStroke*.

Интерфейс *Stroke* используется для задания в рамках объекта класса *Graphics2D* формы и толщины штриха, который будет использоваться для рисования линий и границ геометрических фигур. По умолчанию в объектах класса *Graphics2D* для штриховки используется сплошная линия толщиной 1 пиксель.

Библиотека Java 2D включает только один класс *BasicStroke*, реализующий интерфейс *Stroke*, и позволяющий создавать объекты, ссылки на которые можно передавать в метод *setStroke* объекта класса *Graphics2D*, тем самым изменяя тип штриховки. Данный класс включает набор конструкторов, которые в качестве аргументов принимают параметры штриховки. Конструктор с наиболее полным набором аргументов имеет вид:




```
BasicStroke(float width, int cap, int join, float miterlimit,  
            float[] dash, float dash_phase);
```

Рассмотрим назначение и возможные значения каждого из аргументов, а также их влияние на внешний вид рисуемой линии.

Аргумент *width* задает толщину линии. Значение данного аргумента должно быть неотрицательным вещественным числом типа *float*. Значение 0.0f интерпретируется как наименьшая возможная толщина линии для устройства, на котором ведется рисование.

Аргумент *cap* определяет оформление края линий и может принимать значения, представленные в таблице 2.

Таблица 2 Возможные значения для оформления края линий в классе *BasicStroke*




Название	Описание	Вид края линии*
<i>CAP_BUTT</i>	делает край линии резко обрубленным	
<i>CAP_ROUND</i>	Оформляет край линии в виде полукруга с радиусом равным половине толщины линии	
<i>CAP_SQUARE</i>	Завершает линию половиной квадрата со стороной равной толщине линии	

\*белым цветом прочерчена тонкая линия для того, чтобы показать где находятся координаты края линии.



Аргумент *join* задает оформление смыкания двух линий и может принимать значения, описанные в таблице 3.

Таблица 3 Возможные значения для оформления смыкания двух линий в классе *BasicStroke*

Название	Описание	Вид смыкания линий*
<i>JOIN_BEVEL</i>	объединяет линии путем соединения внешних углов их широких контуров прямой линией, перпендикулярной биссектрисе угла между соединяемыми линиями	
<i>JOIN_ROUND</i>	соединяет линии путем закругления угла с использованием радиуса равного половине ширины линии	
<i>JOIN_MITER</i>	соединяет линии путем удлинения их внешних сторон до соединения, образуя острый «шип»	

\*белым цветом прочерчена тонкая линия для того, чтобы показать где находятся координаты края линий.

Для использования значения *JOIN\_MITER* введен минимальный угол (по умолчанию 10 градусов), при котором оно может применяться. Это связано с тем, что при малых углах между смыкающимися линиями применение параметра *JOIN\_MITER* может приводить к появлению очень длинных «шипов». Если угол между объединяемыми прямыми не превышает минимального значения, то вместо *JOIN\_MITER* применяется *JOIN\_BEVEL*.

Аргумент *miterlimit* позволяет задать предельный угол, при превышении которого может использоваться параметр *JOIN\_MITER*.

Аргумент *dash* определяет шаблон штриховки, который будет использоваться для рисования линии. Значение аргумента является ссылкой на массив чисел типа *float*, которые определяют длину чередующихся штрихов («линия есть») и пробелов («линии нет»). Примеры применения различных шаблонов приведены на рисунке 4. Как видно из данного рисунка, если длина линии превышает длину шаблона, то после окончания рисования одного блока, соответствующего заданному шаблону, сразу же начинается рисование следующего такого же блока. При этом штрихи и пробелы будут чередоваться в рамках всей линии в целом, а не в рамках отдельных блоков, соответствующих шаблону. По этой причине при нечетном количестве элементов в массиве *dash* нечетные блоки, соответствующие шаблону, будут начинаться и заканчиваться штрихами, а четные – пробелами, т.е. в четных на

месте штрихов в шаблоне будут пробелы, а на месте пробелов – штрихи, что продемонстрировано на рисунке 5.

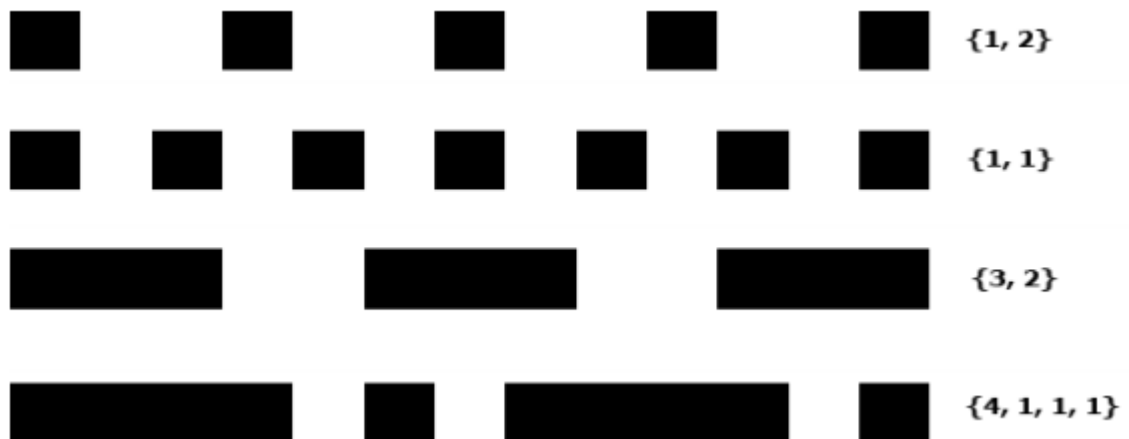


Рис 4. Примеры применения различных шаблонов для штриховки



Рис 5. Пример применения шаблона с нечетным количеством значений

Аргумент *dash\_phase* задает смещение относительно начала шаблона, с которым он будет рисоваться в начале линии. По умолчанию данное значение равно 0. Использование данного параметра продемонстрировано на рисунке 6.



Рис 6. Пример применения смещения начала шаблона

Как видно из рисунка 6, если значение параметра *dash\_phase*=1, то при прорисовке первого блока изображение шаблона начинается с позиции, смещенной на 1 (нижняя линия) по сравнению с его обычным видом (верхняя линия).

### 2.3 Создание и использование цветовых схем. Интерфейс *Paint*.

Интерфейс *Paint* предназначен для задания цветовой схемы, которая будет использована при рисовании методами *draw* и *fill* объекта класса *Graphics2D*. Для задания цветовой схемы в объекте класса *Graphics2D* используется метод *setPaint*, в который в качестве аргумента передается ссылка

на объект типа *Paint*. Иерархия классов, реализующих интерфейс *Paint*, представлена на рисунке 7.

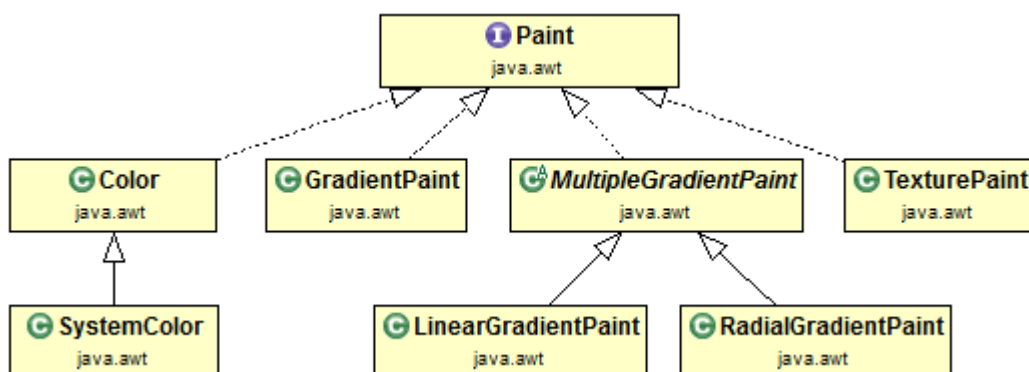


Рис 7. Иерархия классов, реализующих интерфейс *Paint*

### 2.3.1 Создание однородной цветовой схемы. Классы *Color* и *SystemColor*.

Классы *Color* и *SystemColor* позволяют задать однородную цветовую схему, включающую один цвет.

Класс *Color* предоставляет ряд статических констант для задания 13 стандартных цветов, каждая из которых существует как в виде заглавных, так и прописных букв. Набор этих констант приведен в таблице 4.

Таблица 4 Константы стандартных цветов, определенные в классе *Color*

Название	Описание	Название	Описание
<i>BLACK, black</i>	черный	<i>MAGENTA, magenta</i>	пурпурный
<i>BLUE, blue</i>	синий	<i>ORANGE, orange</i>	оранжевый
<i>CYAN, cyan</i>	голубой	<i>PINK, pink</i>	розовый
<i>DARK_GRAY, darkGray</i>	темно-серый	<i>RED, red</i>	красный
<i>GRAY, gray</i>	серый	<i>WHITE, white</i>	белый
<i>GREEN, green</i>	зеленый	<i>YELLOW, yellow</i>	желтый
<i>LIGHT_GRAY, lightGray</i>	светло-серый		

Класс *Color* также позволяет создать собственный цвет на основе вкладов красной (*red*), зеленой (*green*) и синей (*blue*) компонент, которые задаются константами от 0 до 255. Для этого может использоваться соответствующий конструктор класса *Color* вида:

```
Color(int red, int green, int blue, int alpha)
```

Дополнительный аргумент *alpha* задает степень прозрачности: 0 – полностью прозрачный, 255 – полностью непрозрачный.

Класс *SystemColor* содержит статические константы, соответствующие цветам, используемым при отображении различных элементов

пользовательского интерфейса. Название и описание этих констант может быть найдено в справочной документации по данному классу.

### 2.3.2 Создание градиентных цветовых схем.


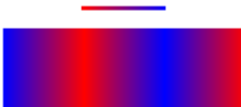
Класс *GradientPaint* позволяет создавать градиентную цветовую схему, в которой цвет постепенно меняется между двумя заранее заданными значениями. Для вычисления промежуточных цветов используется интерполяция. Для задания параметров градиентной цветовой схемы используются конструкторы класса *GradientPaint*. Конструкторы, позволяющие задать наибольшее число установок имеют вид:

```
GradientPaint(float x1, float y1, Color color1,  
              float x2, float y2, Color color2, boolean cyclic);  
GradientPaint(Point2D pt1, Color color1,  
              Point2D pt2, Color color2, boolean cyclic);
```

Оба конструктора используют две точки, координаты которых в первом из конструкторов задаются по отдельности ((*x1*, *y1*) и (*x2*, *y2*)), а во втором – с использованием объектов типа *Point2D*, которые содержат обе координаты *x* и *y*. Данные точки определяют в каком направлении и на каком расстоянии в двухмерном пространстве цвет будет изменяться от значения *color1*, до значения *color2*. В направлении перпендикулярном заданному отрезку цвет изменяться не будет. Аргумент *cyclic* определяет, как будет изменяться цвет вне заданного отрезка в том же направлении. Если значение *cyclic* равно *false* (значение по умолчанию), то цвет на прямой, на которой лежит заданный отрезок, до его первой точки всегда будет равен *color1*, а после второй точки – всегда будет равен *color2*. Если аргумент *cyclic* равен *true*, градиент цветов будет периодически повторяться левее первой и правее второй точки.

Пример использования градиентной заливки приведен на рисунке 8. В приведенном фрагменте кода градиент цветов меняется от красного до синего цвета на горизонтальном отрезке между точками *startPt* и *endPt*. Далее с использованием данной градиентной цветовой схемы рисуется линия, между точками *startPt* и *endPt*, позволяющая визуально оценить отрезок, в рамках которого была задана цветовая схема. После этого рисуется прямоугольник, который по оси *X* начинается до точки *startPt* и заканчивается после точки *endPt*. Внешний вид прямоугольника показывает, что градиент цветов наблюдается только вдоль горизонтальной оси. Если значение *cyclic* не задано, а, следовательно, равно *false*, то цвет слева от первой точки остается красным, а справа от последней – синим.

```
Graphics2D g2d = (Graphics2D)g;
float length=40;
Point2D startPt = new Point2D.Float(10+length,10);
Point2D endPt = new Point2D.Float(10+2*length,10);
g2d.setStroke(new BasicStroke(2));
```

<pre>g2d.setPaint(new GradientPaint(     startPt, Color.red,     endPt, Color.blue));</pre>	<pre>g2d.setPaint(new GradientPaint(     startPt, Color.red,     endPt, Color.blue, true));</pre>
	

```
//Создаем линию 'line' от startPt до endPt
...
g2d.draw(line);

//Создаем прямоугольник 'rect'
//по оси X: от точки startPt.x-length, шириной 3*length
//по оси Y: от точки startPt.y+10, высотой length
...
g2d.fill(rect);
```

Рис 8. Применение градиентной цветовой схемы, созданной с использованием класса `GradientPaint`

Если *cyclic* равно *true*, то градиент цветов распространяется и за пределы исходного отрезка, причем каждый раз является зеркальным отражением градиента на предыдущем отрезке относительно границы между отрезками.

**Класс *LinearGradientPaint*** по аналогии с *GradientPaint* позволяет создавать градиентную цветовую схему с изменением цвета на некотором отрезке прямой. Однако есть и ряд существенных отличий от *GradientPaint*:

1. можно использовать более двух опорных цветов;
2. можно регулировать местоположение каждого опорного цвета в рамках градиента;
3. используется расширенный набор опций для задания поведения градиента вне отрезка, на котором он определен.

Определение конструкторов класса *LinearGradientPaint*, имеет вид:

```
LinearGradientPaint(float startX, float startY, float endX, float endY,
    float[] fractions, Color[] colors,
    MultipleGradientPaint.CycleMethod cycleMethod);
LinearGradientPaint(Point2D start, Point2D end,
    float[] fractions, Color[] colors,
    MultipleGradientPaint.CycleMethod cycleMethod);
```

Первых четыре аргумента в первом конструкторе и два первых аргумента во втором конструкторе определяют границы отрезка, на котором определен градиент цветов. Аргумент *colors* задает массив опорных цветов (не менее

двух), местоположение которых в рамках отрезка, определяется значениями массива *fractions*. Значения в массиве *fractions* должны быть в диапазоне от 0 до 1, идти строго по возрастанию и их количество должно быть равно числу элементов в массиве *colors*. Сами значения задают расстояние от начала отрезка до точки, где должен располагаться соответствующий цвет из массива *colors*, в предположении, что длина всего отрезка эквивалентна 1. Если первое и последнее значения в массиве *fractions* не равны, соответственно, 0 и 1, то данный массив автоматически будет дополнен этими значениями, а в начало и конец массива *colors* так же будет добавлено две величины, равные первому и последнему элементам исходного массива *colors*, соответственно. Принцип дополнения массивов *fractions* и *colors* продемонстрирован на примере, приведенном в таблице 5.

Таблица 5 Принцип дополнения массивов *fractions* и *colors* при использовании классов, порожденных от *MultipleGradientPaint*

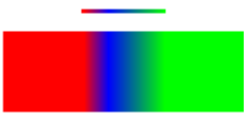

Исходные массивы	<code>colors = {Color.blue, Color.red};</code> <code>fractions = {0.3f, 0.7f};</code>
Дополненные массивы	<code>colors = {Color.blue, Color.blue, Color.red, Color.red};</code> <code>fractions = {0f, 0.3f, 0.7f, 1f};</code>


Параметр *cycleMethod* определяет поведение цветовой схемы за пределами отрезка, на котором она была определена и может принимать одно из значений перечисления *MultipleGradientPaint.CycleMethod*:

- *NO\_CYCLE* (используется по умолчанию) – цвета на краях отрезка будут использованы для заполнения пространства за его пределами;
- *REFLECT* – цвета градиента будут циклически повторяться путем зеркального отражения на границе отрезка;
- *REPEAT* – цвета градиента будут циклически повторяться путем копирования последовательности цветов исходного градиента.

