МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение высшего образования «Национальный исследовательский Нижегородский государственный университет им. Н. И. Лобачевского»

Е.М. Макаров

Элементы двумерной графики в Java

Учебно-методическое пособие

Рекомендовано методической комиссией
Института информационных технологий, математики и механики
для студентов ННГУ, обучающихся по направлениям подготовки
01.03.01 «Математика»,
02.03.01 «Математика и компьютерные науки»

Нижний Новгород
2017

УДК 004.92(075.8)
ББК 32.97
М15

М15 Макаров Е.М. ЭЛЕМЕНТЫ ДВУМЕРНОЙ ГРАФИКИ В JAVA: Учебно-методическое пособие. – [электронный ресурс] – Нижний Новгород: Нижегородский госуниверситет, 2017. – 56 с.

Фонд электронных образовательных ресурсов ННГУ

Рецензент: д-р техн. наук, профессор кафедры математического обеспечения и суперкомпьютерных технологий ИИТММ **В.Е. Турлапов**

Учебно-методическое пособие «Элементы двумерной графики в Java» описывает элементы объектно-ориентированного программирования, создание графического интерфейса, рисование в терминах пикселов, векторную графику, аффинные отображения и программирование анимации.

Пособие предназначено для поддержки дисциплины «Компьютерная геометрия и геометрическое моделирование», читаемой студентам третьего курса Института информационных технологий, математики и механики ННГУ, обучающихся по направлениям подготовки «Математика» и «Математика и компьютерные науки». Пособие полезно и для более широкого использования: для желающих освоить язык Java; в поддержку освоения и других курсов математики.

УДК 004.92(075.8)
ББК 32.97

Содержание

1. Элементы объектно-ориентированного программирования

1.1. Классы и объекты

Java является объектно-ориентированным языком программирования. Структурной единицей программы на Java является класс. Класс — это набор полей (переменных) и методов (функций), вместе называемых членами класса. Класс также может содержать другие классы, называемые вложенными. Объект класса — это набор значений полей этого класса. Например, класс `Point`, описывающий точку на экране, может содержать поля `x` и `y` типа `int`, а также метод `move(x, y)` для перемещения точки. Имя объекта отделяется от имени поля или метода точкой. Таким образом, если `p` — объект класса `Point`, то `p.x` есть значение поля `x`, а вызов метода `p.move(100, 200)` устанавливает значения полей `x` и `y` объекта `p` в 100 и 200, соответственно.

Класс объявляется следующим образом.

```
class ClassName {
  // описание полей и методов
}
```

Первая буква имени класса обычно заглавная, а объекта, поля или метода — строчная. Перед словом `class` могут быть модификаторы, например, `public` или `abstract`, о которых речь пойдет ниже.

В файле с исходным кодом может быть только один открытый (`public`) класс. Имя файла должно совпадать с именем такого класса. Например, класс `Point`, объявленный `public`, должен находиться в файле `Point.java`.

1.2. Конструктор и ключевое слово `this`

Класс может содержать особые методы, называемые конструкторами, которые вызываются при создании объектов класса. От обычных методов их отличает отсутствие возвращаемого типа, а также то, что имя конструктора совпадает с именем класса. Класс может иметь несколько конструкторов, отличающихся количеством и типами своих параметров. Это же справедливо и для других методов с одинаковым именем. Такой прием называется перегрузкой. Если не объявлено ни одного конструктора, то создается конструктор по умолчанию, не имеющий параметров.

Конструкторы часто используются для инициализации полей класса. Если при этом имя параметра конструктора или другого метода совпадает с именем поля, то такой параметр *скрывает* поле. К полю можно обратиться, указав имя объекта. Существует также специальное имя объекта `this`, значением которого является текущий объект, то есть объект, которому принадлежит вызываемый конструктор или метод. Например, конструктор класса `Point` может выглядеть следующим образом.

```
Point(int x, int y) {
  this.x = x;
  this.y = y;
}
```

Имя `this` используется также для вызова одного конструктора из другого. Если первым оператором конструктора является `this(arguments)`, то будет вызван конструктор,

соответствующий списку аргументов, после чего продолжится выполнение данного конструктора. Например, к приведенному выше конструктору можно добавить конструктор без параметров.

```
Point() {
  this(0, 0);
}
```

1.3. Класс как тип

Класс выступает как тип переменных, содержащих объекты данного класса. Объекты класса создаются следующим образом.

```
ClassName objectName = new ClassName(arguments);
```

Аргументы соответствуют заголовку одного из конструкторов. На самом деле, значением переменной является не сам объект, а ссылка на него. Так, результатом исполнения

```
C x = new C();
C y = x;
```

являются две ссылки `x` и `y`, указывающие на один и тот же объект класса `C`.

1.4. Статические поля и методы

Разные объекты одного класса могут содержать разные значения полей данного класса. Обычные поля называются также переменными экземпляра (то есть объекта класса), а обычные методы — методами экземпляра. Существуют также поля и методы, относящиеся к классу в целом, а не к отдельным объектам. Они называются переменными и методами класса и объявляются с помощью ключевого слова `static`. По этой причине переменные класса называются также статическими полями, а методами класса — статическими методами.

Если `z` — переменная класса `C`, то ее значение есть `C.z`, то есть перед точкой указывается имя класса, а не объекта (последнее тоже допустимо, но не рекомендуется). К переменной или методу класса можно обращаться, даже если не создан ни один объект данного класса. Методы класса не могут использовать переменные экземпляра.

Переменную класса можно использовать, например, для подсчета количества созданных объектов класса. Также переменными класса часто являются константы; в этом случае у них есть модификатор `final`. По соглашению имя константы должно состоять из заглавных букв и подчерков. Примеры констант:

```
public static final double PI = 3.14159265358979323846;
public final static Color WHITE = new Color(255, 255, 255);
```

1.5. Метод `main`

Работа программы на Java начинается с метода

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

Здесь `args` — массив строк, содержащих аргументы, указанные в командной строке при вызове программы. Например, если метод `main()` содержится в классе `MainClass`, то при запуске из командной строки

```
> java MainClass -i file.txt
```

методу `main()` будет передан массив `args`, содержащий строки `"-i"` и `"file.txt"`.

Поскольку метод является статическим, среда исполнения Java может вызвать его перед созданием любых объектов.

1.6. Пакеты

Классы и другие типы объединяются в пакеты, которые могут быть вложены в другие пакеты. Полное имя класса включает имена содержащих его пакетов, разделенные точками, например `java.lang.Math` и `java.awt.Point`. Имена пакетов обычно состоят из строчных букв. Пакеты, являющиеся частью языка Java, начинаются с `java` или `javax`.

Важно отметить, что если один пакет вложен в другой, это не означает, что классы, входящие в первый пакет, входят также и во второй. В этом смысле пакеты и содержащиеся в них классы похожи на каталоги и находящиеся в них файлы. Вложенность пакетов показывает их смысловую структуру.

Пакеты служат для управления пространством имен. Они позволяют создавать разные классы с одинаковыми короткими именами. Чтобы поместить класс в пакет `package_name`, нужно написать

```
package package_name;
```

в первой строчке (не считая комментариев) файла, содержащего данный класс. Если инструкции `package` нет, то класс помещается в пакет по умолчанию.

Структура пакетов совпадает со структурой каталогов, содержащих исходный код классов. Так, класс `graphics.shapes.Point` находится в файле `graphics\shapes\Point.java`.

Есть три способа использовать класс не из текущего пакета: указать его полное имя, импортировать класс или импортировать весь пакет, содержащий класс. Для импорта нужно написать, например,

```
import java.awt.geom.Line2D;
```

после директивы `package`, но до всяких определений. Чтобы импортировать весть пакет `java.awt.geom`, нужно написать

```
import java.awt.geom.*;
```

Поскольку вложенный пакет не является подмножеством того пакета, в который он вложен, инструкция

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

импортирует только классы пакета `java.awt`, но не `java.awt.geom`. Импортировать классы, входящие в текущий пакет, не требуется.

1.7. Доступ к классам и их членам

Класс может быть объявлен как `public` (открытый). В этом случае он будет доступен для использования в любых классах. Без модификатора `public` класс является видимым только в своем пакете.

Члены класса могут быть объявлены как `public`, `private` (закрытые) или `protected` (защищенные); они также могут не иметь модификатора доступа. Смысл описания `public` и отсутствия описания такой же, как и для классов. Если член класса объявлен `private`,

он видим только в своем классе, а члены, объявленные `protected`, доступны в своем пакете и в подклассах, даже если последние не принадлежат тому же пакету. Так, защищенные члены классов стандартной библиотеки могут использоваться в подклассах, определенных пользователем.

Объявление членов класса закрытыми служит для ограничения доступа к деталям реализации класса. Таким образом, при изменении реализации нет необходимости менять остальные части программы. Такое ограничение доступа называется инкапсуляцией.

1.8. Наследование

Можно объявить, что один класс расширяет другой, или является его подклассом. Второй класс при этом называется родительским, базовым или надклассом. Подкласс наследует все члены родительского класса, кроме тех, которые имеют модификатор `private`, а также может объявлять собственные члены. Каждый класс является непосредственным подклассом ровно одного класса. Единственное исключение из этого правила — это класс `Object`, являющийся вершиной иерархии.

Наследование объявляется следующим образом.

```
class SubClassName extends SuperClassName {
  // поля и методы SubClassName
}
```

Ключевой особенностью объектно-ориентированного программирования является то, что объекты подкласса считаются также и объектами родительского класса. Это означает, что следующее присваивание допустимо.

```
SuperClassName x = new SubClassName(arguments);
```

Присваивать объекты подкласса можно не только локальным переменным, но и параметрам методов. Например, метод с заголовком `void m(C x)` может принимать объекты не только класса `C`, но и любого подкласса `C`. Ситуация, когда метод может принимать аргументы разных типов, называется полиморфизмом и является важнейшей особенностью объектно-ориентированного программирования.

Таким образом, глядя на текст программы, в общем случае невозможно определить, (ссылка на) объект какого класса содержится в переменной. Однако, если известно, что в переменной `x` типа `C` содержится объект класса `D`, являющегося подклассом `C`, то тип `x` можно преобразовать следующим образом.

```
D y = (D) x;
```

Эта операция называется приведением типа. Она необходима для вызова метода, объявленного в `D`, но которого нет в `C`. Если `m()` — такой метод, то вызов `y.m()` допустим, в то время как `x.m()` вызывает ошибку компиляции, поскольку на этапе компиляции неизвестно, является ли `x` объектом `D`.

Подкласс может переопределить метод экземпляра надкласса, если он содержит новое определение метода с тем же заголовком, то есть именем, числом и типом параметров, а также возвращаемым типом[1]. В этом случае выбор вызываемого метода определяется во время исполнения в зависимости от настоящего типа объекта, а не типа переменной, содержащей ссылку на данный объект. Пусть, например, `C2` является подклассом `C1` и оба класса определяют метод экземпляра `f()`. В результате исполнения

[1] В более общем случае возвращаемый тип метода подкласса может быть подклассом возвращаемого типа соответствующего метода надкласса

```
C1 x = new C2();
x.f();
```

будет вызван метод класса `C2`, так как объект `x` является экземпляром этого класса, несмотря на то, что ссылка на него содержится в переменной типа `C1`.

Метод подкласса может вызвать метод надкласса, который он переопределяет. Для этого вместо имени объекта нужно указать ключевое слово `super`. Например, метод `f()` класса `C2` может вызвать `super.f()`. Аналогично, конструктор подкласса может вызывать конструктор родительского класса с помощью `super(arguments)`, но это можно делать только в первой строке конструктора подкласса. На самом деле, если конструктор родительского класса не вызывается явным образом, то компилятор вставляет вызов конструктора по умолчанию, то есть без аргументов. Если в родительском классе нет такого конструктора, то это ведет к ошибке компиляции.

Перед переопределяемыми методами рекомендуется ставить аннотацию `@Overrides`. В этом случае компилятор сообщит об ошибке, если заголовок переопределяющего метода не совпадает с заголовком соответствующего метода надкласса.

1.9. Абстрактные классы и интерфейсы

Любой класс можно объявить абстрактным с помощью ключевого слова `abstract`. Это делается в том случае, когда реализация класса определена не полностью. Объекты абстрактного класса создать нельзя, но можно определить подклассы такого класса. Абстрактными также могут быть методы, если у них есть только заголовок, но нет тела, например:

```
abstract void abstractMethod(int argument);
```

Класс, содержащий хотя бы один абстрактный метод, должен быть объявлен абстрактным. Для того, чтобы подкласс абстрактного класса не был в свою очередь абстрактным, нужно написать реализации всех абстрактных методов родительского класса.

Абстрактный класс может иметь один или несколько конструкторов. Поскольку напрямую вызвать эти конструкторы (то есть создать объект класса) нельзя, нет смысла объявлять их открытыми (`public`). Однако конструкторы могут быть вызваны из подкласса с помощью ключевого слова `super`, поэтому конструкторы абстрактного класса нужно объявить защищенными (`protected`).

Интерфейс похож на абстрактный класс, у которого все методы абстрактные[2]. Таким образом, создать объект интерфейса нельзя. Интерфейс может объявлять константы с помощью модификаторов `static` и `final`. Все члены интерфейса по умолчанию имеют доступ `public`.

Про подкласс интерфейса говорят, что он *реализует* этот интерфейс. В отличие от ключевого слова `extends`, используемого при наследования классов, реализация интерфейса объявляется словом `implements`. Еще одно отличие от наследования классов состоит в том, что класс может реализовывать более одного интерфейса. Реализация интерфейса выглядит следующим образом.

```
class ClassName implements InterfaceName1, InterfaceName2, ... { ... }
```

Как и в случае классов, интерфейс выступает в качестве типа, и переменной с таким типом можно присвоить объекты классов, реализующих данный интерфейс.

[2]Интерфейс может также иметь методы класса и некоторые другие члены.

1.10. Вложенные классы

Кроме полей и методов, в классе могут быть объявлены другие классы. Они называются вложенными и также считаются членами внешнего класса. В частности, в отличие от классов верхнего уровня, которые могут быть объявлены только `public` или без модификатора, то есть видимыми в своем пакете, вложенные классы могут быть объявлены как `public`, `private` или `protected`.

Вложенные классы бывают двух типов. Классы, объявленные с ключевым словом `static`, называются статическими, а без него — внутренними. Более того, внутренние классы могут быть объявлены в теле метода внешнего класса. В этом случае они называются локальными. Также есть специальный синтаксис, который позволяет объявить локальный класс и одновременно создать его объект; при этом имя класса не указывается. Такие классы называются анонимными.

Вложенные классы обычно небольшие и используются только в контексте класса, в котором они объявлены (в таком случае их можно объявить `private`). В частности, они полезны, если класс должен получить доступ к членам другого класса, которые по принципу инкапсуляции должны быть объявлены `private`. Внутренние классы, аналогично методам экземпляра, имеют доступ ко всем членам внешнего класса, в том числе закрытым. Статические вложенные классы, аналогично статическим методам, имеют доступ только к статическим полям внешнего класса, включая закрытые.