Пример использования класса *LinearGradientPaint* приведен на рисунке 9.

```
Graphics2D g2d = (Graphics2D)g;
float length=40;
Point2D startPt = new Point2D.Float(10+length,10);
Point2D endPt = new Point2D.Float(10+2*length,10);
Color[] colors = {Color.red, Color.blue, Color.green};
float[] fractions = {0f, 0.3f, 1f};
```

	<code>g2d.setPaint(new LinearGradientPaint( startPt, endPt, fractions, colors));</code>
	<code>g2d.setPaint(new LinearGradientPaint( startPt, endPt, fractions, colors, MultipleGradientPaint.CycleMethod.REFLECT));</code>

	<pre>g2d.setPaint(new LinearGradientPaint(     startPt, endPt, fractions, colors,     MultipleGradientPaint.CycleMethod.REPEAT));</pre>
---	---

```
//Создаем линию 'line' от startPt до endPt
...
g2d.draw(line);

//Создаем прямоугольник 'rect'
//по оси X: от точки startPt.x-length, шириной 3*length
//по оси Y: от точки startPt.y+10, высотой length
...
g2d.fill(rect);
```

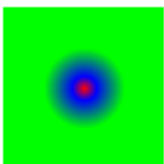
Рис 9. Применение градиентной цветовой схемы, созданной с использованием класса *LinearGradientPaint*

Класс ***RadialGradientPaint*** позволяет создавать круговую градиентную цветовую схему. Конструктор класса *RadialGradientPaint* с максимальным числом параметров имеет вид:

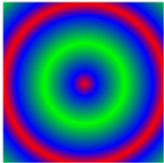
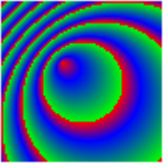
```
RadialGradientPaint(Point2D center, float radius, Point2D focus,
    float[] fractions, Color[] colors,
    MultipleGradientPaint.CycleMethod cycleMethod);
```

Конструктор позволяет задать круг с центром в точке *center* и радиусом равным *radius*. Точка *focus* задает местоположение внутри круга, откуда будет начинаться расчет градиента (по умолчанию совпадает с *center*). Если данная точка задана вне круга, то для начала расчета градиента будет использована точка, лежащая на пересечении внешней границы круга и прямой, соединяющей точку *center* с точкой *focus*. Аргументы *fractions*, *colors* и *cycleMethod* интерпретируются точно так же, как и для класса *LinearGradientPaint*. Пример использования класса *RadialGradientPaint* приведен на рисунке 10.

```
Graphics2D g2d = (Graphics2D)g;
Point2D center = new Point2D.Float(50,50);
float radius = 20;
Point2D focus = new Point2D.Float(40,40);
Color[] colors = {Color.red, Color.blue, Color.green};
float[] fractions = {0f, 0.3f, 1f};
```

	<pre>g2d.setPaint(new RadialGradientPaint(     center, radius, fractions, colors));</pre>
---	---



	<pre>g2d.setPaint(new RadialGradientPaint(     center, radius, fractions, colors,     MultipleGradientPaint.CycleMethod.REFLECT));</pre>
	<pre>g2d.setPaint(new RadialGradientPaint(     center, radius, focus, fractions, colors,     MultipleGradientPaint.CycleMethod.REPEAT));</pre>

```
//Создаем прямоугольник 'rect'
//по оси X: от точки center.x-2*radius, шириной 4*radius
//по оси Y: от точки center.y-2*radius, высотой 4*radius
...
g2d.fill(rect);
```

Рис 10. Применение градиентной цветовой схемы, созданной с использованием класса *RadialGradientPaint*

На рисунке 10 представлено три различных круговых градиентных цветowych схемы, каждая из которых содержит три цвета. Для первой и второй схем точка фокуса совпадает с центром круга, а для третьей смещена влево и вверх относительно центра. Первая схема не использует повторение градиента, вторая использует зеркальное отражение, а третья – повторение путем копирования.

### 2.3.3 Создание цветowych схем на основе текстур.

Класс *TexturePaint* позволяет создавать цветovou схему с использованием текстуры, представленной изображением в виде объекта класса *BufferedImage*. Конструктор класса *TexturePaint* имеет следующий вид:




```
TexturePaint(BufferedImage texture, Rectangle2D anchorRect);
```

Второй аргумент конструктора задает *прямоугольник привязки* (*anchor rectangle*). Для создания цветовой схемы изображение *texture* копируется в верхний-левый угол прямоугольника привязки и масштабируется таким образом, чтобы полностью заполнить прямоугольник. Затем полученный прямоугольник многократно повторяется вдоль осей X и Y на плоскости настолько это необходимо для рисования требуемого графического элемента. Пример использования цветовой схемы, созданной с использованием класса *TexturePaint*, приведен на рисунке 11.



– использованная текстура размером 38x38 пикселей из файла texture.gif

```
Graphics2D g2d = (Graphics2D)g;
BufferedImage texture = ImageIO.read(new File("texture.gif"));
```

	<pre>Rectangle2D anchorRect = new Rectangle2D.Float(     0, 0, texture.getWidth(), texture.getHeight());</pre>
	<pre>Rectangle2D anchorRect = new Rectangle2D.Float(     0, 0, texture.getWidth()+20, texture.getHeight()+20);</pre>
	<pre>Rectangle2D anchorRect = new Rectangle2D.Float(     0, 0, texture.getWidth()-20, texture.getHeight()-20);</pre>

```
//Создаем эллипс 'ellipse'
...
g2d.fill(ellipse);
```

Рис 11. Применение цветовой схемы, созданной с использованием класса *TexturePaint*

Как видно из приведенного примера вид текстуры сохраняется, если прямоугольник привязки имеет размеры, точно соответствующие изображению текстуры. В противном случае изображение масштабируется для его заполнения и, соответственно, рисунок заполнения рисуемых фигур (эллипса в приведенном примере) будет выглядеть крупнее или мельче, чем рисунок на образце исходной текстуры.

## 2.4 Построение геометрических фигур.

При рисовании изображений компонентов GUI средствами библиотеки Java 2D используются различные геометрические фигуры, представленные в виде объектов стандартных классов, включенных в библиотеку.

Все классы геометрических фигур библиотеки Java 2D реализуют интерфейс *Shape*, определяющий поведение некоторой геометрической формы. Методы, определенные в интерфейсе *Shape*, приведены в таблице 6.

Таблица 6 Методы интерфейса *Shape*

Название метода	Описание
<i>boolean contains(double x, double y)</i> <i>boolean contains(Point2D p)</i>	Проверяет, находится ли точка (заданная координатами (x, y) или объектом <i>p</i> ) внутри границ геометрической фигуры
<i>boolean contains(double x, double y, double w, double h)</i> <i>boolean contains(Rectangle2D r)</i>	Проверяет, находится ли прямоугольная область (заданная левой верхней точкой с координатами (x, y), шириной <i>w</i> и высотой <i>h</i> или объектом <i>r</i> ) полностью внутри границ геометрической фигуры

<i>Rectangle</i> <b>getBounds()</b> <i>Rectangle2D</i> <b>getBounds2D()</b>	Возвращают объект прямоугольника, полностью описывающего геометрическую фигуру. Поскольку координаты прямоугольника типа <i>Rectangle</i> целочисленные, а для типа <i>Rectangle2D</i> вещественные, то второй метод дает более точные результаты
<i>PathIterator</i> <b>getPathIterator()</b> <i>AffineTransform at)</i> <i>PathIterator</i> <b>getPathIterator()</b> <i>AffineTransform at, double flatness)</i>	Возвращают ссылку на объект типа <i>PathIterator</i> , который позволяет получить доступ к границе геометрической фигуры как к последовательности фрагментов, представленных кривой Безье 1-го, 2-го или 3-го порядков.
<i>boolean intersects(double x, double y, double w, double h)</i> <i>boolean intersects(Rectangle2D r)</i>	Проверяет, пересекается ли прямоугольная область (заданная левой верхней точкой с координатами (x, y), шириной w и высотой h или объектом r) с геометрической фигурой

Классы, название которых заканчивается на '2D', являются абстрактными и описывают поведение и свойства некоторой геометрической фигуры. Для внутреннего представления координат такие классы используют значения типа *float*. Вместе с тем, многие методы в Java работают с числами типа *double*, для конвертации которых в *float* требуется явное преобразование типов. Поэтому для удобства программиста в Java 2D каждый из абстрактных классов геометрических фигур содержит внутри себя определение двух унаследованных от него вложенных классов, которые называются *Float* и *Double* и должны использоваться для создания объектов, описывающих данную геометрическую фигуру. Так, например, в рамках абстрактного класса *Line2D*, предназначенного для описания прямой линии, определено два вложенных класса, которые используют координаты линии соответствующего типа. Оба эти класса унаследованы от *Line2D* (смотри рисунок 12).

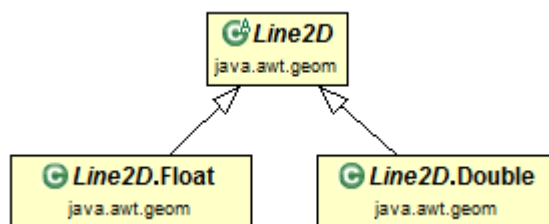


Рис 12. Класс Line2D и его вложенные классы

Для создания объектов типа *Line2D* можно использовать один из двух представленных ниже способов.

```

Line2D floatLine = new Line2D.Float(0f,0f,10f,5f);
Line2D doubleLine = new Line2D.Double(0,0,10,5);

```

Как показано в приведенных выше примерах, переменные в которых сохраняются ссылки на созданные объекты имеют тип *Line2D*, что удобно для дальнейшей работы с объектом геометрической фигуры, поскольку

использование типа (*float* или *double*) необходимо только при передаче данных в конструктор при создании объекта.

### 2.4.1 Создание сегментов прямых линий. Класс *Line2D*.

Класс *Line2D* используется для создания сегментов прямых линий. Конструкторы его вложенных классов имеют следующий вид (для типа *float* вид конструкторов аналогичен).

```
Line2D.Double(double x1, double y1, double x2, double y2);  
Line2D.Double(Point2D p1, Point2D p2);
```

Точки с координатами (*x1*, *y1*) и (*x2*, *y2*) в первом конструкторе, либо *pt1* и *pt2* во втором конструкторе задают координаты начала и конца сегмента линии.

Для изменения местоположения начальной и конечной точек ранее созданных сегментов линий используется один из трех методов *setLine*:

```
void setLine(double x1, double y1, double x2, double y2);  
void setLine(Point2D p1, Point2D p2);  
void setLine(Line2D l);
```

Новые координаты точек могут быть заданы либо аналогично тому, как это делалось для конструкторов данного класса, либо при помощи другого объекта типа *Line2D*.

Класс *Line2D* предоставляет также набор методов (смотри таблицу 7), позволяющих соотнести положение данной линии на плоскости с положением других графических объектов.

Таблица 7 Некоторые методы класса *Line2D*

Название метода	Описание
<i>boolean intersectsLine(double x1, double y1, double x2, double y2);</i> <i>boolean intersectsLine(Line2D line);</i>	Проверяет, пересекается ли данная линия с другой линией, координаты которой передаются как аргументы метода.
<i>double ptLineDist(double px, double py);</i> <i>double ptLineDist(Point2D pt);</i>	Вычисляет наикратчайшее расстояние от точки до линии, при этом линия считается продолженной до бесконечности в обе стороны
<i>double ptLineDistSq(double px, double py);</i> <i>double ptLineDistSq (Point2D pt);</i>	Вычисляет квадрат результата, возвращаемого методом <i>ptLineDist</i>
<i>double ptSegDist(double px, double py);</i> <i>double ptSegDist(Point2D pt);</i>	Вычисляет расстояние до точки от ближайшей точки сегмента линии
<i>double ptSegDistSq(double px, double py);</i> <i>double ptSegDistSq(Point2D pt);</i>	Вычисляет квадрат результата, вычисляемого методом <i>ptSegDist</i>

Класс `Line2D` также содержит статические варианты методов, представленных в таблице 7, которые принимают значения, определяющие как сегмент линии, так и объект, с которым он сопоставляется.

Пример использования класса `Line2D` приведен на рисунке 13.

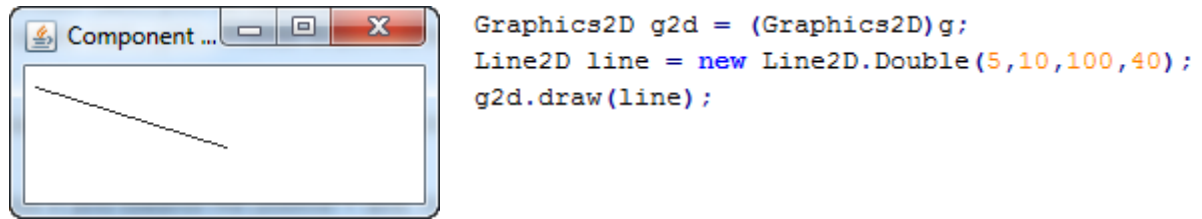


Рис 13. Пример использования класса `Line2D`

### 2.4.2 Создание сложных линий. Класс `GeneralPath`.

Класс `GeneralPath` предназначен для конструирования сложных линий, состоящих из различных сегментов. Сегментами сложной линии могут являться как линии, так и контуры произвольных геометрических фигур. Для создания сложной кривой с использованием класса `GeneralPath` необходимо следовать следующей схеме.

Шаг 1. Создаем объект класса `GeneralPath`.

```
GeneralPath path = new GeneralPath();
```

Шаг 2. Устанавливаем текущую точку с координатами (x, y).

```
path.moveTo(x, y);
```

Шаг 3. Добавляем элементы сложной линии одним из представленных ниже способов.

- Рисуем линию из текущей точки в точку (x2, y2).

```
path.lineTo(x2, y2);
```

- Рисуем кривую второго порядка из текущей точки в точку (x2, y2), используя контрольную точку (ctrlx, ctrlly).

```
path.quadTo(ctrlx, ctrlly, x2, y2);
```

- Рисуем кривую третьего порядка из текущей точки в точку (x2, y2), используя две контрольных точки (ctrlx1, ctrlly1) и (ctrlx2, ctrlly2).

```
path.curveTo(ctrlx1, ctrlly1, ctrlx2, ctrlly2, x2, y2);
```

- Добавляем к формируемой кривой границу некоторой геометрической фигуры, объект `shape` для которой был создан ранее.

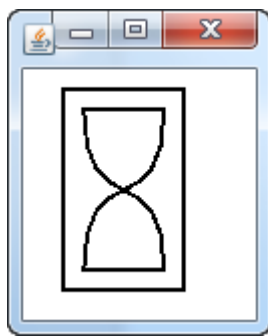
```
path.append(shape, connect);
```

Параметр 'connect' (*true/false*) определяет, будет ли граница добавляемой геометрической фигуры соединена прямой с окончанием последнего на момент ее добавления сегмента сложной линии.