Внутренние классы. Объявление внутреннего класса имеет следующий вид.

```
class OuterClass {
  // члены внешнего класса
  class InnerClass {
    // члены внутреннего класса
  }
}
```

Объект внутреннего класса может существовать только как часть объекта содержащего его класса. Если объект внутреннего класса требуется создать в методе экземпляра или конструкторе внешнего класса (то есть тогда, когда объект внешнего класса уже создан), то используется стандартный синтаксис.

```
InnerClass innerObject = new InnerClass();
```

Если объект `outerObject` класса `OuterClass` уже существует, то объект `InnerClass` можно создать и «снаружи».

```
OuterClass.InnerClass innerObject = outerObject.new InnerClass();
```

В методах экземпляра `InnerClass` слово `this` относится к объекту этого класса, а специальная конструкция `OuterClass.this` относится к объемлющему объекту класса `OuterClass`. Ее можно использовать для доступа к полю `OuterClass`, если оно скрывается полем `InnerClass` с тем же именем.

По техническим причинам внутренний класс не может иметь статические члены, кроме констант.

Статические вложенные классы. Объявление класса имеет следующий вид.

```
class OuterClass {
  // члены внешнего класса
```

```
  static class StaticNestedClass {
    // члены статического вложенного класса
  }
}
```

В отличие от объектов внутреннего класса, объекты статического вложенного класса не привязаны к объектам внешнего класса. Статический вложенный класс во многом похож на класс верхнего уровня. Объявлять статический вложенный класс имеет смысл, если он используется только в одном классе или если он логически тесно связан с внешним классом.

В методе или конструкторе `OuterClass` объект `StaticNestedClass` создается стандартным способом.

```
StaticNestedClass nestedObject = new StaticNestedClass();
```

За пределами внешнего класса нужно указать полное имя вложенного класса.

```
OuterClass.StaticNestedClass nestedObject =
  new OuterClass.StaticNestedClass();
```

В качестве примера рассмотрим класс `java.awt.geom.Point2D`, представляющий точку на плоскости. В самом этом классе нет полей, содержащих координаты точки, но есть два статических вложенных класса: `Float` и `Double`. Эти вложенные классы содержат поля `x` и `y` типов `float` и `double`, соответственно. Таким образом, `Point2D` выступает как маркер пространства имен: имена `Float` и `Double` можно использовать только внутри `Point2D`, а вне его к вложенным классам нужно обращаться как `Point2D.Float` и `Point2D.Double`. Аналогично, имеется класс `Line2D`, представляющий отрезок. Внутри него (то есть в пространстве имен `Line2D`) также есть классы `Float` и `Double`, обращаться к которым извне нужно, добавляя имя внешнего класса. Класс `Point2D.Double` можно было бы вынести на верхний уровень с именем, например, `Point2D_Double`, однако разработчики языка сделали его вложенным, чтобы подчеркнуть, что понятие `Point2D.Double` логически относится к категории `Point2D`.

Локальные классы. Классы можно создавать даже внутри блока кода, в частности, в методах. Такие классы называются локальными. Как и внутренние классы, они имеют доступ ко всем членам внешнего класса, но не могут иметь статические члены, кроме констант. Кроме того, локальные классы могут использовать переменные, объявленные в окружающем методе, только если эти переменные не меняют своего значения после инициализации.

Локальные классы широко используются для передачи методов в качестве аргументов. Такая необходимость возникает при написании обработчиков событий. Например, конструктор класса `javax.swing.Timer` принимает два аргумента: интервал между срабатываниями таймера и действие, которое нужно выполнить при каждом срабатывании. До Java версии 8 методы нельзя было присваивать переменным и передавать как аргументы другим методам[3]. Поэтому для передачи в качестве аргументов использовались объекты классов с одним или несколькими методами. Такие классы обычно используются только в одном месте и фактически служат оболочкой для методов. Поэтому их удобно создавать прямо в месте использования, то есть внутри метода.

Чтобы передать объект класса `C` некоторому методу, тип параметра этого метода должен быть или `C`, или надклассом `C`, или интерфейсом, который реализуется классом `C`.

[3]В Java 8 для этого могут использоваться лямбда-выражения.

```
interface Action {
  void actionPerformed();
}

class Timer {
  public Timer(int interval, Action action) {
    // ...
  }
}

class OuterClass {
  public OuterClass() {
    class LocalClass implements Action {
      public void actionPerformed() {
        // Описание действия при срабатывании таймера
      }
    }
    Action action = new LocalClass();
    Timer t = new Timer(100, action);
    // ...
  }
}
```

Рис. 1.1. Пример локального класса

На рис.1.1 представлен интерфейс `Action` с единственным методом. Конструктор класса `Timer` принимает объект любого класса, реализующего `Action`. Конструктор внешнего класса `OuterClass` создает локальный класс `LocalClass`, в котором определяется метод, описывающий действие при срабатывания таймера. Объект этого класса передается конструктору `Timer`.

Анонимные классы. Анонимный класс позволяет описать класс и создать его объект в одно действие. С помощью анонимного класса конструктор `OuterClass()` на рис. 1.1 можно переписать следующим образом.

```
public OuterClass() {
  Timer t = new Timer(100,
                      new Action() {
                        public void actionPerformed() {
                          // Описание действия при срабатывании таймера
                        }
                      });
  // ...
}
```

Обратите внимание, что `new Action() { ... }` является выражением, а не определением или оператором. Это значит, что оно не может появляться в методе само по себе и должно являться частью оператора. В приведенном выше фрагменте оно является одним из аргументов конструктора, который, в свою очередь, является частью оператора

присваивания.

В общем случае предположим, что есть класс верхнего уровня `S` с конструктором `S(C1 x1, ..., Cn xn)`. Тогда выражение

```
new S(e1, ..., en) {
  // описание класса
}
```

приблизительно эквивалентно

```
new A(e1, ..., en)
```

где класс `A` объявлен следующим образом.

```
class A extends S {
  A(C1 x1, ..., Cn xn) {
    super(x1, ..., xn)
  }
  // описание класса
}
```

Если `S` является интерфейсом, то расшифровка происходит похожим образом, но количество аргументов $n = 0$.

1.11. Компилирование и запуск программы из командной строки

Разрабатывать программу на Java удобнее всего в интегрированной среде разработки, такой как NetBeans, IntelliJ IDEA или Eclipse. Такая среда предоставляет множество удобств, таких как подсветка синтаксиса, автоматические отступы, автодополнение, автоматическое добавление инструкций импортирования и показ документации. Среда разработки также вызывает компилятор и показывает ошибки в тексте программы. Более того, NetBeans компилирует программу даже во время ее редактирования, поэтому запуск всего проекта занимает мало времени.

Тем не менее, небольшие программы легко писать в любом текстовом редакторе, а компилировать и запускать непосредственно из командной строки. В Windows командная оболочка реализована в программе `cmd.exe`. Чтобы запускать программу на Java, необходимо установить среду исполнения: Java Runtime Environment, или JRE. Она включает интерпретатор байт-кода `java`, а также реализацию стандартных классов. Однако чтобы разрабатывать программы на Java, нужно установить комплект разработчика: Java Development Kit, или JDK. Он включает в себя JRE, а также содержит компилятор `javac`, переводящий программу в байт-код, и другие инструменты разработчика.

Как говорилось выше, в файле может находиться только один открытый (`public`) класс. При этом расширением файла должно быть `.java`, а имя должно совпадать с именем открытого класса. В файле могут находиться дополнительные классы верхнего уровня с доступом по умолчанию (без модификатора доступа), а также вложенные классы. Тем не менее, рекомендуется помещать каждый класс верхнего уровня в отдельный файл.

Пусть класс `graphics.shapes.Test` с методом `main(String[] args)` находится в файле `C:\Users\Username\java\graphics\shapes\Test.java`. (В Linux каталоги разделяются обычной косой чертой `/`.) Важно отметить, что компилировать и запускать этот файл следует не из каталога, где находится файл, а из родительского каталога внешнего пакета, то есть в данном случае `C:\Users\Username\java`. Файл компилируется следующей командой.

```
C:\Users\Username\java> javac graphics\shapes\Test.java
```

При успешном завершении компиляции генерируется файл `Test.class`, содержащий байт-код. Он запускается с помощью программы `java`. Другой важной особенностью является то, что аргументом `java` является полное имя класса (включая содержащие его пакеты), а не имя файла. Запускать `java` нужно снова из родительского каталога внешнего пакета, поскольку имена пакетов будут интерпретированы как имена подкаталогов.

```
C:\Users\Username\java> java graphics.shapes.Test
```

Если в файле нет инструкции `package` и содержащийся в нем класс находится в пакете по умолчанию, то запускать его нужно из того каталога, в котором находится скомпилированный файл с расширением `.class`.

Среда исполнения Java также ищет классы в каталогах, входящих в путь к классам (classpath). Пусть к классам — это последовательность каталогов, содержащих пакеты скомпилированных классов. Когда среда исполнения ищет класс в одном из этих каталогов, имена пакетов, входящих в полное имя класса, присоединяются к имени каталога. В приведенном выше примере путь к классам должен содержать `C:\Users\Username\java`; тогда поиск класса `graphics.shapes.Test` будет осуществляться в каталоге `C:\Users\Username\java\graphics\shapes`.

Путь к классам можно задать в командной строке с помощью опций `-classpath` или `-cp`. В этом случае компиляцию и запуск программы можно осуществлять из любого каталога. Каталоги, входящие в путь, разделяются точками с запятой в Windows и двоеточиями в Linux. Поиск осуществляется в том порядке, в котором указаны каталоги. Так, можно дать следующую команду.

```
C:\Users\Username\Desktop> java -classpath
   C:\java\classes;C:\Users\Username\java graphics.shapes.Test
```

Путь к классам можно также задать в переменной окружения `CLASSPATH`. Вот примеры ее задания из командной строки в Windows и Linux.

```
set CLASSPATH=C:\java\classes;C:\Users\Username\java
export CLASSPATH=/usr/local/lib/jvm:/home/username/java
```

В Windows вокруг знака равенства не должно быть пробелов.

По умолчанию текущий каталог, обозначаемый точкой, входит в путь к классам, однако если этот путь переопределен, то среда исполнения может не найти классы в текущем каталоге. В этом случае нужно явно включить текущий каталог в путь, например, следующим образом.

```
C:\Users\Username> java -classpath .;C:\Users\Username\java
   graphics.shapes.Test
```

Упражнения

1.1. Статический вложенный класс `java.awt.geom.Rectangle2D.Double` с конструктором

```
Double(double x, double y, double w, double h)
```

представляет прямоугольник с левым верхним углом (x, y), шириной w и высотой h в экранной системе координат, где ось y направлена вниз. Напишите класс `Square` с полями `centerX`, `centerY` и `side` типа `double`, представляющий квадрат и расширяющий `Rectangle2D.Double`. Напишите соответствующий конструктор.

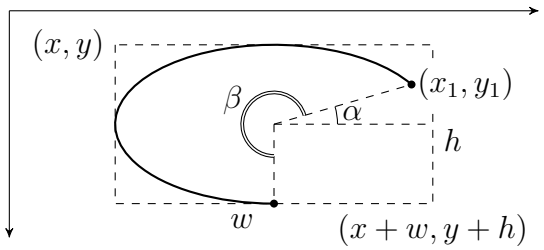


Рис. 1.2. Дуга эллипса, представляемая классом `Arc2D.Double`

1.2. Создайте объект `Rectangle2D.Double`, вызовите его метод `toString()`, возвращающий представление объекта в виде строчки, и напечатайте результат с помощью метода `System.out.println(String s)`. В классе `Square` переопределите `toString()` так, чтобы он возвращал информацию о квадрате, например,

```
return "Square: " + centerX + ", " + centerY + ", " + side;
```

1.3. Статический вложенный класс `java.awt.geom.Ellipse2D.Double` с конструктором

```
Double(double x, double y, double w, double h)
```

представляет эллипс, вписанный в прямоугольник с левым верхним углом (x, y), шириной w и высотой h. Напишите класс `Circle` с полями `centerX`, `centerY` и `radius` типа `double`, представляющий круг и расширяющий `Ellipse2D.Double`. Напишите соответствующий конструктор.

Подобно предыдущему упражнению, переопределите метод `toString()` так, чтобы он возвращал информацию о круге.

1.4. Статический вложенный класс `java.awt.geom.Arc2D.Double` с конструктором

```
Double(double x, double y, double w, double h,
  double alpha, double beta, int type)
```

представляет дугу эллипса, изображенную на рис. 1.2. Последний параметр `type` имеет тип `int` и должен быть равен одной из констант: `Arc2D.OPEN`, `Arc2D.CHORD` или `Arc2D.PIE`. В первом случае фигура представляет собой разомкнутую дугу, во втором концы дуги соединяются отрезком, а в третьем концы соединяются отрезками с центром эллипса.

Начальный угол α и размер дуги β измеряются в градусах. На самом деле, эти параметры являются реальными величинами углов, показанных на рисунке, только если дуга есть часть окружности, то есть $w = h$. В случае, когда дуга есть часть эллипса, α и β соответствуют значениям t в параметрическом задании эллипса $x = (w/2)\cos t$, $y = (h/2)\sin t$ в системе координат с началом в центре эллипса и осью y, направленной вверх. Так, значение $t = 45$ указывает направление из центра в верхний правый угол прямоугольника, описанного вокруг эллипса. Обратите внимание, что углы отсчитываются против часовой стрелки, но ось y экранной системы координат направлена вниз.

Напишите класс `Arc2D_Double`, расширяющий `Arc2D.Double`, с конструктором

```
Arc2D_Double(double x1, double y1, double rx, double ry,
  double alpha, double beta)
```

который отличается от `Arc2D.Double` следующим образом.

1) Вместо ширины и высоты габаритного прямоугольника даны горизонтальная и вертикальная полуоси эллипса $r_x = w/2$ и $r_y = h/2$.

2) Вместо координат угла габаритного прямоугольника даны координаты (x_1, y_1) начальной точки дуги.
3) Параметры α и β имеют тот же смысл и также измеряются в градусах, но углы отсчитываются в направлении, соответствующем оси y. То есть если ось y направлена вверх, то углы отсчитываются против часовой стрелки, как принято в математике.
4) Дуга является разомкнутой, то есть значение параметра `type` есть `Arc2D.OPEN`.

Конструктор `Arc2D_Double` должен состоять из одного вызова `super(...)`. (Некоторые вспомогательные значения для этого вызова можно было бы вычислить заранее и присвоить локальным переменным для ясности, однако вспомните, что `super(...)` должен быть первым оператором в конструкторе.)

1.5. Могут ли классы из одного пакета находиться в разных каталогах? Подсказка: рассмотрите путь к классам, состоящий из нескольких каталогов.