Шаг 4. При необходимости соединяем точку, в которую последний раз выполнялось смещение методом *moveTo*, с концом последнего на данный момент добавленного в сложную линию сегмента, формируя тем самым замкнутый контур.

```
path.closePath();
```

Возможно многократное добавление элементов в сложную кривую путем повторения шага 3, при котором каждый раз выбирается наиболее приемлемый из перечисленных способ добавления элемента. При этом метод *moveTo* (шаг 2) также может применяться многократно при формировании сложной кривой, что наряду с применением метода *append* со вторым аргументом равным *false* дает возможность формировать сложные линии с разрывами. Пример использования класса *GeneralPath* приведен на рисунке 14.



```
GeneralPath path = new GeneralPath();  
path.moveTo(30, 20);  
path.curveTo(30, 80, 70, 40, 70, 100);  
path.lineTo(30, 100);  
path.curveTo(30, 40, 70, 80, 70, 20);  
path.closePath();  
path.append(new Line2D.Double(20,10, 80, 10), false);  
path.append(new Line2D.Double(80, 110, 20, 110), true);  
path.closePath();  
g2d.draw(path);
```

Рис 14. Пример использования класса *GeneralPath*

В примере, приведенном на рисунке 14, вызов первых пяти методов после создания объекта *path* формирует изображение песочных часов. Для этого используется два сегмента, представляющих собой кривые третьего порядка и две прямых горизонтальных линии, первая из которых создается при помощи метода *lineTo*, а вторая за счет вызова метода *closePath*. После этого формируется прямоугольная рамка, горизонтальные линии которой создаются как отдельные объекты класса *Line2D*, и затем добавляются в объект *path* путем вызова метода *append*. Вертикальные линии рамки формируются самим объектом *path*. Правая вертикальная линия создается за счет того, что при втором вызове *append*, значение его второго аргумента установлено в *true*, а левая вертикальная линия создается за счет вызова *closePath*.



### 2.4.3 Создание дуг. Класс Arc2D.

Класс *Arc2D* представляет поведение объекта, позволяющего нарисовать дугу. Конструктор класса для создания дуг с максимальным числом аргументов имеет вид.

```
Arc2D.Double(double x, double y, double w, double h,  
             double start, double extent, int type);
```

Первых четыре аргумента конструктора задают прямоугольную область рисования, внутри которой рисуется дуга. Точка  $(x, y)$  задает положение левого верхнего угла этой области, а аргументы  $w$  и  $h$ , ее ширину (ширину скругления) и высоту (высоту скругления) (смотри рисунок 15).




Рис 15. Принципы построения дуги

Аргумент *start* задает начальный угол развертки в градусах, который откладывается против часовой стрелки от горизонтальной оси декартовой системы координат, начало отсчета которой находится на пересечении диагоналей прямоугольной области рисования дуги. Аргумент *extent* задает угол развертки дуги в градусах. Аргумент *type* задает тип представления создаваемой дуги и может принимать одно из трех возможных значений, описание которых приведено в таблице 8.

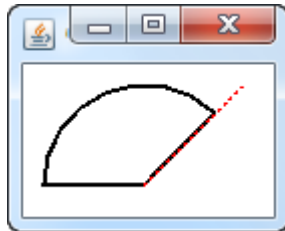
Таблица 8 Возможные значения, задающие тип дуги

Значение	Вид	Описание
<i>Arc2D.CHORD</i>		Тип дуги, при котором ее начальная и конечная точки соединены прямой линией, образуя таким образом замкнутый контур
<i>Arc2D.OPEN</i>		Тип дуги, когда ее начало и окончание ничем не соединены.



<i>Arc2D.PIE</i>		Тип дуги, при котором рисуются прямые линии от начала и конца дуги до центра, соответствующей ей прямоугольной области
------------------	---	--

Пример, демонстрирующий создание и прорисовку дуги окружности со стартовым углом  $45^\circ$  и углом развертки  $135^\circ$  приведен на рисунке 16.



```
Graphics2D g2d = (Graphics2D)g;

double x=10, y=10;
double w=100, h=100;
double start = 45, extend = 135;

g2d.setStroke(new BasicStroke(2));
g2d.setColor(Color.black);
Arc2D arc = new Arc2D.Double(x, y, w, h,
                             start, extend, Arc2D.PIE);
g2d.draw(arc);

float[] dash = new float[] {2,2};
g2d.setStroke(new BasicStroke(1,
                              BasicStroke.CAP_BUTT, BasicStroke.JOIN_BEVEL,
                              10f, dash, 0));

g2d.setColor(Color.red);
g2d.draw(new Line2D.Double(x+w/2, y+h/2,
                           x+w/2+h/2, y));
```

Рис 16. Пример использования класса Arc2D

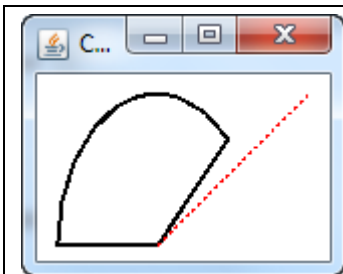
В конце приведенного в примере фрагмента программного кода дополнительно добавлена часть, рисующая пунктирную линию красного цвета под углом  $45^\circ$ , для демонстрации корректности отображения стартового угла дуги.

Для корректного отображения углов при рисовании эллиптических дуг необходимо выполнять пересчет данных углов следующим образом:

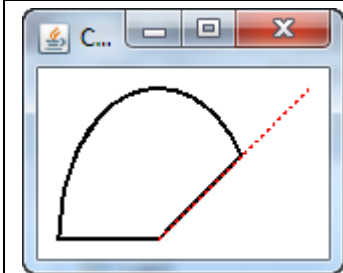
```
angle = Math.toRadians(angle); //перевод в радианы
//корректировка угла
angle = Math.atan2(Math.sin(angle)*w, Math.cos(angle)*h);
angle = Math.toDegrees(angle); //перевод в градусы
if(angle<0) angle+=360;
```

Представленная корректировка должна быть выполнена для стартового угла, а также для полного угла, рассчитанного как сумма стартового угла и угла развертки. Далее на базе скорректированных значений стартового и полного углов рассчитывается скорректированный угол развертки, после чего полученные скорректированные значения стартового угла и угла развертки сопоставляются с 0, и в случае, если они отрицательны, то к их значениям прибавляется  $360^\circ$ . Применение процедуры корректировки углов продемонстрировано на примере, представленном на рисунке 17.

```
Graphics2D g2d = (Graphics2D)g;
double x=10, y=10;
double w=100, h=150;
double start = 45, extend = 135;
g2d.setStroke(new BasicStroke(2));
g2d.setColor(Color.black);
```



```
arc = new Arc2D.Double(x, y, w, h,
                       start, extend, Arc2D.PIE);
g2d.draw(arc);
```



```
double total = start+extend;
start = Math.toDegrees(Math.atan2(
    Math.sin(Math.toRadians(start))*w,
    Math.cos(Math.toRadians(start))*h));
total = Math.toDegrees(Math.atan2(
    Math.sin(Math.toRadians(total))*w,
    Math.cos(Math.toRadians(total))*h));
extend = total-start;
if(start<0) start+=360;
if(extend<0) extend+=360;

arc = new Arc2D.Double(x, y, w, h,
                       start, extend, Arc2D.PIE);
g2d.draw(arc);
```

```
float[] dash = new float[] {2,2};
g2d.setStroke(new BasicStroke(1,
    BasicStroke.CAP_BUTT, BasicStroke.JOIN_BEVEL, 10f, dash, 0));
g2d.setColor(Color.red);
g2d.draw(new Line2D.Double(x+w/2, y+h/2, x+w/2+h/2, y));
```

Рис 17. Применение корректировки углов для правильного отображения эллиптической дуги на рисунке

Как видно из рисунка 17 применение процедуры корректировки углов привело к рисованию (смотри второй снимок экрана) эллиптической дуги со стартовым углом, соответствующим  $45^\circ$  (линия от центра области рисования до начала дуги совпадает с прямой, нарисованной под углом  $45^\circ$ ), чего не наблюдалось без пересчета углов (смотри первый снимок экрана на рисунке 17).

#### 2.4.4 Создание эллипсов. Класс *Ellipse2D*.

Класс *Ellipse2D* может использоваться для рисования эллипсов и, в частном случае, окружностей. Для создания объектов, представляющих эллипсы, может быть использован конструктор вида:

```
Ellipse2D.Double(double x, double y, double w, double h);
```

Параметры конструктора задают прямоугольную область рисования, описывающую создаваемый эллипс. Точка  $(x, y)$  задает положение левого верхнего угла этой области, а аргументы  $w$  и  $h$ , ее ширину и высоту. Для создания окружности необходимо использовать равные значения  $w$  и  $h$ . Примеры использования класса *Ellipse2D* представлены на рисунке 18.

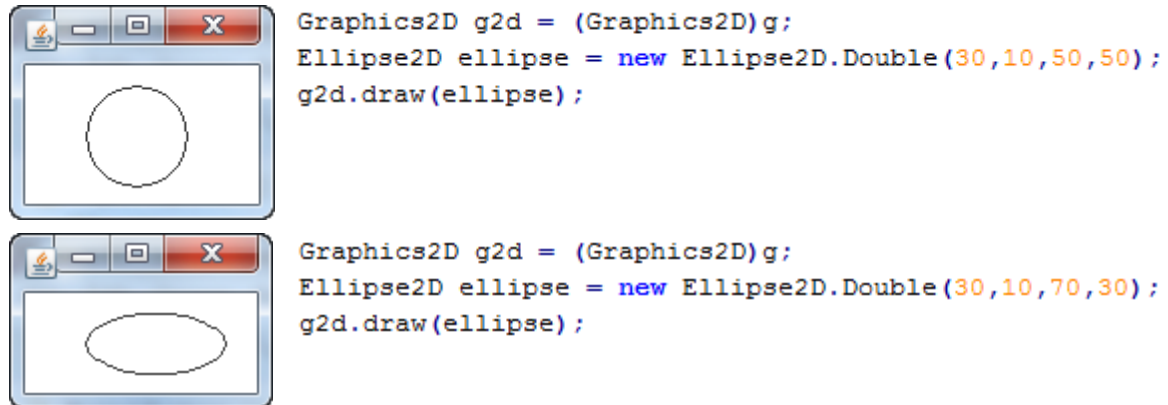


Рис 18. Примеры использования класса *Ellipse2D*

#### 2.4.5 Создание прямоугольников. Класс *Rectangle2D*.

Класс *Rectangle2D* описывает прямоугольник, заданный координатами левого верхнего угла  $(x, y)$  и своей шириной  $w$  и высотой  $h$ , которые могут быть использованы как аргументы конструктора вида:

```
Rectangle2D.Double(double x, double y, double w, double h);
```

Пример построения прямоугольника на рисунке 19.

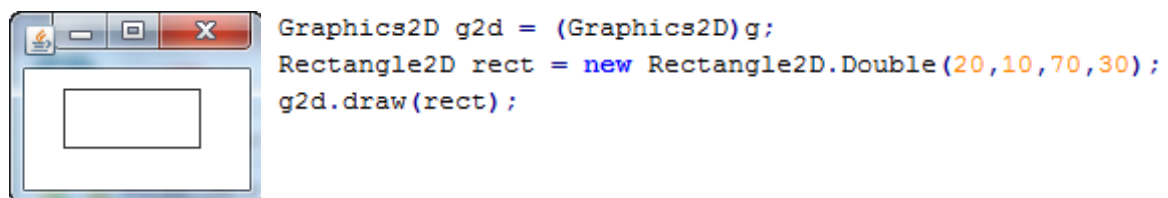


Рис 19. Пример построения прямоугольника с использованием класса *Rectangle2D*

Для изменения местоположения и разметов ранее созданного объекта класса *Rectangle2D* можно использовать один из вариантов метода *setRect*:

```
void setRect(double x, double y, double w, double h);
void setRect(Rectangle2D r);
```

Первый вариант метода принимает такие же аргументы, как и приведенный выше конструктор класса, второй имеет лишь один аргумент, который должен содержать ссылку на объект типа *Rectangle2D*.

### 2.4.6 Создание сложных геометрических фигур. Класс *Area*.

Класс *Area* позволяет создавать сложные фигуры, состоящие только из набора замкнутых областей, которые могут быть сформированы в результате наложения более простых фигур (прямоугольников, эллипсов, многоугольников и т.д.). Конструкторы данного класса имеют вид:

```
Area() ;  
Area(Shape s) ;
```

Конструктор без аргументов создает пустую область, не содержащую ни одной точки, а конструктор с одним аргументом позволяет создать область, включающую все точки геометрической фигуры *s*.

Для наложения геометрических фигур класс *Area* предоставляет методы, описанные в таблице 9.

Таблица 9 Методы класса *Area* для наложения геометрических фигур.

Название метода	Описание
<i>void add</i> ( <i>Area rhs</i> )	формирует область, которая содержит все точки исходной области и геометрической фигуры <i>rhs</i>
<i>void subtract</i> ( <i>Area rhs</i> )	формирует область, в которой сохраняются все точки исходной области, не входящие в геометрическую фигуру <i>rhs</i>
<i>void intersect</i> ( <i>Area rhs</i> )	формирует область, которая содержит только те точки, которые принадлежат одновременно исходной области и геометрической фигуре <i>rhs</i>
<i>void exclusiveOr</i> ( <i>Area rhs</i> )	формирует область, которая содержит все точки исходной области и геометрической фигуры <i>rhs</i> , которые одновременно не принадлежат им обеим

Аргумент типа *Area* для методов, приведенных в таблице, обычно создается на базе объектов геометрических фигур Java 2D с использованием конструктора класса *Area* с одним аргументом.

Использование описанных выше операций класса *Area* для наложения эллипса и прямоугольника и вид получившихся в результате областей представлены на рисунке 20. Для большей наглядности контуры исходных фигур, на базе которых строится более сложная фигура, изображены пунктирной линией. Внутреннее пространство получившейся сложной фигуры залито зеленым цветом.

```
Graphics2D g2d = (Graphics2D)g;
Rectangle2D rect = new Rectangle2D.Double(20,10,70,30);
Ellipse2D ellipse = new Ellipse2D.Double(45,20,70,30);

float[] dash = new float[] {2,2};
g2d.setStroke(new BasicStroke(1, BasicStroke.CAP_BUTT,
                             BasicStroke.JOIN_BEVEL, 10f, dash, 0));
g2d.setColor(Color.gray);
g2d.draw(rect);
g2d.draw(ellipse);
```

	<pre>Area area = new Area(rect); area.add(new Area(ellipse));  g2d.setColor(Color.green); g2d.fill(area);</pre>
	<pre>Area area = new Area(rect); area.subtract(new Area(ellipse));  g2d.setColor(Color.green); g2d.fill(area);</pre>
	<pre>Area area = new Area(rect); area.intersect(new Area(ellipse));  g2d.setColor(Color.green); g2d.fill(area);</pre>
	<pre>Area area = new Area(rect); area.exclusiveOr(new Area(ellipse));  g2d.setColor(Color.green); g2d.fill(area);</pre>

Рис 20. Применение класса *Area* для комбинирования геометрических фигур

## 2.5 Рисование текста.

**Рисование текста** выполняется методом класса *Graphics2D* *drawString*:

```
void drawString(String str, int x, int y);
```

где аргумент *str* задает строку, которую необходимо нарисовать, а аргументы *x* и *y* определяют координаты, где будет располагаться точка, соответствующая началу строки по горизонтали и базовой линии рисуемого текста по вертикали.

Для того чтобы правильно задавать местоположение строки в рамках области, доступной для рисования, используя метод *drawString*, рассмотрим ряд величин, характеризующих рисуемую строку (рисунок 21):



Рис 21. Иллюстрация основных терминов, применяемых при наборе текста

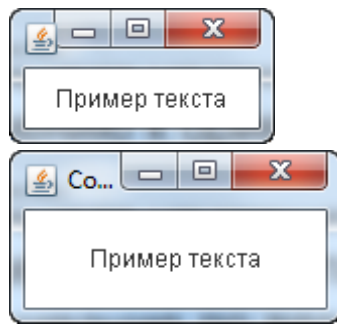
- *базовая линия (baseline)* – это воображаемая линия, которая касается снизу букв в строке (без учета частей некоторых букв, таких, например, как ‘p’, выступающих вниз);
- *максимальная высота символов (ascent)* – это расстояние от базовой линии до верхней части высоких символов, таких как ‘b’, ‘k’ и т.д., или символов верхнего регистра;
- *максимальная глубина посадки символов (descent)* – это расстояние от базовой линии до нижней части низких символов, таких как p, g и т.д.;
- *интерлиньяж* – это расстояние между максимальной глубиной посадки символов предыдущей строки и максимальной высотой символов следующей строки;
- *высота строки* – это расстояние между последовательными базовыми линиями, то есть  $ascent + descent + \text{интерлиньяж}$ .