2. Создание графического интерфейса

В платформе Java есть три коллекции пакетов, отвечающих за создание графического пользовательского интерфейса. В порядке появления это AWT (Abstract Window Toolkit), Swing и JavaFX. Каждый из них содержит средства для создания элементов пользовательского интерфейса, таких как окна, кнопки, поля ввода и меню. AWT использует операционную систему, в которой выполняется программа на Java, для рисования этих элементов. Таким образом, программа, использующая AWT и исполняемая под Windows, выглядит как любая другая программа под Windows, но та же программа, исполняемая под Linux, выглядит по-другому. Библиотека Swing содержит собственный код, называемый *стилем оформления*, для рисования элементов интерфейса и не использует для этого операционную систему. Программа на Swing, использующая один и тот же стиль оформления, выглядит одинаково в любой операционной системе. Программа может также использовать стили оформления, имитирующие вид разных операционных систем.

Библиотека Swing создана на основе AWT и широко ее использует, например, для рисования и обработки событий. Мы будем создавать двумерные рисунки с помощью AWT, а пользовательский интерфейс с помощью Swing. Описание JavaFX выходит за рамки данной работы.

Каждому элементу интерфейса, такому как окно, метка (участок экрана с надписью или изображением) или кнопка, соответствует класс Swing. Многие классы из Swing начинаются на букву J. Дерево наследования некоторых классов показано на рис. 2.1.

Элементы интерфейса Swing делятся на так называемые *контейнеры верхнего уровня*, такие как `JFrame` и `JDialog`, и подклассы `JComponent`, называемые *компонентами*. `JFrame` и `JDialog` представляют собой отдельные окна (главное окно программы и окно диалога, соответственно), которые могут содержать в себе различные компоненты, такие как метка `JLabel` или кнопка `JButton`. Компоненты, наряду с контейнерами верхнего уровня, являются подклассами `Container` и могут содержать другие компоненты. Примером компонента, используемой в первую очередь как контейнер, является панель `JPanel`. Кроме того, `JPanel` удобно использовать как поверхность для рисования.

В наших программах мы будем использовать окно, содержащее одну панель. Для со-

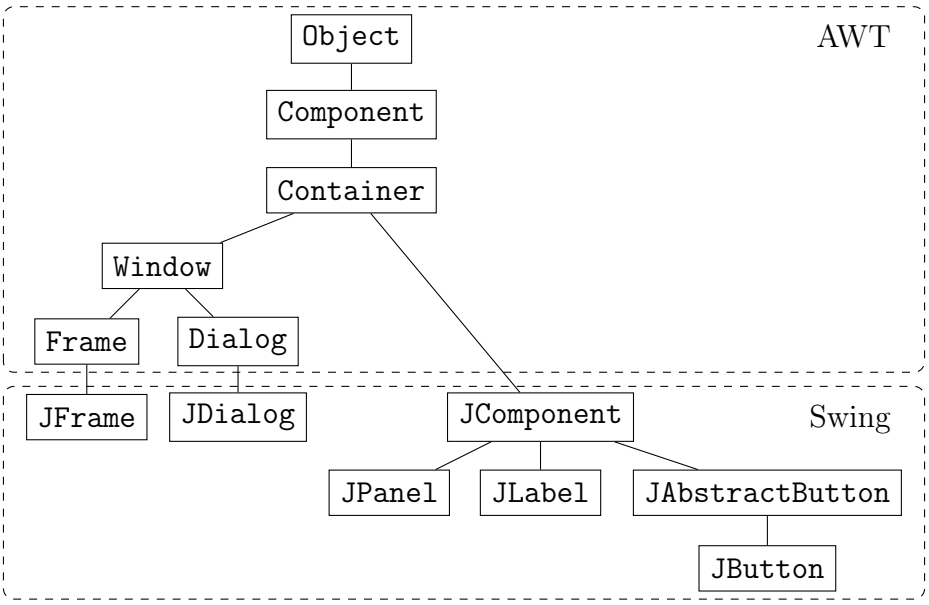


Рис. 2.1. Некоторые классы, соответствующие элементам интерфейса

```
package drawing;

import javax.swing.JFrame;
import javax.swing.SwingUtilities;

public class Drawing {

  private static void createAndShowGUI() {
    JFrame frame = new JFrame("Рисунок");
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    DrawingPanel panel = new DrawingPanel();
    frame.add(panel);
    frame.pack();
    frame.setVisible(true);
  }

  public static void main(String[] args) {
    SwingUtilities.invokeLater(new Runnable() {
      @Override
      public void run() { createAndShowGUI(); }
    });
  }
}
```

Рис. 2.2. Класс с методом `main()`

здания окна будет использоваться стандартный класс `JFrame`. Для панели мы создадим подкласс `DrawingPanel` класса `JPanel`, в котором изменим стандартный размер, цвет фона и (в следующем разделе) прорисовку. Главный класс программы и `DrawingPanel` поместим в пакет `drawing`.

2.1. `JFrame`: создание и вывод на экран

Главный класс программы, содержащий метод `main()`, показан на рис. 2.2. Метод `createAndShowGUI()` создает окно приложения и выводит его на экран. Это включает в себя следующие шаги.

1. Создать объект класса `JFrame`. Конструктору можно передать заголовок окна.
2. Сделать так, чтобы при закрытии окна работа программы заканчивалась. По умолчанию программа продолжает работать.
3. Добавить остальные элементы интерфейса.
4. Сделать размер окна соответствующим размерам содержащихся в нем компонентов.
5. Показать окно.

Устройство окна `JFrame` довольно сложное, поскольку в него встроен целый ряд вспомогательных контейнеров. Наиболее важным из них является *панель содержимого*, которая содержит все элементы интерфейса, кроме меню вверху окна, если оно присутствует. Панель содержимого возвращается методом `getContentPane()`, и именно к ней

добавляется панель `DrawingPanel`. Вызов `frame.add(...)` на самом деле эквивалентен `frame.getContentPane().add(...)`. Вместо добавления к панели содержимого можно также заменить ее на объект `DrawingPanel` следующим образом.

```
frame.setcontentpane(new DrawingPanel());
```

2.2. Многопоточность и графический интерфейс

Запуск метода `createAndShowGUI()` из `main()` выглядит довольно сложно. Это связано с использованием *потоков* в программах на Swing. Поток — это легковесный процесс. В отличие от настоящих процессов, которые обычно представляют отдельные программы, несколько потоков могут относиться к одной программе и иметь общее адресное пространство, то есть обращаться к одним и тем же структурам данных.

Потоки имеют важное значение в Swing. Программы с графическим интерфейсом управляются событиями, такими как нажатие кнопки или необходимость перерисовать окно. Программа заносит события в очередь и затем исполняет их в процессе получения. Обработка событий выполняется в так называемом потоке диспетчеризации событий. Для того, чтобы интерфейс был отзывчивым, в этом потоке нельзя исполнять действия, занимающие много времени, такие как загрузка большого файла с диска или из Интернета. Для таких действий следует создавать отдельные потоки.

Поскольку последовательность действий в программе с несколькими потоками может быть труднопредсказуемой, требуется обращать особое внимание, когда разные потоки обращаются к одной и той же структуре данных в памяти. Классы, отвечающие за компоненты Swing, не являются *потокобезопасными*, то есть обращение к ним из разных потоков в неправильном порядке может привести к повреждению интерфейса. Это сделано специально, потому что попытки сделать библиотеку интерфейса потокобезопасной приводят к ее усложнению и замедлению. Поэтому для корректной работы компонентов Swing требуется, чтобы их создание и любое изменение выполнялись из потока диспетчеризации событий. Это гарантирует правильную последовательность действий.

Метод `main()` исполняется в главном потоке программы, отличном от потока диспетчеризации. Следовательно, для выполнения правила об однопоточном доступе к компонентам Swing, метод `main()` должен попросить поток диспетчеризации запланировать создание интерфейса, описанное в методе `createAndShowGUI()`. Это делается с помощью статического метода `SwingUtilities.invokeLater(Runnable doRun)`. Он принимает объект любого класса, реализующего интерфейс `Runnable`. Этот интерфейс объявляет единственный метод `void run()`. Программа на рис. 2.2 создает анонимный класс для передачи `createAndShowGUI()` методу `invokeLater()`, как описано в разделе 1.10[4].

Раньше считалось допустимым создавать (но не изменять впоследствии) интерфейс в главном потоке, то есть непосредственно в методе `main()`. Вероятность повреждения интерфейса при этом чрезвычайно мала, но разработчики Swing приняли решение, что любой доступ к компонентам, включая их создание, должен осуществляться только из потока диспетчеризации.

Более подробно о потоках и о запуске программы на Swing можно прочитать в [5].

Классы и их члены	Примечания
Класс `java.awt.Dimension`	Размер прямоугольной области
Поля	
`int width`	Ширина
`int height`	Высота
Конструктор	
`Dimension(int width, int height)`	
Класс `java.awt.Color`	Цвет
Примеры констант	
`final static Color WHITE`	Белый
`final static Color LIGHT_GRAY`	Светло-серый
`final static Color BLUE`	Синий
Конструктор	
`Color(int r, int g, int b)`	Красная, зеленая и синяя компоненты цвета, каждая от 0 до 255
Класс `java.awt.Component`	
Методы	
`void setSize(int width, int height)`	Устанавливает размер
`void setSize(Dimension d)`	
Класс `javax.swing.JComponent`	Подкласс `java.awt.Component` и надкласс `javax.swing.JPanel`
Методы	
`void setPreferredSize(Dimension ps)`	Устанавливает предпочтительный размер
`Dimension getPreferredSize()`	Возвращает предпочтительный размер
`int getHeight()`	Возвращает высоту компонента
`int getWidth()`	Возвращает ширину компонента
`void setBackground(Color bg)`	Устанавливает цвет фона
`void setOpaque(boolean isOpaque)`	Устанавливает непрозрачность

Рис. 2.3. Классы и их члены, используемые для установки размера и цвета `JPanel`

2.3. `JPanel`: установка размера

Следующей задачей является задание размера и цвета фона панели `DrawingPanel`. Это делается с помощью классов и их членов, показанных на рис. 2.3.

Swing имеет гибкий механизм определения размера и положения компонентов в контейнере, называемый *диспетчером компоновки*. Вместо того, чтобы указывать абсолютный размер компонента, можно задать предпочтительный, минимальный и максимальный размеры и предоставить диспетчеру определить окончательный размер и положение. Swing содержит несколько диспетчеров компоновки, каждый из которых реализует интерфейс `LayoutManager`. Одни из них располагают компоненты по строчкам, как слова в тексте, другие собирают их в таблицу и т. д.

По умолчанию диспетчером компоновки панели содержимого, к которой добавляется `DrawingPanel`, является класс `BorderLayout`. Он помещает компоненты в центр или по краям контейнера. Если используется метод `add()` с одним параметром, как на рис. 2.2,

[4]Метод `SwingUtilities.invokeLater()` просто вызывает `EventQueue.invokeLater()`, поэтому часто можно увидеть и эту конструкцию.

то добавляемый компонент помещается в центр контейнера. Компонент в центре занимает все пространство, оставшееся от компонентов, расположенных по краям. Таким образом, если других компонентов нет, то компонент в центре занимает весь контейнер.

Каждый компонент, находящийся в контейнере, информирует диспетчер компоновки данного контейнера о своем размере с помощью трех методов: `getMinimalSize()`, `getPreferredSize()` и `getMaximalSize()`, которые возвращают минимальный, предпочтительный и максимальный размеры, соответственно. Однако эти значения являются только рекомендациями, и не все диспетчеры компоновки их учитывают. Тем не менее, метод `pack()` класса `JFrame` (унаследованный от `Window`), устанавливает размер окна таким, чтобы размеры всех компонентов, расположенных в окне, были не меньше возвращаемых методом `getPreferredSize()`. Таким образом, когда окно в программе на рис. 2.2 показывается в первый раз, оно содержит панель со своим предпочтительным размером. В дальнейшем пользователь может изменить размер окна, но панель по-прежнему будет занимать все окно.

Есть два способа заставить `getPreferredSize()` возвращать требуемый размер. Во-первых, можно вызвать метод `setPreferredSize(size)`; в этом случае последующие вызовы `getPreferredSize()` будут возвращать `size`. Метод `setPreferredSize()` является открытым (`public`), поэтому его можно вызвать в конструкторе подкласса панели или из другого класса, например, в методе `createAndShowGUI()` на рис. 2.2. Во-вторых, можно переопределить `getPreferredSize()` в подклассе `JPanel`. Для более сложных компонентов второй способ является предпочтительным, поскольку сам компонент может лучше определить оптимальный размер, который может меняться со временем в зависимости от своего внутреннего состояния.

Наконец, размер компонента можно также задать с помощью методов `setSize()` или `setBounds()` класса `Component`. Однако этот способ имеет смысл использовать лишь для главного окна программы, то есть `JFrame`, а также для компонентов, находящихся в контейнере, который не имеет диспетчера компоновки. Во втором случае программист должен явно задать координаты и размер каждого компонента, и такой способ компоновки называется абсолютным позиционированием.

Размер компонентов можно получить с помощью методов `getSize()`, `getWidth()` и `getHeight()`. Все размеры, используемые в рис. 2.3, измеряются в пикселях.

2.4. `JPanel`: установка цвета фона и прозрачности

Цвет панели устанавливается методом `setBackground()`. В качестве аргумента можно указать константы, определенные в классе `Color`, или создать цвет, передав его красную, зеленую и синюю составляющие конструктору класса `Color`.

Каждый компонент Swing имеет логическое свойство «непрозрачность», устанавливаемое методом `setOpaque(boolean isOpaque)`. Применительно к `JPanel` объявление панели непрозрачной имеет двойной эффект. Во-первых, оно гарантирует Swing, что панель закрасит все пиксели в пределах своих границ, поэтому не требуется рисовать другие компоненты, которые могут находиться под данной панелью. Это делает прорисовку интерфейса более эффективной. Второй эффект связан с рисованием самой панели. У каждого компонента Swing есть рисующий его метод `paintComponent`, подробнее о котором речь идет в следующем разделе. Работа этого метода зависит от текущего стиля оформления. Если объект класса `JPanel` объявлен непрозрачным, то `paintComponent` закрашивает панель цветом фона. В противном случае сквозь нее будет видна панель содержимого, которая обычно имеет светло-серый цвет.

```
package drawing;

import javax.swing.JPanel;
import java.awt.Color;
import java.awt.Dimension;

public class DrawingPanel extends JPanel {
  public DrawingPanel() {
    setBackground(Color.WHITE);
    setOpaque(true);
  }

  @Override
  public Dimension getPreferredSize() {
    return new Dimension(500, 500);
  }
}
```

Рис. 2.4. Подкласс `JPanel`

Прозрачность панели может зависеть от стиля оформления. В большинстве случаев `JPanel` является непрозрачной по умолчанию, но это не гарантировано, поэтому рекомендуется явно вызывать `setOpaque(true)`.