Как следует из рассмотренных характеристик рисуемого текста для вычисления положения по вертикали базовой линии строки относительно ее верхнего края необходимо к координате верхнего края прибавить максимальную высоту символов (ось Y направлена вниз).

Для позиционирования строки в рамках области рисования необходимо знать высоту и ширину прямоугольника, описывающего рисуемую строку, которые зависят от трех факторов:

1. содержания выводимой строки (т.е. того, какие именно символы она включает);
2. шрифта, используемого для начертания символов (объект класса *Font*);
3. характеристик устройства, на котором рисуется строка (экран, принтер и т.д.) (объект класса *FontRenderContext*).

Объекты, указанные выше во втором и третьем пунктах, могут быть получены напрямую из объекта класса *Graphics2D*. Программный код, показывающий как получить прямоугольник, описывающий рисуемую строку, и использующий его для центрирования рисуемой строки в рамках области рисования приведен на рисунке 22.



```
String str = "Пример текста";
//Получаем текущий шрифт
Font f = g2d.getFont();
//Получаем объект, характеризующий
//устройство для рисования
FontRenderContext context =
    g2d.getFontRenderContext();
//Вычисляем прямоугольник, описывающий строку
Rectangle2D strRect =
    f.getStringBounds(str, context);
//Вычисляем координату начала строки
//центрированной по горизонтали
double strX =
    (getWidth() - strRect.getWidth())/2;
//Вычисляем координату базовой линии строки
//центрированной по вертикали
double strY =
    (getHeight() - strRect.getHeight())/2 -
    strRect.getY();
//Рисуем строку
g2d.drawString(str, (int)strX, (int)strY);
```

Рис 22. Расположение рисуемого текста по центру области рисования

В системе координат, связанной с прямоугольником строки, начало отсчета по вертикали находится на базовой линии, ось Y направлена вниз, а верхняя граница прямоугольника отрицательна, т.к. находится на величину *strRect.getY()* выше начала отсчета. Поэтому положение базовой линии строки по вертикали вычисляется как позиция верхней стороны отцентрированного прямоугольника плюс максимальная высота символов, которая равна *strRect.getY()* с обратным знаком.

Для изменения шрифта, которым рисуются символы строки, можно использовать метод *setFont*, класса *Graphics2D*, принимающий в качестве аргумента ссылку на объект типа *Font*. Для создания таких объектов можно использовать конструктор класса *Font* следующего вида.

```
Font(String name, int style, int size);
```

Аргумент *name* задает имя шрифта (например, "Arial Bold"), либо имя семейства шрифтов (например, "Arial"), *style* – его стиль, а *size* определяет его размер в пунктах.

Поскольку на различных компьютерах могут присутствовать различные наборы шрифтов, то шрифта с определенным именем на компьютере может не оказаться. Для решения этой проблемы в библиотеке *AWT* определено пять логических имен шрифтов, которые всегда связываются со шрифтами, действительно присутствующими на компьютере пользователя:

- *SansSerif*



- *Serif*
- *Monospaced*
- *Dialog*
- *DialogInput*

Например, в ОС Windows логическое имя *SansSerif* будет связано со шрифтом *Arial*. При использовании в качестве имени названия семейства, конкретный шрифт ищется с учетом параметра *style*.

В качестве значений для параметра *style* могут использоваться следующие константы:

- *Font.PLAIN* – обычный;
- *Font.BOLD* – жирный;
- *Font.ITALIC* – курсив;
- *Font.BOLD + Font.ITALIC* – жирный курсив.

Пример рисования строки с использованием установленного программно логического шрифта приведен на рисунке 23.

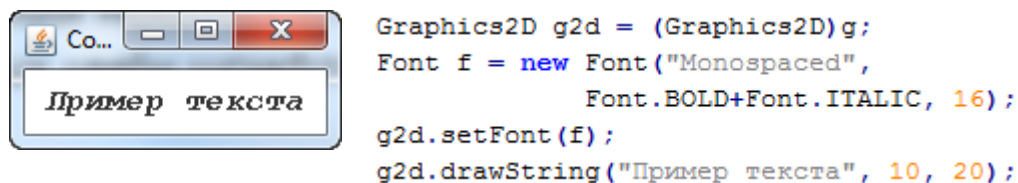


Рис 23. Пример задания шрифта для рисования текста

Кроме использования логических имен шрифтов в Java есть поддержка шрифтов *TrueType*. Для использования таких шрифтов первым шагом создается поток ввода для загрузки шрифта, например, из файла с расширением .ttf. Далее на базе этого потока при помощи статического метода *Font.createFont* создается объект класса *Font*, который будет представлять шрифт размером 1pt. Для создания шрифта другого размера используется метод *deriveFont*, определенный в классе *Font*. Пример использования шрифта *TrueType* представлен на рисунке 24.

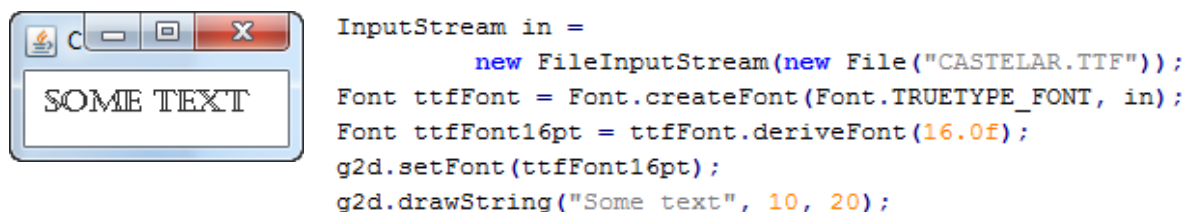


Рис 24. Пример использования шрифта *TrueType* для рисования текста

## 2.6 Преобразование изображений. Класс *AffineTransform*.

С целью упрощения рисования изображения компонента, либо реализации различных режимов его отображения могут использоваться преобразования, которые позволяют:

- изменять ориентацию изображения на плоскости;
- изменять размер изображения;
- изменять положение изображения в рамках области рисования;
- изменять форму изображения.

Из теории матриц известно, что любые преобразования координат на плоскости могут быть представлены в виде единого аффинного преобразования (*affine transformation*) вида:

$$\begin{bmatrix} x_{new} \\ y_{new} \\ 1 \end{bmatrix} = \begin{bmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix},$$

где  $(x, y)$  – координаты точки до преобразования,  $(x_{new}, y_{new})$  – координаты точки после преобразования,  $a, b, c, d, e, f$  – коэффициенты, от значений которых зависит вид преобразования.

Представленное выше матричное преобразование в Java 2D API описывается с помощью класса *AffineTransform*. При известных коэффициентах преобразования его объект может быть непосредственно создан с использованием конструктора данного класса следующим образом.

```
AffineTransform t = new AffineTransform(a,b,c,d,e,f);
```

Использовать объект  $t$  аффинного преобразования при рисовании с помощью объекта  $g2d$  класса *Graphics2D* можно двумя способами:

- заменив им текущее преобразование в объекте  $g2d$ , вызвав метод *setTransform*:

```
g2d.setTransform(t);
```

- скомбинировав данное преобразование с текущим преобразованием в объекте  $g2d$ , вызвав метод *transform*:

```
g2d.transform(t);
```

Для четырех стандартных преобразований (масштабирование, вращение, смещение, сдвиг) дополнительно предусмотрены:

1. отдельный статический метод в классе *AffineTransform* для создания объекта преобразования;
2. отдельный метод в классе *AffineTransform*, позволяющий сделать ранее созданное преобразование, преобразованием данного типа;
3. отдельный метод в классе *Graphics2D*, для добавления данного преобразования без использования отдельного объекта.

Конкретные названия указанных выше методов будут приведены для каждого стандартного преобразования ниже в соответствующих подразделах.

Для использования аффинного преобразования только для части рисуемых элементов изображения необходимо:

1. сохранить исходное преобразование, для получения ссылки на которое используется метод *getTransform*, определенный в классе *Graphics2D*;
2. добавить/установить в объект класса *Graphics2D* новое преобразование;
3. выполнить рисование части элементов изображения;
4. восстановить исходное преобразование.

Описанная выше схема продемонстрирована в следующем фрагменте программного кода.

```
Graphics2D g2d = (Graphics2D)g;  
AffineTransform oldTransform = g2d.getTransform(); //сохранение  
//создание дополнительного преобразования  
AffineTransform t = new AffineTransform(...);  
g2d.transform(t); //изменение преобразования в g2d  
//Рисование с учетом измененного преобразования  
...  
g2d.setTransform(oldTransform); //восстановление
```

### 2.6.1 Преобразование масштабирования.

Преобразование масштабирования изменяет размеры рисуемой геометрической фигуры.

Отдельный объект для такого преобразования может быть создан следующими способами:

```
AffineTransform scaling;  
//С использованием статического метода  
scaling = AffineTransform.getScaleInstance(sx, sy);  
//Путем изменения типа преобразования для ранее созданного объекта  
scaling = new AffineTransform();  
scaling.setToScale(sx, sy);
```

где *sx* и *sy* задают коэффициенты масштабирования по осям X и Y, соответственно.

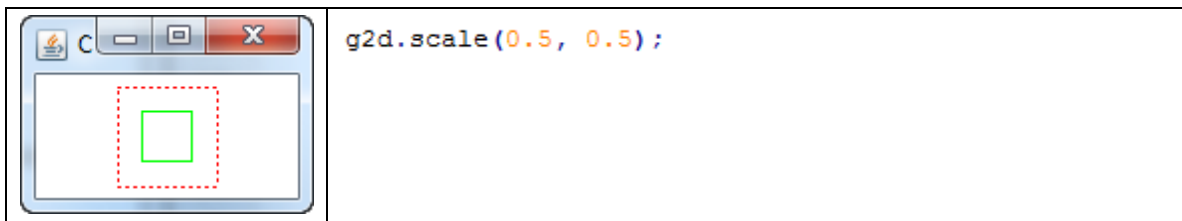
Для добавления преобразования масштабирования к объекту *g2d* типа *Graphics2D* используется метод *scale*:

```
g2d.scale(sx, sy);
```

Пример программного кода, выполняющий преобразование масштабирования для квадрата приведен на рисунке 25.

```
Graphics2D g2d = (Graphics2D)g;
//Смещаем начало координат в центр компонента
g2d.translate(getWidth()/2,getHeight()/2);

Rectangle2D rect = new Rectangle2D.Double(-25,-25,50,50);
float[] dash = new float[] {2,2};
g2d.setStroke(new BasicStroke(1, BasicStroke.CAP_BUTT,
                             BasicStroke.JOIN_BEVEL, 10f, dash, 0));
g2d.setColor(Color.red);
g2d.draw(rect);
```



```
g2d.scale(0.5, 0.5);
```

```
g2d.setStroke(new BasicStroke(1));
g2d.setColor(Color.green);
g2d.draw(rect);
```

Рис 25. Применение преобразования масштабирования

В приведенном примере красной пунктирной линией нарисован исходный квадрат, а зеленой сплошной линией квадрат после применения преобразования.

### 2.6.2 Преобразование вращения.

Преобразование вращения поворачивает геометрическую фигуру на некоторый угол вокруг точки с координатами (x, y). По умолчанию x=y=0 (верхний левый угол области рисования).

Отдельный объект для такого преобразования может быть создан следующими способами:

```
AffineTransform rotation;
//С использованием статического метода
rotation = AffineTransform.getRotateInstance(alpha, x, y);
//Путем изменения типа преобразования для ранее созданного объекта
rotation = new AffineTransform();
rotation.setToRotate(alpha, x, y);
```

где *alpha* задает угол вращения в радианах по часовой стрелке.

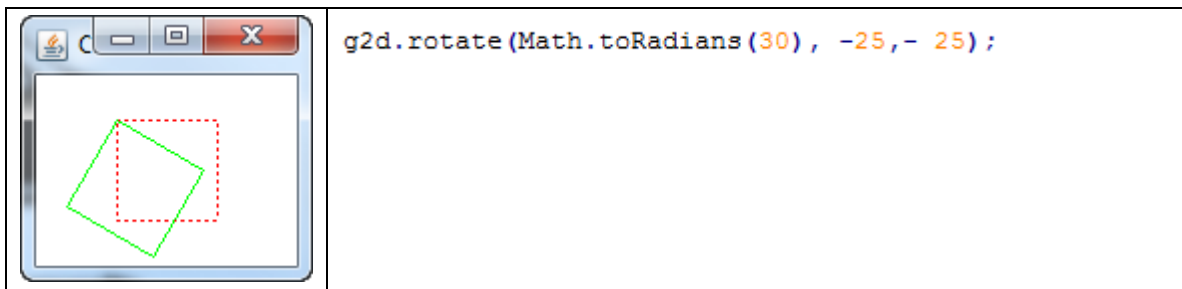
Для добавления преобразования вращения к объекту *g2d* типа *Graphics2D* используется метод *rotate*:

```
g2d.rotate(alpha, x, y);
```

Пример программного кода, выполняющий преобразование вращения для квадрата приведен на рисунке 26.

```
Graphics2D g2d = (Graphics2D)g;
//Смещаем начало координат в центр компонента
g2d.translate(getWidth()/2,getHeight()/2);

Rectangle2D rect = new Rectangle2D.Double(-25,-25,50,50);
float[] dash = new float[] {2,2};
g2d.setStroke(new BasicStroke(1, BasicStroke.CAP_BUTT,
                             BasicStroke.JOIN_BEVEL, 10f, dash, 0));
g2d.setColor(Color.red);
g2d.draw(rect);
```



```
g2d.setStroke(new BasicStroke(1));
g2d.setColor(Color.green);
g2d.draw(rect);
```

Рис 26. Применение преобразования вращения

В приведенном примере красной пунктирной линией нарисован исходный квадрат, а зеленой сплошной линией квадрат после применения преобразования.

### 2.6.3 Преобразование смещения.

Преобразование смещения позволяет сдвинуть рисуемую фигуру на определенное расстояние  $tx$  по оси X и на расстояние  $ty$  по оси Y.

Отдельный объект для такого преобразования может быть создан следующими способами:

```
AffineTransform translation;
//С использованием статического метода
translation = AffineTransform.getTranslateInstance(tx, ty);
//Путем изменения типа преобразования для ранее созданного объекта
translation = new AffineTransform();
translation.setToTranslation(tx, ty);
```

Для добавления преобразования смещения к объекту  $g2d$  типа *Graphics2D* используется метод *translate*:

```
g2d.translate(tx, ty);
```

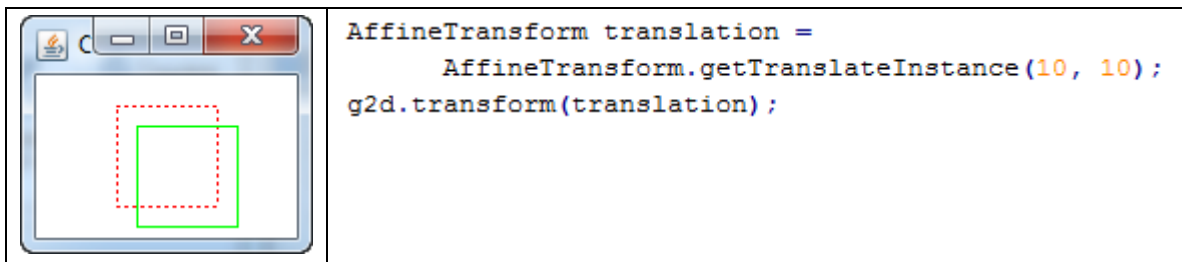
Пример программного кода, выполняющий преобразование смещения для квадрата приведен на рисунке 27.

```

Graphics2D g2d = (Graphics2D)g;
//Смещаем начало координат в центр компонента
g2d.translate(getWidth()/2,getHeight()/2);

Rectangle2D rect = new Rectangle2D.Double(-25,-25,50,50);
float[] dash = new float[] {2,2};
g2d.setStroke(new BasicStroke(1, BasicStroke.CAP_BUTT,
                             BasicStroke.JOIN_BEVEL, 10f, dash, 0));
g2d.setColor(Color.red);
g2d.draw(rect);

```



```

g2d.setStroke(new BasicStroke(1));
g2d.setColor(Color.green);
g2d.draw(rect);

```

Рис 27. Применение преобразования смещения

В приведенном примере красной пунктирной линией нарисован исходный квадрат, а зеленой сплошной линией квадрат после применения преобразования.