Полный код простейшего варианта класса `DrawingPanel` приведен на рис. 2.4. Вместе с классом `Drawing` на рис. 2.2 он образует программу, которая выводит на экран окно заданного размера с белым фоном.

Упражнения

2.1. Поместите оба класса `Drawing` и `DrawingPanel` в один файл, убрав модификатор `public` у одного из них.

2.2. Сделайте `DrawingPanel` единственным классом (не считая анонимного класса в теле `main()`), поместив в него методы `createAndShowGUI()` и `main()`.

2.3. Сделайте класс `DrawingPanel` вложенным в класс `Drawing`. Должен ли он быть статическим вложенным или внутренним?

3. Рисование в терминах пикселей

Как сказано в предыдущем разделе, прорисовкой компонента занимается метод `paintComponent(Graphics g)` класса `JComponent`. Поскольку метод является защищенным (`protected`), он доступен в подклассах `JComponent` и может быть переопределен в них. Именно в этих переопределенных методах можно рисовать на поверхности компонента.

Напрямую вызывать метод `paintComponent()` не следует. Java вызывает его, когда компонент нужно перерисовать: например, когда окно показывается в первый раз или меняет размер. Если внутреннее состояние компонента изменилась и требуется перерисовка, программа может запросить ее с помощью метода `repaint()`, определенном в классе `Component`.

Единственный параметр метода `paintComponent` имеет тип `java.awt.Graphics`. Передаваемый этому методу аргумент называется графическим контекстом. Об этом объекте можно думать, как о поверхности компонента. Он содержит состояние инструментов рисования: текущие цвет и шрифт, область рисования, режим рисования (как вновь рисуемые точки комбинируются с существующими) и другую информацию.

На самом деле, метод `paintComponent()` рисует компонент не сам, а поручает это *представителю пользовательского интерфейса*, который изображает компонент в соответствии с текущем стилем оформления. В случае `JPanel` представителю пользовательского интерфейса закрашивает панель цветом фона, если панель объявлена непрозрачной. Следовательно, при переопределении `paintComponent()` в подклассе `JPanel` первым делом следует вызвать переопределяемый метод.

```
@Override
protected void paintComponent(Graphics g) {
  super.paintComponent(g);
  ...
}
```

Нет ничего страшного, если часть рисунка выйдет за пределы панели: она будет просто отброшена. На самом деле, графический контекст содержит прямоугольник, который следует перерисовать: он возвращается методом `getClipBounds()`. Метод `paintComponent()` может использовать эту информацию и рисовать только внутри данного прямоугольника для большей эффективности.

Класс `Graphics` содержит методы для рисования отрезков, прямоугольников, многоугольников, эллипсов и их дуг. Они показаны на рис. 3.1. Есть также методы для вывода текста и растровых изображений.

Начало координат находится в левом верхнем углу компонента, ось x направлена вправо, а ось y — вниз. Все координаты, используемые в методах класса `Graphics`, указываются в пикселях и имеют тип `int`. Углы α и β, являющиеся аргументами `drawArc()`, измеряются в градусах и также имеют тип `int`. Как сказано в упражнении 1.4, α и β являются настоящими величинами углов, только если дуга есть часть окружности.

В дополнении к методам на рис. 3.1, в классе `Graphics` есть методы `fillRect()`, `fillPolygon()`, `fillOval()` и `fillArc()`, которые закрашивают соответствующие фигуры текущим цветом. Последний метод закрашивает сектор эллипса (фигуру, ограниченную дугой и двумя радиусами).

Цвет рисования устанавливается методом `setColor(Color c)` класса `Graphics`.

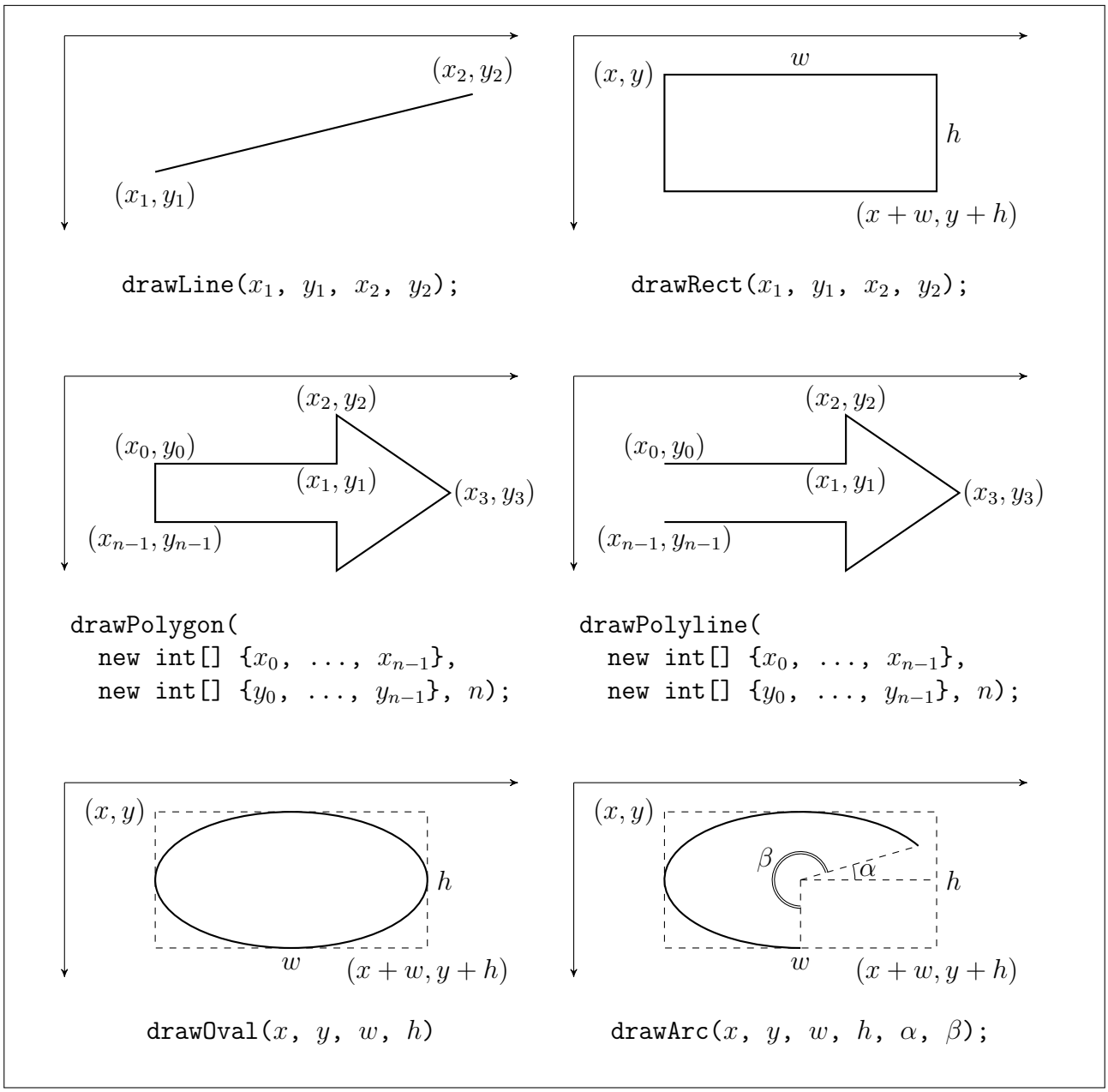


Рис. 3.1. Некоторые методы для рисования в классе `Graphics`

Упражнения

3.1. Дополните программу на рис. 2.2 и 2.4 так, чтобы она рисовала окружность в центре окна, касающуюся по крайней мере двух противоположных сторон окна. Убедитесь, что изображение перерисовывается правильным образом, когда вы меняете размер окна с помощью мыши.