#### 2.6.4 Преобразование сдвига.

Преобразование сдвига смещает все точки геометрической фигуры вдоль оси X на расстояние, равное расстоянию этих точек от начала координат по оси Y, умноженное на коэффициент  $shx$ . По оси Y точки смещаются на расстояние, равное расстоянию этих точек от начала координат по оси X, умноженное на коэффициент  $shy$ .

Отдельный объект для такого преобразования может быть создан следующими способами:

```

AffineTransform shear;
//С использованием статического метода
shear = AffineTransform.getShearInstance(shx, shy);
//Путем изменения типа преобразования для ранее созданного объекта
AffineTransform shear = new AffineTransform();
shear.setToShear(shx, shy);

```

Для добавления преобразования сдвига к объекту *g2d* типа *Graphics2D* используется метод *shear*:

```

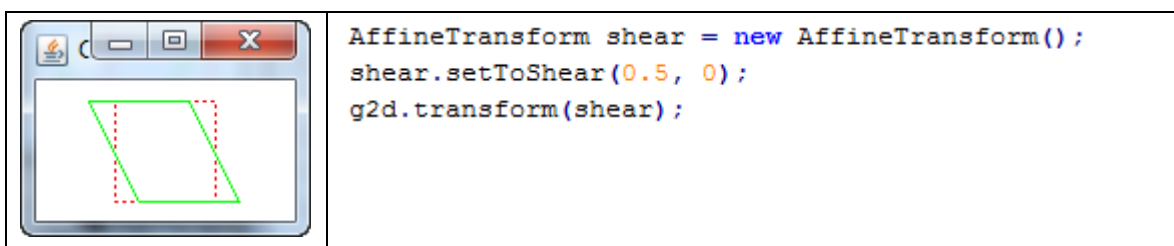
g2d.shear(shx, shy);

```



Пример программного кода, выполняющий преобразование сдвига для квадрата приведен на рисунке 28.

```
Graphics2D g2d = (Graphics2D)g;  
//Смещаем начало координат в центр компонента  
g2d.translate(getWidth()/2,getHeight()/2);  
  
Rectangle2D rect = new Rectangle2D.Double(-25,-25,50,50);  
float[] dash = new float[] {2,2};  
g2d.setStroke(new BasicStroke(1, BasicStroke.CAP_BUTT,  
    BasicStroke.JOIN_BEVEL, 10f, dash, 0));  
g2d.setColor(Color.red);  
g2d.draw(rect);
```



```
g2d.setStroke(new BasicStroke(1));  
g2d.setColor(Color.green);  
g2d.draw(rect);
```

Рис 28. Применение преобразования сдвига

В приведенном примере красной пунктирной линией нарисован исходный квадрат, а зеленой сплошной линией квадрат после применения преобразования.

## 2.7 Использование фигур усечения и правил композиции.

Кроме использования аффинных преобразований на то, как будет выглядеть отдельный рисуемый элемент в рамках итогового изображения, могут оказывать влияние следующие настройки объекта холста типа *Graphics2D*:

1. используемая в момент рисования *фигура усечения* (объект типа *Shape*);
2. установленные в момент рисования *правила композиции* (объект типа *Composite*).

**Фигура усечения** определяет подобласть в рамках области рисования интерфейсного компонента, вне которой изображение будет оставаться неизменным.

Объект класса *Graphics2D* предоставляет следующие методы для работы с фигурой усечения:

- для получения текущей фигуры усечения используется метод *getClip*;
- для установки в качестве текущей фигуры усечения результата ее пересечения с некоторой новой фигурой используется метод *clip*;



- для замены текущей фигуры усечения на новую используется метод *setClip*.

**Правила композиции** представлены объектом типа *Composite* и определяют каким образом будут комбинироваться цвет (а точнее красная, зеленая и синяя его составляющие) и значение прозрачности (задается в Java 2D дополнительно к составляющим цвета величиной *альфа-канала* (*alpha channel*)) отдельных точек рисуемой геометрической фигуры с цветом и прозрачностью пикселей уже существующего изображения.

Объект класса *Graphics2D* предоставляет следующие методы для работы с правилами композиции:

- для получения текущих правил композиции используется метод *getComposite*;
- для установки нового объекта правил композиции используется метод *setComposite*.

Интерфейс *Composite* реализован в классе *AlphaComposite*, который и используется для создания объектов правил композиции. Класс *AlphaComposite* содержит ряд констант, задающих определенную схему слияния цветов и прозрачности пикселей, на основании которых можно создавать объекты данного класса с использованием статического метода *getInstance*, как показано в следующем примере.

```
Composite composite = AlphaComposite.getInstance(AlphaComposite.SRC);
```

Более подробную информацию о правилах комбинирования пикселей при наложении изображений можно найти в справочной документации.

### 3 Построение многопоточных приложений

*Поток выполнения* представляет собой последовательность инструкций, выполняющуюся в контексте процесса, и использующую его ресурсы совместно с другими потоками.

При запуске Java-приложения немедленно создается и запускается на выполнение поток, называемый *главным потоком приложения*. Данному потоку присваивается имя “*main*”, в нем запускается метод *main()*, а также все вызываемые из него методы. Особая роль данного потока состоит в следующем:

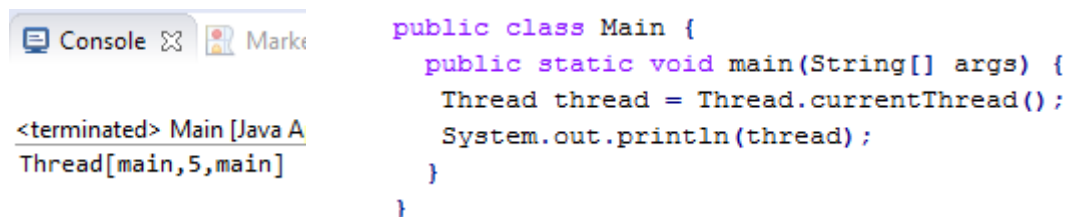
- из главного потока порождаются другие потоки приложения, которые называются *дочерними*;
- главный поток является последним потоком с завершением работы которого завершается работа программы.

### 3.1 Представление потоков выполнения в Java. Класс *Thread*.

Для доступа к потокам и управления ими в библиотеке Java используется класс *Thread*.

Каждому отдельному потоку соответствует объект данного класса, доступ к которому может быть получен из кода, выполняющегося внутри потока, путем вызова статического метода *currentThread*.

Для главного потока объект класса *Thread* создается автоматически при запуске программы. Фрагмент кода, представленный ниже получает ссылку на объект класса *Thread* для главного потока приложения и выводит в консоль его строковое представление.



```
public class Main {
    public static void main(String[] args) {
        Thread thread = Thread.currentThread();
        System.out.println(thread);
    }
}
```

Значения в квадратных скобках соответствуют (в порядке их отображения) имени потока, его приоритету и имени группы, к которой принадлежит данный поток.

Поток, описываемый классом *Thread*, может быть охарактеризован рядом величин, представленных ниже.

*Имя потока* изначально задается при создании потока с возможностью последующего изменения. Имя может быть явно указано как аргумент конструктора класса *Thread*. При использовании конструкторов, не имеющих данного аргумента, имя генерируется автоматически в соответствии с шаблоном:

```
String name = "Thread" + n;
```

где *n* является целым числом. Имя потока может использоваться для удобства идентификации потоков, но не является уникальным. Для получения имени потока используется метод *getName*, а для его задания метод *setName*.

*Уникальный идентификатор потока* присваивается потоку системой при его создании и не изменяется до завершения работы потока. После завершения работы потока этот же идентификатор может использоваться для присвоения другому вновь создаваемому потоку. Пока поток существует гарантируется что данное число будет уникальным среди других потоков выполнения. Получить идентификатор потока можно путем вызова метода *getId*.

*Приоритет потока* используется планировщиком потоков для принятия решений о том, когда необходимо разрешать выполнение тому или иному потоку. Значение данной характеристики может изменяться в диапазоне от 1 до

10. С целью более осмысленной работы с этими значениями для некоторых из них в классе *Thread* зарезервированы специальные статические константы:

- *Thread.MIN\_PRIORITY* – минимальный приоритет (эквивалентно 1);
- *Thread.NORM\_PRIORITY* – нормальный приоритет (эквивалентно 5);
- *Thread.MAX\_PRIORITY* – максимальный приоритет (эквивалентно 10).

По умолчанию приоритет потока устанавливается равным приоритету родительского потока. В дальнейшем приоритет может быть изменен с использованием метода *setPriority*. Для получения значения приоритета потока используется метод *getPriority*.

*Состояние потока* отражает этап, на котором находится выполнение потока после его создания. Возможные состояния потока представлены набором констант, заданных в виде перечисления *Thread.State*, которые приведены в таблице 10.

Таблица 10 Константы состояний потока

Название	Описание
<i>Thread.State.NEW</i>	состояние потока, который был создан, но еще не был запущен на выполнение
<i>Thread.State.RUNNABLE</i>	поток был запущен на выполнение
<i>Thread.State.BLOCKED</i>	поток находится в ожидании обращения к синхронизированному ресурсу, который в данный момент занят другим потоком
<i>Thread.State.WAITING</i>	поток находится в ожидании неопределенное время пока другой поток выполнит некоторое действие
<i>Thread.State.TIMED_WAITING</i>	поток находится в ожидании пока другой поток выполнит некоторое действие до определенного момента времени
<i>Thread.State.TERMINATED</i>	поток закончил выполнение

Состояние потока может быть получено путем вызова метода *getState*. Еще одним методом, который связан с состоянием потока является метод *isAlive()*. Данный метод возвращает *true* если поток был запущен на выполнение и еще не завершил свою работу.

*Тип потока* определяет принадлежность потока к одной из двух категорий:

- *пользовательские потоки* обычно предназначены для решения некоторой отдельной подзадачи в рамках приложения, не связанной с подзадачами других потоков;
- *потоки-демоны* являются сервисными потоками, единственной целью которых является определенного рода обслуживание других потоков.

Примерами потоков-демонов могут служить поток “сборщика мусора”, который освобождает оперативную память от более неиспользуемых объектов, а также поток таймера, который через определенное время посылает сигналы другим потокам. Без присутствия пользовательских потоков, когда в приложении остаются запущенными только потоки-демоны, их дальнейшее выполнение не имеет смысла и по этой причине такая программа завершает работу. Тип потока можно определить, вызвав метод *isDaemon* объекта класса *Thread*, который возвращает значение *true*, для потока-демона и *false* для пользовательского потока. Задать тип потока можно используя метод *setDaemon*.

### 3.2 Способы создания потоков выполнения.

Для того чтобы запустить собственный код в отдельном потоке в рамках Java приложения необходимо выполнение следующих условий:

1. класс, часть кода которого будет работать в отдельном потоке, должен реализовывать интерфейс *Runnable*, определяющий единственный метод *run*;
2. метод *run* должен быть переопределен и именно он должен содержать код, который будет работать в отдельном потоке;
3. необходимо задействовать объект класса *Thread* для создания нового потока и запуска содержимого метода *run*.

#### 3.2.1 Использование наследования от класса *Thread*.

Поскольку сам класс *Thread* уже реализует интерфейс *Runnable*, а также позволяет организовать и запустить на выполнение отдельный поток, то для запуска собственного кода в отдельном потоке можно поступить следующим образом:

1. создать новый класс, унаследованный от класса *Thread*;
2. переписать в новом классе метод *run*, разместив в нем код, который надо выполнить в отдельном потоке;
3. создать новый поток путем создания объекта нового класса и запустить его на выполнение, вызвав из созданного объекта метод *start*.

Применение описанного выше подхода продемонстрировано в следующем фрагменте программного кода.

```

public class CustomThread extends Thread {
    //Поля класса
    ...

    public void run() {
        //код для запуска в отдельном потоке
        ...
    }

    //Другие методы класса
    ...
}
...
Thread thread = new CustomThread(); //Создание потока
thread.start(); //Запуск потока на выполнение

```

Основным недостатком представленного способа запуска собственного кода в отдельном потоке является необходимость наследования от класса *Thread*. В случае если собственный класс уже является частью иерархии классов и имеет некоторый суперкласс, то дополнительно унаследовать его от класса *Thread* невозможно, поскольку Java не поддерживает множественное наследование.

### ***3.2.2 Использование класса, реализующего интерфейс *Runnable*, без наследования от класса *Thread*.***

В качестве альтернативы прямому наследованию собственного класса от класса *Thread* можно использовать *подход, основанный на прямой реализации в собственном классе интерфейса *Runnable**. При данном подходе необходимо проделать следующее:

1. создать новый класс, реализующий интерфейс *Runnable*;
2. переписать в новом классе метод *run*, разместив в нем код, который надо выполнить в отдельном потоке;
3. в конструкторе нового класса создать объект потока используя конструктор класса *Thread*, в который одним из аргументов передать ссылку *this* на объект собственного класса, и запустить поток на выполнение, вызвав из созданного объекта потока метод *start*;
4. создать объект нового класса.

Использование описанного альтернативного подхода продемонстрировано в следующем фрагменте программного кода.

```

public class CustomClass implements Runnable {
    //Поля класса
    private Thread thread;
    ...
    public CustomClass() {
        this.thread = new Thread(this); //Создание потока
        thread.start(); //Запуск потока на выполнение
    }
    public Thread getThread(){
        return thread;
    }
    public void run() {
        //код для запуска в отдельном потоке
        ...
    }
    // Другие методы класса
    ...
}
...
CustomClass myObject = new CustomClass(); //Объект собственного класса

```

В приведенном фрагменте кода ссылка на объект потока сохраняется в поле *thread*. После создания объекта класса *CustomClass*, объект связанного с ним потока может быть получен путем вызова метода *getThread*.

### 3.3 Организация взаимодействия потоков в многопоточном приложении.

Класс *Thread* предоставляет ряд методов, которые могут использоваться для управления созданным потоком, а также для организации взаимодействия между потоками.

#### 3.3.1 Организация прерывания выполнения потока.

Метод *interrupt* указывает потоку на то, что он должен прервать выполнение. После вызова данного метода для потока устанавливается специальный статус (флаг), указывающий на то, что он находится в состоянии прерывания. Данный статус может быть проверен путем вызова методов *Thread.interrupted()* или *isInterrupted()* для объекта потока. Различие между этими двумя методами состоит в том, что первый из них очищает статус нахождения потока в процессе прерывания, а второй нет. Если поток, для которого был вызван метод *interrupt*, находился в состоянии блокировки, вызванной методами *wait* класса *Object*, *join* либо *sleep* класса *Thread*, то статус нахождения в процессе прерывания будет очищен и для потока будет сгенерировано исключение *InterruptedException*. В большинстве случаев реакцией потока на вызов метода *interrupt* является завершение работы путем выхода из метода *run*, однако окончательное решение о реакции на данный

вызов остается за программистом, который пишет код метода *run* для потока. Пример реализации метода *run*, который реагирует на вызов *interrupt* окончанием выполнения потока имеет следующий вид.

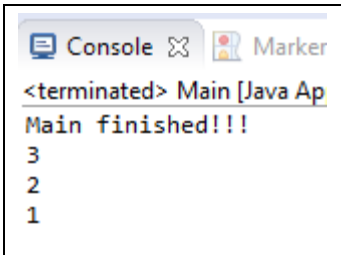
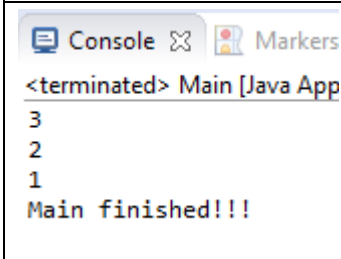
```
public void run() {
    try {
        while(!Thread.interrupted())
        {
            //Выполнение вычислений, связанных с потоком
            ...
        }
    }
    catch (InterruptedException e) {}
}
```

В приведенном примере код, написанный внутри метода *run*, периодически проверяет не находится ли поток в состоянии прерывания и если да, то работа метода *run* завершается. Для отслеживания ситуации, когда поток окажется заблокированным при вызове *interrupt*, в код добавлена обработка исключения *InterruptedException* переход к которой также приводит к окончанию работы метода *run*.

**Метод *join*** позволяет одному потоку приостановить свое выполнение до тех пор, пока не завершится выполнение другого потока. Пример использования данного метода представлен на рисунке 29.

```
public static void main(String[] args) {

    Thread th = new Thread(new Runnable() {
        public void run() {
            int i=3;
            while(i>0)
                System.out.println(i--);
        }
    });
    th.start();
```

	
	<pre>try {     th.join(); } catch (InterruptedException e) {}</pre>



---

```
System.out.println("Main finished!!!");  
}
```

Рис 29. Использование метода *join* для ожидания завершения другого потока

В коде приведенного примера, основной поток приложения создает дополнительный поток (объект *th*) и запускает его на выполнение, после чего выводит в консоль сообщение о завершении работы. Дополнительный поток в своем методе *run* выполняет цикл *while*, который выводит в консоль значения от 3 до 1, после чего работа дополнительного потока завершается. Если между запуском на выполнение дополнительного потока и выводом сообщения о завершении основного потока нет никакого дополнительного кода, то очередность вывода сообщений основным и дополнительным потоками может быть произвольной и зависит от того, как менеджер потоков управляет их активностью. При тестовом запуске, результаты которого показаны на верхнем снимке окна консоли, раньше свое сообщение вывел основной поток. Если же после запуска на выполнение вспомогательного потока добавить вызов *th.join()*, то основной поток будет приостановлен до тех пор, пока дополнительный поток не завершит свой метод *run* и лишь после этого продолжится выполнение кода в основном потоке. Данный факт подтверждается на нижнем снимке окна консоли, приведенном на рисунке 29. На этом снимке видно, что сообщения дополнительного потока появились раньше, чем сообщение основного потока, которое в коде программы выводилось сразу после вызова *th.join()*.

Кроме версии метода *join* без аргументов, в классе *Thread* так же есть вариант данного метода, который в качестве аргумента принимает время в миллисекундах. Если в этот метод передать значений 0, то он будет работать так же, как и *join* без аргументов. Если же передано положительное значение, по поток при выполнении которого был вызван метод *join* будет ожидать завершения другого потока в течение промежутка времени, не превышающего переданное значение.

### **3.3.2 Синхронизация потоков.**

В многопоточном Java-приложении может быть создано множество объектов типа *Thread*. Потоки, соответствующие этим объектам, работают в одном адресном пространстве и, соответственно, могут изменять одни и те же данные. Неупорядоченные операции считывания и записи некоторого значения одновременно несколькими потоками могут приводить к некорректным результатам даже при выполнении простейших операций. По этой причине при работе нескольких потоков с одними и теми же данными необходимо их синхронизировать.

*Синхронизацией потоков* называется механизм, гарантирующий, что одновременно доступ к совместно используемым данным в многопоточном приложении будет иметь только один поток.

Концепция синхронизации потоков в Java основана на использовании мониторов. *Монитор* – механизм блокировки, имеющийся у каждого отдельного объекта, и гарантирующий одновременный доступ к его синхронизированным методам только одного потока.

Порядок синхронизации потоков с использованием мониторов:

- если некоторый поток вызвал синхронизированный метод объекта, то говорят, что объект вошел в монитор (т.е. перешел в заблокированное состояние), а сам поток становится владельцем данного монитора;
- после завершения работы потока с синхронизированным методом говорят, что объект выходит из монитора (т.е. из заблокированного состояния).
- все другие потоки (кроме владельца монитора), пытающиеся обратиться к любому синхронизированному методу заблокированного объекта, будут приостановлены, до выхода объекта из монитора и лишь после этого владельцем монитора сможет стать очередной поток.
- обращение других потоков к несинхронизированным методам возможно даже если объект заблокирован некоторым потоком.

Для того, чтобы указать, что определенные методы класса являются синхронизированными, и тем самым обеспечить последовательный доступ к ним потоков в многопоточном приложении, в их определении используется модификатор *synchronized*:

```
public synchronized void someMethod() {  
    //Код метода  
}
```

Кроме использования синхронизированных методов для синхронизации потоков можно использовать **оператор *synchronized*** и **блоки синхронизации**. Данная возможность обычно применяется в случае, если имеется ранее разработанный класс, не содержащий синхронизированных методов, объекты которого необходимо использовать в многопоточном приложении. Пример кода, содержащего блок синхронизации, имеет вид:

```
synchronized(obj) {  
    //синхронизированные действия  
    method1(obj);  
    obj.method2();  
    ...  
}
```

Во время выполнения блока кода, соответствующего оператору *synchronized*, доступ к объекту *obj* блокируется для всех других потоков.

Для организации дополнительной взаимосвязи между потоками, когда необходимо достичь определенной очередности выполнения ими некоторых действий, можно использовать методы, определенные в классе *Object*: *wait()*, *notify()* и *notifyAll()*.

Метод *wait()* и несколько его перегруженных версий используется для приостановки выполнения потока, который вызвал данный метод. При вызове метода *wait()* внутри синхронизированного метода объекта, с него будет снята блокировка и другие потоки получают доступ к его синхронизированным методам.

Метод *notify()* возобновляет работу потока, который был приостановлен ранее методом *wait()* на этом же самом объекте. Если таких потоков несколько, выбирается один из них (какой именно – зависит от реализации JVM).

Метод *notifyAll()* позволяют возобновить работу всех потоков, которые были ранее приостановлены на этом же самом объекте методом *wait()*.

Обычно метод *wait()* вызывается для потока при выполнении некоторого условия и предполагается, что поток должен быть приостановлен до тех пор, пока это условие выполняется, поскольку после пробуждения потока он продолжит выполнять код, следующий за методом *wait()*. Поэтому при использовании *wait()* при некотором условии обычно применяется следующий шаблон программного кода.

```
...
while(<условие>)
    wait();
// Выполнение операторов после разблокирования
...
```

Применение цикла *while* не позволяет потоку пройти дальше до тех пор, пока не нарушится соответствующее условие.

## 4 Базовое приложение для лабораторной работы

**Задание:** составить приложение, в рамках пространства главного окна которого перемещаются фигуры, имеющие некоторую одинаковую геометрическую форму, но отличающиеся по размеру, цвету, скорости и направлению перемещения, которые задаются случайно. При столкновении движущейся фигуры с краем окна происходит ее «отскок» в обратную сторону, т.е. нормальная составляющая вектора скорости изменяет свой знак, а тангенциальная остается без изменений. Добавление новых движущихся фигур происходит при помощи пункта меню «*Фигуры → Добавить фигуру*». Предусмотрена возможность приостановки движения всех фигур с помощью пункта меню «*Управление → Приостановить движение*», а также его возобновления с помощью «*Управление → Возобновить движение*».

Вид главного окна приложения представлен на рисунке 30.

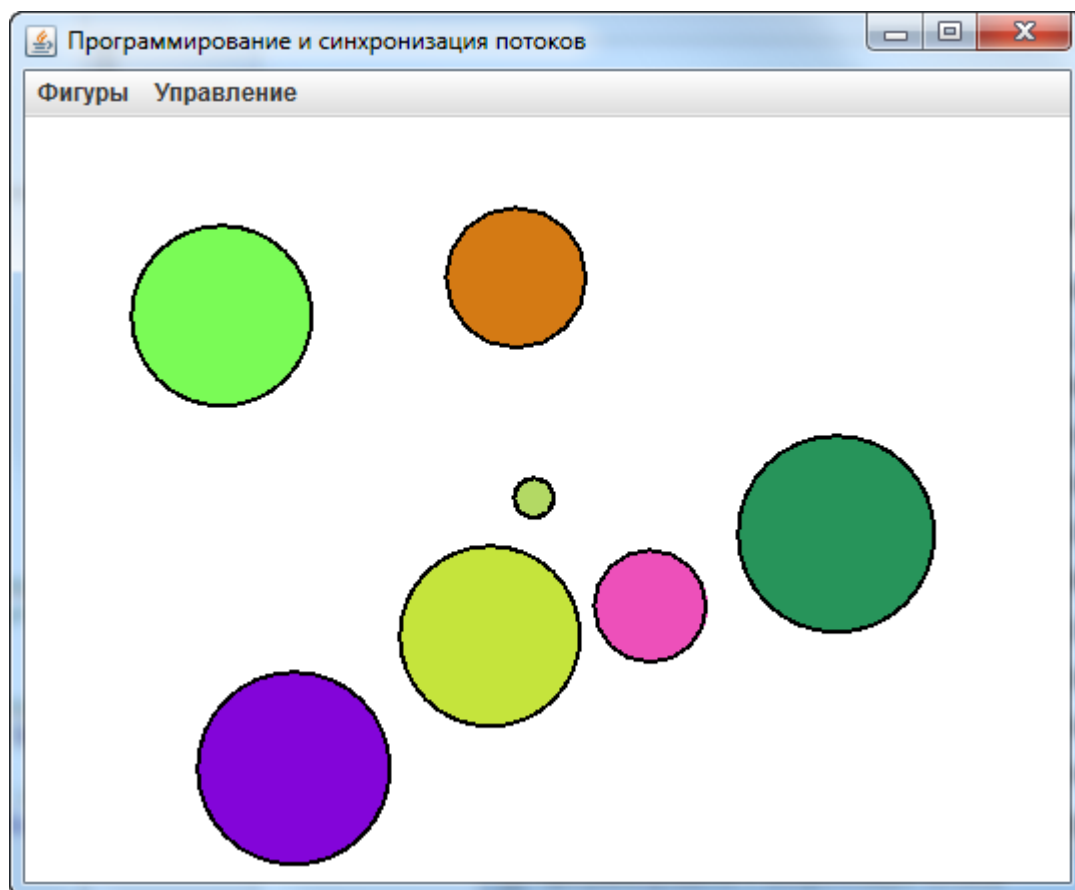


Рис 30. Внешний вид главного окна приложения

### 4.1 Структура приложения

В структуре приложения можно выделить следующие элементы, каждому из которых соответствует отдельный класс:

- **Поле для движения фигур** (класс *Field*), хранящее список всех движущихся фигур, управляющее их движением и ответственное за регулярную перерисовку области окна (с помощью таймера);
- **Движущаяся фигура** (класс *MovingFigure*), ответственная за постоянный пересчет своих координат на основе скорости и направления движения, а также за свое отображение в рамках заданного контекста устройства;
- **Главное окно приложения** (класс *MainFrame*), организующее рабочее пространство окна, показ главного меню и реакцию на действия пользователя.

## 4.2 Реализация класса поля для движения фигур *Field*.

Класс поля для движения фигур *Field* является наследником класса панели *JPanel* и выступает в роли контейнера для перемещающихся в рамках поля фигур. Кроме того, объект класса *Field* используется для синхронизации всех потоков в рамках приложения.

Для хранения списка всех фигур в классе *Field* используется поле данных, представляющее собой динамический список:

```
private ArrayList<MovingFigure> figures = new ArrayList<MovingFigure>(10);
```

Текущее состояние запрета, либо разрешения на движение фигур хранится в булевой переменной-флаге *paused*:

```
private boolean paused;
```

Для обеспечения периодической перерисовки области поля используется объект класса *Timer*, который генерирует события типа *ActionEvent* через заданные промежутки времени. Получателем и обработчиком генерируемых событий является анонимный класс, реализующий интерфейс *ActionListener*, единственной задачей которого является инициирование перерисовки поля для движения фигур:

```
private Timer repaintTimer = new Timer(10, new ActionListener() {
    public void actionPerformed(ActionEvent ev) {
        repaint();
    }
});
```

В конструкторе класса *Field*, задается фоновый цвет поля для движения фигур и запускается таймер:

```

public Field() {
    // Установить цвет заднего фона белым
    setBackground(Color.WHITE);
    // Запустить таймер
    repaintTimer.start();
}

```

Добавление на поле новой фигуры осуществляется в методе *addFigure* простой вставкой в конец списка фигур нового объекта класса *MovingFigure*, так как все настройки параметров фигуры выполняются ей самостоятельно:

```

public void addFigure() {
    figures.add(new MovingFigure(this));
}

```

Код перерисовки поля для движения фигур реализован в методе *paintComponent*, который вызывает одноименный метод суперкласса для прорисовки фона и перебирает объекты фигур, из каждого из которых вызывается метод *paint*, рисующий изображение отдельной фигуры:

```

public void paintComponent(Graphics g) {
    // Вызвать версию метода, унаследованную от предка
    super.paintComponent(g);
    Graphics2D canvas = (Graphics2D) g;
    // Последовательно запросить прорисовку от всех фигур из списка
    for (MovingFigure figure: figures)
        figure.paint(canvas);
}

```

Для обеспечения синхронных действий по приостановке и возобновлению движения всех фигур (т.е. синхронизации связанных с ними потоков) предназначены три метода: *pause*, *resume* и *canMove*. Все они являются синхронизированными (объявлены с модификатором *synchronized*), что ограничивает число потоков, одновременно выполняющихся внутри данных методов, одним.

Метод *pause*, используя эксклюзивный доступ к полю *paused* (оно изменяется только внутри синхронизированных методов), устанавливает ее в значение в *true*, тем самым включая режим приостановки движения фигур:

```

public synchronized void pause() {
    paused = true;
}

```

Метод *canMove* последовательно вызывается каждой из движущихся фигур перед выполнением операции пересчета ее координат. В коде метода проверяется значение переменной *paused*, и если оно равно *true* (т.е. включен режим приостановки движения фигур), то посредством метода *wait* выполняется приостановка потока, связанного с фигурой, вызвавшей метод *canMove*, и право начать выполнение метода *canMove* передается следующей

фигуре. Если значение флага *paused* по-прежнему равно *true*, то и следующий поток, будет приостановлен. Итоговым результатом станет то, что за короткое время после установки флага *paused* в *true* все потоки, связанные с движущимися фигурами, обратившись к методу *canMove* будут приостановлены.

```
public synchronized void canMove(MovingFigure figure)
    throws InterruptedException {
    if (paused)
        wait();
}
```

Для возобновления работы всех ранее приостановленных потоков и соответственно возобновления движения связанных с ними фигур в рамках поля предназначен метод *resume*, который изменяет значение поля *paused* на *false*, после чего с помощью вызова метода *notifyAll* осуществляет возобновление работы всех приостановленных потоков:

```
public synchronized void resume() {
    // Выключить режим паузы
    paused = false;
    // Будим все ожидающие продолжения потоки
    notifyAll();
}
```

#### 4.3 Реализация класса движущейся фигуры *MovingFigure*.

Задачей класса *MovingFigure* является инициализация внутренних атрибутов объекта движущейся фигуры в момент его создания (в теле конструктора), постоянный пересчет координат движущейся фигуры, а также выполнение операций по ее отображению на экране с учетом текущих значений атрибутов (координаты, размер, цвет, вид границы).

Код класса *MovingFigure* начинается с задания трех констант, которые в дальнейшем будут использованы для определения размеров и скорости движущейся фигуры:

```
// Максимальный размер половины стороны квадрата, описывающего фигуру
private static final int maxFramingSquareHalfSize = 60;
// Минимальный размер половины стороны квадрата, описывающего фигуру
private static final int minFramingSquareHalfSize = 10;
// Минимальное время, на которое может засыпать поток после каждого смещения
private static final int minSleepTime = 1;
```

Константы *maxFramingSquareHalfSize* и *minFramingSquareHalfSize* будут использованы для задания размеров движущейся фигуры, а константа *minSleepTime* будет задействована при определении характеристики, влияющей на скорость ее движения.



Поле *field* содержит ссылку на объект поля для движения фигур и используется для приостановки и возобновления движения фигуры (вызывается метод *canMove*), а также при пересчете ее координат для получения информации о размерах области для движения фигур:

```
private Field field;
```

Поскольку в рамках данного приложения движущаяся фигура независимо от ее формы является вписанной в квадрат, то для определения размеров движущейся фигуры задается половина размера стороны описывающего ее квадрата, значение которой хранится в поле *framingSquareHalfSize*:

```
private int framingSquareHalfSize;
```

Для задания цвета заливки внутреннего пространства в границах движущейся фигуры используется поле *color*:

```
private Color color;
```

Поле *stroke* содержит ссылку на объект типа *BasicStroke*, настройки которого определяют штриховку, используемую при рисовании линий и границ для движущейся фигуры:

```
private Stroke stroke = new BasicStroke(2.0f, BasicStroke.CAP_BUTT,
                                         BasicStroke.JOIN_ROUND, 10.0f, null, 0.0f);
```

Поля *x* и *y* содержат значения, определяющие текущие координаты центра движущейся фигуры, который соответствует пересечению диагоналей описывающего ее квадрата:

```
private double x;
private double y;
```

Поле *sleepTime* содержит время сна потока, связанного с движущейся фигурой, между двумя пересчетами ее координат:

```
private int sleepTime;
```

Данное поле фактически определяет скорость движения фигуры. Чем больше время сна потока, в котором пересчитываются координаты движущейся фигуры, тем реже выполняется данный пересчет, и, соответственно, тем медленнее данная фигура перемещается по полю для движения фигур.

Поля *shiftX* и *shiftY* содержат расстояние вдоль осей X и Y соответственно, на которое смещается движущаяся фигура при каждом пересчете ее координат:

```
private double shiftX;
private double shiftY;
```

Стоит отметить, что величина данных смещений определяет лишь направление движения фигуры, а скорость ее перемещения зависит только от значения поля *sleepTime*.

Значения полей, за исключением поля *stroke*, генерируются в конструкторе класса *MovingFigure* случайно в соответствии с равномерным распределением, заданным на некотором интервале:

```
public MovingFigure(Field field) {
    // Необходимо иметь ссылку на поле, по которому передвигается фигура,
    // чтобы отслеживать выход за его пределы через getWidth(), getHeight()
    this.field = field;

    // Случайно выбираем размер половины стороны квадрата, описывающего фигуру
    framingSquareHalfSize = minFramingSquareHalfSize + new Double(Math.random() *
        (maxFramingSquareHalfSize - minFramingSquareHalfSize)).intValue();

    // Абсолютное значение времени сна потока зависит от размера фигуры,
    // чем он больше, тем больше
    sleepTime = 16 -
        new Double(Math.round(210 / framingSquareHalfSize)).intValue();
    if (sleepTime < minSleepTime)
        sleepTime = minSleepTime;

    // Начальное направление движения случайно, угол в пределах от 0 до 2PI
    double angle = Math.random() * 2 * Math.PI;

    // Вычисляются горизонтальная и вертикальная компоненты смещения
    shiftX = 3 * Math.cos(angle);
    shiftY = 3 * Math.sin(angle);

    // Цвет фигуры выбирается случайно
    color = new Color((float) Math.random(), (float) Math.random(),
        (float) Math.random());

    // Начальное положение фигуры случайно
    x = framingSquareHalfSize + Math.random() *
        (field.getSize().getWidth() - 2 * framingSquareHalfSize);
    y = framingSquareHalfSize + Math.random() *
        (field.getSize().getHeight() - 2 * framingSquareHalfSize);

    // Создаем новый экземпляр потока, передавая аргументом
    // ссылку на объект класса, реализующего Runnable (т.е. на себя)
    Thread thisThread = new Thread(this);
    // Запускаем поток
    thisThread.start();
}
```

Кроме инициализации полей, в конце кода конструктора класса *MovingFigure* также создается и запускается на выполнение отдельный поток, предназначенный для пересчета координат движущейся фигуры.

Метод *run*, выполняемый для каждой движущейся фигуры в отдельном потоке, содержит основную логику по периодическому пересчету ее положения в рамках поля для движущихся фигур:

```
public void run() {
    try {
        // Крутим бесконечный цикл, т.е. пока нас не прервут,
        // мы не намерены завершаться
        while(true) {
            // Синхронизация потоков выполняется на объекте field.
            // Если движение разрешено – управление будет возвращено в метод.
            // В противном случае – активный поток заснет
            field.canMove(this);

            if (x + shiftX <= framingSquareHalfSize) {
                // Достигли левой стенки, отскакиваем право
                shiftX = -shiftX;
                x = framingSquareHalfSize;
            } else if (x + shiftX >= field.getWidth() - framingSquareHalfSize) {
                // Достигли правой стенки, отскок влево
                shiftX = -shiftX;
                x=new Double(field.getWidth()-framingSquareHalfSize).intValue();
            } else if (y + shiftY <= framingSquareHalfSize) {
                // Достигли верхней стенки
                shiftY = -shiftY;
                y = framingSquareHalfSize;
            } else if (y + shiftY >= field.getHeight() - framingSquareHalfSize) {
                // Достигли нижней стенки
                shiftY = -shiftY;
                y=new Double(field.getHeight() - framingSquareHalfSize).intValue();
            } else {
                // Просто смещаемся
                x += shiftX;
                y += shiftY;
            }

            // Засыпаем на sleepTime миллисекунд
            Thread.sleep(sleepTime);
        }
    } catch (InterruptedException ex) {
        // Если нас прервали, то ничего не делаем
        // и просто выходим (завершаемся)
    }
}
```

Кроме того, перед каждым пересчетом координат в методе *run* вызывается метод *canMove* объекта *field* который обеспечивает участие движущейся фигуры в процессе приостановки/возобновления движения. После блока кода, пересчитывающего координаты фигуры вызывается метод *sleep* класса *Thread*, приостанавливающий работу потока на *sleepTime* миллисекунд, что позволяет регулировать скорость движения фигуры.

Метод *paint*, рисует изображение движущейся фигуры в границах поля для движения фигур, используя связанный с данным компонентом объект класса *Graphics2D*:

```
public void paint(Graphics2D canvas) {
    //Создаем объект фигуры
    Ellipse2D.Double figure = new Ellipse2D.Double(
        x-framingSquareHalfSize, y-framingSquareHalfSize,
        2*framingSquareHalfSize, 2*framingSquareHalfSize);

    //Задаем цвет и выполняем заливку фигуры
    canvas.setPaint(color);
    canvas.fill(figure);

    //Задаем цвет и стиль линии границы и рисуем границу фигуры
    canvas.setStroke(stroke);
    canvas.setPaint(Color.black);
    canvas.draw(figure);
}
```

Для базового приложения данный метод создает объект для окружности, вписанной в квадрат со стороной равной  $2*framingSquareHalfSize$ , центр которой задан координатами  $x, y$ . Далее внутреннее пространство окружности заливается цветом *color* путем вызова метода *fill*. После этого прорисовывается граница окружности черным цветом и с использованием штриховки, заданной объектом *stroke*.

#### 4.4 Реализация главного окна приложения

Задачами главного класса приложения *MainFrame* является создание и компоновка элементов пользовательского интерфейса (поля для движения фигур и полосы меню, включающей меню «*Фигуры*» и меню «*Управление*»), а также конструирование и показ главного окна приложения. Эти задачи решаются способом, аналогичным рассмотренному в лабораторной работе 2, и поэтому подробно здесь описываться не будут. С полным исходным кодом класса можно ознакомиться в Приложении 1.











.



## 5 Задания

### 5.1 Вариант сложности А

- а) Модифицировать в соответствии с вариантом задания (таблица 11) вид движущихся фигур и стиль линии для рисования их контура.
- б) Добавить в соответствии с вариантом задания (таблица 11) возможность выборочной остановки движущихся фигур, для чего добавить в меню «Управление» еще один пункт меню, доступный для использования только при наличии хотя бы одной фигуры и при условии, что не был включен режим остановки всех фигур или режим выборочной остановки.

Таблица 11 Варианты заданий для уровня сложности А

№ п/п	Вид фигуры*	Стиль линии контура фигуры	Условие выборочной остановки фигур
1		— — — — —	размер стороны описывающего фигуру квадрата менее 40 точек
2		— — — — —	фигура движется с малой скоростью (время сна потока более 8 мс)
3		- - - - -	угол скорости фигуры находится между 90 и 180 градусов (2-я четверть).
4		- - - - -	интенсивность зеленой компоненты в цвете заливки фигуры превышает сумму двух других компонент
5		— — — — —	размер стороны описывающего фигуру квадрата более 80 точек
6		- - - - -	фигура движется с большой скоростью (время сна потока менее 8 мс)
7		— — — — —	угол скорости фигуры находится между 0 и 90 градусов (1-я четверть)
8		— — — — —	останавливается 5 фигур, остальные продолжают движение
9		— — — — —	угол скорости фигуры находится между 180 и 270 градусов (3-я четверть)
10		- - - - -	интенсивность красной компоненты в цвете заливки фигуры превышает сумму двух других компонент

№ п/п	Вид фигуры*	Стиль линии контура фигуры	Условие выборочной остановки фигур
11		— — — — —	останавливается все фигуры, за исключением трех, которые продолжают движение
12		— — — — —	угол скорости фигуры находится между 270 и 360 градусов (4-я четверть)

\* пунктирной линией серого цвета изображен квадрат описывающий фигуру, который не является ее частью и представлен в целях пояснения позиционирования элементов фигуры в его пределах.

## 5.2 Вариант сложности В

- Выполнить задание а) соответствующего варианта уровня сложности А.
- Выполнить задание б) соответствующего варианта уровня сложности А.
- Добавить в меню «Управление» приложения пункт меню с флажком, обеспечивающий возможность включения/выключения дополнительной опции, которая описана в таблице 12. Название пункта меню должно совпадать с названием дополнительной опции.

Таблица 12 (для уровня сложности В)

№ п/п	Описание дополнительной опции
1	Опция «Трение» в активном состоянии постоянно снижает скорость движения фигур. Для регулировки степени снижения скорости движения фигур использовать специальный коэффициент, для задания значения которого добавить в меню «Управление» пункт меню «Задать коэффициент трения», отображающий соответствующее диалоговое окно. При снижении скорости движения фигуры до определенного порогового уровня (значение выбрать самостоятельно) прекратить пересчет ее координат путем завершения работы соответствующего ей дополнительного потока выполнения.
2	Опция «Регулировка скорости», в активном состоянии позволяет изменять скорость движения всех фигур путем нажатия кнопок «+» и «-». При нажатии кнопки «+» скорость возрастает, при нажатии «-» – уменьшается.
3	Опция «Магнетизм», в активном состоянии обеспечивает остановку («прилипание») фигур к границам области их перемещения в момент касания соответствующей границы. Когда эффект выключается (при условии, что движение фигур разрешено другими опциями приложения), все «прилипшие» фигуры разом отскакивают от стенок и продолжают движение с теми направлениями и скоростями, которые были до «прилипания». Обеспечить корректную работу опции совместно с остановками и возобновлениями движения фигур, реализованными при выполнении задания б).
4	Опция «Наждачная бумага» в активном состоянии при каждом ударе фигуры о границу области ее перемещения уменьшает размер стороны описывающего ее квадрата на X пикселей. Для задания X добавить в меню «Управление» пункт меню «Задать степень истирания», отображающий соответствующее диалоговое окно. Если размер стороны описывающего фигуру квадрата достиг нуля, то выполнение связанного с ней потока прекращается, и объект фигуры уничтожается.



№ п/п	Описание дополнительной опции
5	Опция «Снежный ком» в активном состоянии через каждые X пикселей пройденного фигурой пути увеличивает размер описывающего ее квадрата на Y. Для задания величин X и Y использовать отдельное диалоговое окно, отображаемое при нажатии на дополнительный пункт меню «Задать параметры роста». При каждом увеличении размера фигуры, ее скорость должна пересчитываться в сторону уменьшения (для пересчета скорости использовать тот же алгоритм, который применяется для задания начальной скорости движения фигуры в базовом приложении). Если скорость фигуры уменьшилась до нуля, прекратить пересчет ее координат путем завершения работы соответствующего ей дополнительного потока выполнения.
6	Опция «Только команда» в активном состоянии останавливает все фигуры, имеющие специальный идентификатор, равный заданному пользователем. Для обеспечения работы опции случайно генерировать и ставить в соответствии каждой фигуре идентификатор в виде строки (например, “А”, “В”, “С”, или “D”), который должен отображаться рядом с фигурой. Для задания пользователем значения идентификатора группы фигур, движение которых при включении опции сохраняется, добавить в меню «Управление» пункт меню «Задать идентификатор команды», отображающий соответствующее диалоговое окно. Движение всех фигур, идентификатор которых не совпадает с введенным пользователем, после включения опции приостанавливается, а после ее выключения возобновляется в том же направлении и с той же скоростью. Обеспечить корректную работу опции совместно со остановками и возобновлениями движения фигур, реализованными при выполнении задания б).
7	Опция «Точка сбора» в активном состоянии заставляет все фигуры оставить свои траектории движения и двигаться с сохранением своей скорости к точке, над которой сейчас находится курсор, до тех пор, пока центр фигуры не совместится с данной точкой. При изменении положения курсора фигуры смещаются соответствующим образом со своими скоростями. При выключении опции фигуры разлетаются в случайные стороны.
8	Опция «Ловушка» в активном состоянии добавляет в область движения фигур ловушку (прямоугольник с красной сплошной границей и размером стороны 200 пикселей), которая двигается в рамках области движения аналогично обычным фигурам. В случае, если во время своего движения обычная фигура полностью влетает внутрь ловушки, то выйти за ее пределы она не может и движется внутри ее, каждый раз отскакивая от ее внутренних стенок. При выключении опции ловушка исчезает, а фигуры, находившиеся в ней, получают способность двигаться в рамках всей области движения.
9	Опция «Теория относительности» в активном состоянии делает размер главного окна приложения равным 700x500 пикселей, помещает его левый верхний угол в случайную позицию на экране, однако так, чтобы все окно оставалось в пределах экрана, и вызывает движение окна главного приложения в рамках экрана аналогично движению фигур внутри окна. При этом фигуры внутри окна продолжают перемещаться. Для упрощения работы с приложением обеспечить отключение опции «Теория относительности» при нажатии клавиши «Esc» на клавиатуре, что должно наряду с остановкой главного окна приложения и переводом его в максимизированное состояние также снимать флажок у соответствующего пункта меню.




№ п/п	Описание дополнительной опции
10	Опция «Сортировка по цвету» в активном состоянии отображает в рамках области движения фигур три непересекающихся прямоугольника со сплошной границей красного, зеленого и синего цвета соответственно. Прямоугольники должны покрывать все пространство области перемещения фигур. При попадании фигуры в цвет которой интенсивность одной компоненты (например, красной) больше, чем интенсивность других компонент (например, зеленой и синей) в прямоугольник с соответствующим цветом границы (например, красным) данная фигура остается двигаться только в рамках данного прямоугольника. Со временем, таким образом, достигается разделение (сортировка) фигур по соответствующим прямоугольникам. При выключении опции прямоугольники исчезают, а фигуры, находившиеся в них, получают способность двигаться в рамках всей области движения. Для обеспечения большей наглядности модифицировать алгоритм генерации цветов при создании фигур таким образом, чтобы генерировались только фигуры красного, зеленого, либо синего цвета.
11	Опция «Бермудский треугольник» в активном состоянии отображает в центре области движения фигур равносторонний треугольник с границей красного цвета и стороной 300 пикселей по умолчанию. При изменении размеров области движения фигур треугольник должен оставаться в центре области и быть равносторонним. Если при уменьшении области движения фигур места для треугольника не хватает, необходимо уменьшить его размер, так, чтобы он влезал в окно. При увеличении окна размер треугольника должен увеличиваться вплоть до достижения его сторонами размера 300 пикселей. Если опция включена, при попадании корпуса корабля (не включая мачту и флаг) полностью внутрь треугольника, корабль должен исчезать, при этом связанный с ним объект удаляется из памяти.
12	Опция «Граница дня и ночи» в активном состоянии разделяет область движения фигур две равные части белого (слева) и черного (справа) цвета. При включенной опции цвет заливки круга фигуры из задания сложности А) должен быть желтым, если круг полностью находится в черной части области движения фигур и белым в противном случае. Цвет заливки треугольника в нижней части фигуры остается таким, каким он был определен при создании фигуры. При выключении опции «Граница дня и ночи» цвет заливки круга фигуры становится таким, как и цвет треугольного основания.

### 5.3 Вариант сложности С

- а) Выполнить задание а) соответствующего варианта уровня сложности А.
- б) Выполнить задание б) соответствующего варианта уровня сложности А.
- в) Выполнить задание в) соответствующего варианта уровня сложности В.
- г) Реализовать дополнительные возможности приложения в соответствии с вариантом задания (таблица 13).

Таблица 13 (для уровня сложности С)

№ п/п	Описание дополнительных возможностей приложения
1	<p>Добавить в меню «Управление» пункт меню с флажком «Вращение» и обычный пункт меню «Направление вращения». При установке пользователем флажка для пункта меню «Вращение» реализовать вращение всех движущихся фигур вокруг центров, соответствующих им описывающих квадратов, которое должно происходить наряду с их поступательным движением. Все фигуры поворачиваются на 5° за один пересчет координат и вращаются в одну сторону (по часовой стрелке, либо против нее). Обеспечить при нажатии пункта меню «Направление вращения» отображение диалогового окна с двумя радиокнопками: «По часовой стрелке» и «Против часовой стрелки», позволяющими пользователю выбрать направление вращения фигур.</p>
2	<p>Добавить в меню «Управление» пункт меню с флажком «Телепорт». При установке пользователем флажка для данного пункта меню отображать в области движения фигур вход и выход телепорта в виде двух кругов диаметром 150 пикселей. Внутреннее пространство входа телепорта залить радиальным градиентом красного цвета, полностью прозрачным в центре и непрозрачным на периферии круга. Для выхода телепорта использовать аналогичное заполнение зеленого цвета. Обеспечить автоматическое изменение местоположения входа и выхода телепорта с целью их нахождения внутри области движения фигур при изменении размеров окна. Если пользователь отмечает флажок пункта меню «Телепорт», а размер окна недостаточен для показа входа и выхода телепорта, то увеличивать размеры окна автоматически. Обеспечить перемещение входа и выхода телепорта при помощи мыши в рамках области движения фигур. При попадании движущейся фигуры полностью внутрь области входа телепорта автоматически перемещать ее в область выхода телепорта с сохранением расстояния до центра области телепорта, скорости и направления движения фигуры.</p>
3	<p>Добавить в меню «Управление» пункт меню с флажком «Лупа». При установке пользователем флажка для данного пункта меню отображать в области движения фигур лупу в виде круга диаметром 300 пикселей. Для рисования границы лупы использовать сплошную линию толщиной 2 пикселя. Внутреннее пространство лупы заполнять полупрозрачным содержимым с использованием цветовой схемы на базе текстуры вида:</p>  <p>Обеспечить автоматическое изменение местоположения лупы с целью ее нахождения внутри области движения фигур при изменении размеров окна. Если пользователь отмечает флажок пункта меню «Лупа», а размер окна недостаточен для ее показа, то увеличивать размеры окна автоматически. Обеспечить перемещение изображения лупы при помощи мыши в рамках области движения фигур. При попадании движущейся фигуры частично, либо полностью внутрь области лупы отображать данную часть в двукратно увеличенном размере.</p>

## Приложение 1. Исходный код базового приложения

### Класс поля для движения фигур *Field*

```
package mainPackage;

import java.awt.Color;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.ArrayList;
import javax.swing.JPanel;
import javax.swing.Timer;

@SuppressWarnings("serial")
public class Field extends JPanel {
    // Флаг приостановки движения
    private boolean paused;
    // Динамический список движущихся фигур
    private ArrayList<MovingFigure> figures = new ArrayList<MovingFigure>(10);
    // Класс таймер отвечает за регулярную генерацию событий ActionEvent
    // При создании его экземпляра используется анонимный класс,
    // реализующий интерфейс ActionListener
    private Timer repaintTimer = new Timer(10, new ActionListener() {
        public void actionPerformed(ActionEvent ev) {
            // Задача обработчика события ActionEvent - перерисовка окна
            repaint();
        }
    });

    // Конструктор класса Field
    public Field() {
        // Установить цвет заднего фона белым
        setBackground(Color.WHITE);
        // Запустить таймер
        repaintTimer.start();
    }

    // Унаследованный от JPanel метод перерисовки компонента
    public void paintComponent(Graphics g) {
        // Вызвать версию метода, унаследованную от предка
        super.paintComponent(g);
        Graphics2D canvas = (Graphics2D) g;
        // Последовательно запросить прорисовку от всех фигур из списка
        for (MovingFigure figure: figures)
            figure.paint(canvas);
    }

    // Метод добавления новой фигуры в список
    public void addFigure() {
        // Заключается в добавлении в список нового экземпляра MovingFigure
        // Всю инициализацию положения, скорости, размера, цвета
        // MovingFigure выполняет в своем конструкторе
        figures.add(new MovingFigure(this));
    }

    // Синхронизированный метод приостановки движения фигур
    // (только один поток может одновременно быть внутри)
    public synchronized void pause() {
```

```

        // Включить режим паузы
        paused = true;
    }

    // Синхронизированный метод возобновления движения фигур
    // (только один поток может одновременно быть внутри)
    public synchronized void resume() {
        // Выключить режим паузы
        paused = false;
        // Будим все ожидающие продолжения потоки
        notifyAll();
    }

    // Синхронизированный метод проверки, может ли фигура двигаться
    // (не включен ли режим паузы?)
    public synchronized void canMove(MovingFigure figure) throws
    InterruptedException {
        // Если режим паузы включен, то поток, зашедший
        // внутрь данного метода, засыпает
        if (paused)
            wait();
    }
}

```

## Класс движущейся фигуры *MovingFigure*

```

package mainPackage;

import java.awt.BasicStroke;
import java.awt.Color;
import java.awt.Graphics2D;
import java.awt.Stroke;
import java.awt.geom.Ellipse2D;

public class MovingFigure implements Runnable {
    // Максимальный размер половины стороны квадрата, описывающего фигуру
    private static final int maxFramingSquareHalfSize = 60;
    // Минимальный размер половины стороны квадрата, описывающего фигуру
    private static final int minFramingSquareHalfSize = 10;
    // Минимальное время, на которое может засыпать поток после каждого смещения
    private static final int minSleepTime = 1;

    private Field field;

    // Характеристики фигуры
    // Половина стороны квадрата, описывающего фигуру
    private int framingSquareHalfSize;
    // Цвет для заливки внутреннего пространства фигуры
    private Color color;
    // Штриховка для рисования границы фигуры
    private Stroke stroke = new BasicStroke(2.0f, BasicStroke.CAP_BUTT,
                                             BasicStroke.JOIN_ROUND, 10.0f, null, 0.0f);

    // Текущие координаты центра фигуры
    private double x;
    private double y;

    // Время сна потока между двумя пересчетами координат
    private int sleepTime;
    // Вертикальная и горизонтальная компоненты смещения
    private double shiftX;

```

```

private double shiftY;

// Конструктор класса MovingFigure
public MovingFigure(Field field) {
    // Необходимо иметь ссылку на поле, по которому передвигается фигура,
    // чтобы отслеживать выход за его пределы через getWidth(), getHeight()
    this.field = field;

    //Случайно выбираем размер половины стороны квадрата, описывающего фигуру
    framingSquareHalfSize = minFramingSquareHalfSize + new Double(
        Math.random()*(maxFramingSquareHalfSize -
            minFramingSquareHalfSize)).intValue();

    // Абсолютное значение времени сна потока зависит от размера фигуры,
    // чем он больше, тем больше
    sleepTime = 16 -
        new Double(Math.round(210 / framingSquareHalfSize)).intValue();
    if (sleepTime < minSleepTime)
        sleepTime = minSleepTime;

    // Начальное направление движения случайно, угол в пределах от 0 до 2PI
    double angle = Math.random()*2*Math.PI;

    // Вычисляются горизонтальная и вертикальная компоненты смещения
    shiftX = 3*Math.cos(angle);
    shiftY = 3*Math.sin(angle);

    // Цвет фигуры выбирается случайно
    color = new Color((float)Math.random(), (float)Math.random(),
        (float)Math.random());

    // Начальное положение фигуры случайно
    x = framingSquareHalfSize + Math.random()*
        (field.getSize().getWidth()-2*framingSquareHalfSize);
    y = framingSquareHalfSize + Math.random()*
        (field.getSize().getHeight()-2*framingSquareHalfSize);

    // Создаем новый экземпляр потока, передавая аргументом
    // ссылку на объект класса, реализующего Runnable (т.е. на себя)
    Thread thisThread = new Thread(this);
    // Запускаем поток
    thisThread.start();
}

// Метод run() выполняется внутри потока. Когда он завершает работу,
// то завершается и поток
public void run() {
    try {
        // Крутим бесконечный цикл, т.е. пока нас не прервут,
        // мы не намерены завершаться
        while(true) {
            // Синхронизация потоков выполняется на объекте field.
            // Если движение разрешено - управление будет возвращено в метод.
            // В противном случае - активный поток заснет
            field.canMove(this);

            if (x + shiftX <= framingSquareHalfSize) {
                // Достигли левой стенки, отскакиваем право
                shiftX = -shiftX;
                x = framingSquareHalfSize;
            }
        }
    }
}

```

```

    } else if (x + shiftX >= field.getWidth()-
                                                framingSquareHalfSize) {
        // Достигли правой стенки, отскок влево
        shiftX = -shiftX;
        x=new Double(field.getWidth()-
                                                framingSquareHalfSize).intValue();
    } else if (y + shiftY <= framingSquareHalfSize) {
        // Достигли верхней стенки
        shiftY = -shiftY;
        y = framingSquareHalfSize;
    } else if (y + shiftY >= field.getHeight()-
                                                framingSquareHalfSize) {
        // Достигли нижней стенки
        shiftY = -shiftY;
        y=new Double(field.getHeight()-
                                                framingSquareHalfSize).intValue();
    } else {
        // Просто смещаемся
        x += shiftX;
        y += shiftY;
    }

    // Засыпаем на sleepTime миллисекунд
    Thread.sleep(sleepTime);
}
} catch (InterruptedException ex) {
    // Если нас прервали, то ничего не делаем
    // и просто выходим (завершаемся)
}

}

// Метод прорисовки самой себя
public void paint(Graphics2D canvas) {
    //Создаем объект фигуры
    Ellipse2D.Double figure = new Ellipse2D.Double(
        x-framingSquareHalfSize, y-framingSquareHalfSize,
        2*framingSquareHalfSize, 2*framingSquareHalfSize);

    //Задаем цвет и выполняем заливку фигуры
    canvas.setPaint(color);
    canvas.fill(figure);

    //Задаем цвет и стиль линии границы и рисуем границу фигуры
    canvas.setStroke(stroke);
    canvas.setPaint(Color.black);
    canvas.draw(figure);
}
}

```

## **Главный класс приложения *MainFrame***

```

package mainPackage;

import java.awt.BorderLayout;
import java.awt.Toolkit;
import java.awt.event.ActionEvent;
import javax.swing.AbstractAction;
import javax.swing.Action;
import javax.swing.JFrame;
import javax.swing.JMenu;
import javax.swing.JMenuBar;

```

```

import javax.swing.JMenuItem;

@SuppressWarnings("serial")
public class MainFrame extends JFrame {
    // Константы, задающие размер окна приложения, если оно
    // не распахнуто на весь экран
    private static final int WIDTH = 700;
    private static final int HEIGHT = 500;
    private JMenuItem pauseMenuItem;
    private JMenuItem resumeMenuItem;
    // Поле, по которому движутся фигуры
    private Field field = new Field();

    // Конструктор главного окна приложения
    public MainFrame() {
        super("Программирование и синхронизация потоков");
        setSize(WIDTH, HEIGHT);
        Toolkit kit = Toolkit.getDefaultToolkit();
        // Отцентрировать окно приложения на экране
        setLocation((kit.getScreenSize().width - WIDTH)/2,
                    (kit.getScreenSize().height - HEIGHT)/2);
        // Установить начальное состояние окна развернутым на весь экран
        setExtendedState(MAXIMIZED_BOTH);

        // Создать меню
        JMenuBar menuBar = new JMenuBar();
        setJMenuBar(menuBar);

        JMenu figuresMenu = new JMenu("Фигуры");
        Action addFigureAction = new AbstractAction("Добавить фигуру") {
            public void actionPerformed(ActionEvent event) {
                field.addFigure();
                if (!pauseMenuItem.isEnabled() &&
                    !resumeMenuItem.isEnabled()) {
                    // Ни один из пунктов меню не является
                    // доступным - сделать доступным "Паузу"
                    pauseMenuItem.setEnabled(true);
                }
            }
        };
        menuBar.add(figuresMenu);
        figuresMenu.add(addFigureAction);

        JMenu controlMenu = new JMenu("Управление");
        menuBar.add(controlMenu);
        Action pauseAction = new AbstractAction("Приостановить движение"){
            public void actionPerformed(ActionEvent event) {
                field.pause();
                pauseMenuItem.setEnabled(false);
                resumeMenuItem.setEnabled(true);
            }
        };
        pauseMenuItem = controlMenu.add(pauseAction);
        pauseMenuItem.setEnabled(false);

        Action resumeAction = new AbstractAction("Возобновить движение") {
            public void actionPerformed(ActionEvent event) {
                field.resume();
                pauseMenuItem.setEnabled(true);
                resumeMenuItem.setEnabled(false);
            }
        };
    }
}

```



```

    };
    resumeMenuItem = controlMenu.add(resumeAction);
    resumeMenuItem.setEnabled(false);

    // Добавить в центр граничной компоновки поле Field
    getContentPane().add(field, BorderLayout.CENTER);
}

// Главный метод приложения
public static void main(String[] args) {
    // Создать и сделать видимым главное окно приложения
    MainFrame frame = new MainFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
}
}

```