3.2. Напишите программу, рисующую картинку, изображенную на рис. 3.2 (без осей координат и координатной решетки).

3.3. Класс `JComponent`, являющийся родительским для всех компонентов Swing, не имеет представителя пользовательского интерфейса. Поэтому метод `paintComponent()` класса

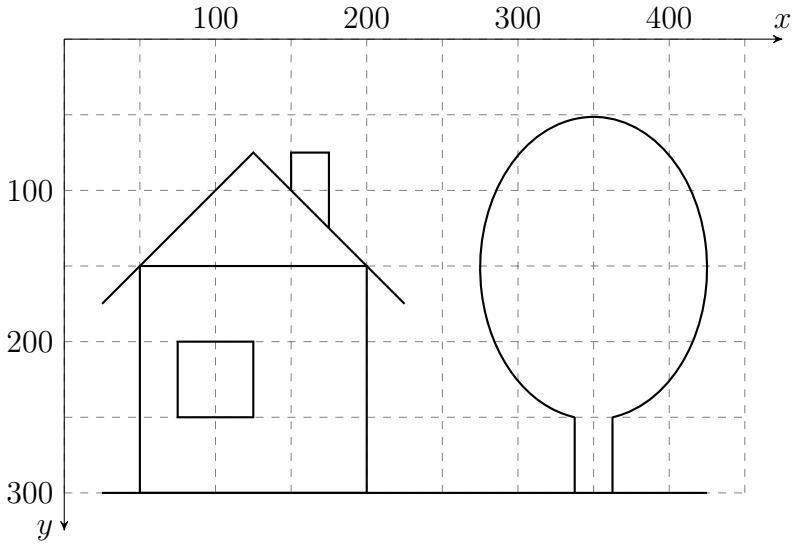


Рис. 3.2. Пример рисунка

`JComponent` ничего не делает. Однако `JComponent` можно также использовать как поверхность для рисования. Замените `JPanel` на `JComponent` в классе на рис. 2.4, а вместо вызова `super.paintComponent(g)` закрасьте все окно цветом фона, который возвращается методом `getBackground()`. Измените цвет рисования на черный и нарисуйте обе диагонали окна.

4. Векторная графика

4.1. Классы и методы векторной графики

Рисование с помощью методов класса `Graphics` имеет ряд недостатков:
1) невозможность изменить ширину и стиль линии,
2) отсутствие методов для рисования кривых, кроме дуг эллипсов,
3) невозможность нарисовать эллипс, повернутый на угол, не кратный 90°,
4) погрешность из-за того, что углы, определяющие дугу в методе `drawArc`, являются целыми числами,
5) отсутствие представления сложной фигуры как единого целого,
6) необходимость явного преобразования координат из пользовательского пространства (например, из сантиметров) в пиксели.

Библиотека AWT поддерживает другой подход к рисованию, использующий описание фигур с помощью абстрактных координат и параметров типа `float` или `double`. Сложные фигуры строятся из более простых. Такое математическое описание фигур называется *векторной графикой*.

Классы, описывающие базовые и составные фигуры, находятся в пакете `java.awt.geom`. Почти все они реализуют интерфейс `java.awt.Shape`. Таким образом, программа, использующая методы, объявленные в `Shape`, является полиморфной и может работать с разными фигурами. Интерфейс `Shape` описывает фигуру, построенную из отрезков прямых, дуг эллипсов, а также *кривых Безье* второго и третьего порядка.[5] Фигура не обязана быть непрерывной, то есть она может состоять из нескольких связных компонент. Среди методов, объявленных в `Shape`, представляют интерес `getBounds2D()`, возвращающий прямоугольник, содержащий данную фигуру, и `contains(double x, double y)`, определяющий, находится ли точка с координатами (x, y) внутри фигуры. Есть также методы, последовательно возвращающие каждый из отрезков или кривых, составляющих фигуру.

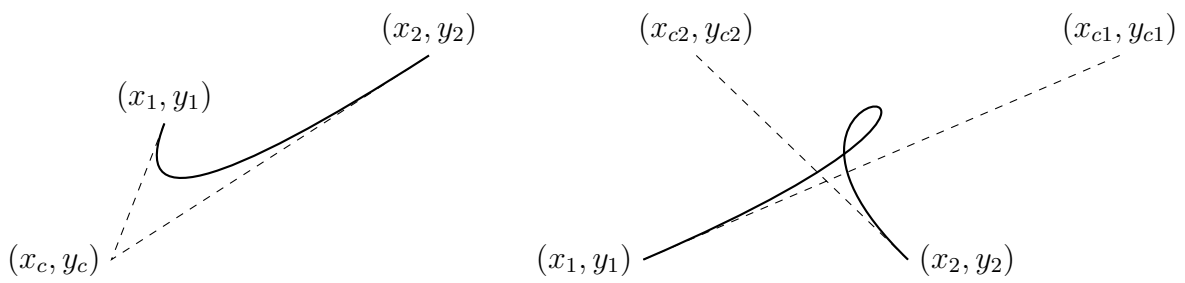
Как описано в §1.10, класс `java.awt.geom.Point2D` содержит статические вложенные классы `Point2D.Float` и `Point2D.Double`, описывающие точку на плоскости с координатами типа `float` и `double`, соответственно. Несколько необычным решением является то, что будучи вложенными в `Point2D`, эти два класса также являются его подклассами. Сам `Point2D` является абстрактным, поэтому программисту нужно создавать объекты либо `Point2D.Float`, либо `Point2D.Double`, но присваивать их можно переменной `Point2D`, что позволяет писать полиморфный код.

Для большинства задач графики достаточно точности типа `float`, однако работа с этим типом имеет ряд неудобств. Присваивание значения типа `double` переменной типа `float` может повлечь потерю точности, поэтому в таком случае компилятор сигнализирует об ошибке. Чтобы ее избежать, нужно сделать приведение типа. Например, метод `getX()` класса `Point2D` возвращает тип `double`, поэтому корректное присваивание может выглядеть следующим образом.

```
float x = (float) p.getX();
```

Константы с десятичной точкой по умолчанию имеют тип `double`, поэтому они также не могут быть присвоены переменным типа `float`. Константы типа `float` должны заканчиваться суффиксом `f` или `F`, например, `3.1415f`. По этим причинам в данной работе мы будем пользоваться классами с полями типа `double`.

[5] На самом деле вместо дуг эллипсов рисуются их приближения с помощью кривых Безье.



`QuadCurve2D.Double(`
x_1, y_1, x_c, y_c, x_2, y_2)

`CubicCurve2D.Double(`
x_1, y_1, x_{c1}, y_{c1}, x_{c2}, y_{c2}, x_2, y_2)

Рис. 4.1. Кривые Безье

```
Point2D.Double(x, y)
Line2D.Double(x1, y1, x2, y2)
Rectangle2D.Double(x, y, w, h)
Ellipse2D.Double(x, y, w, h)
Arc2D.Double(x, y, w, h, start, extent, type)
QuadCurve2D.Double(x1, y1, xc, yc, x2, y2)
CubicCurve2D.Double(x1, y1, xc1, yc1, xc2, yc2, x2, y2)
```

Рис. 4.2. Некоторые классы, представляющие фигуры на плоскости

Пакет `java.awt.geom` содержит классы `Line2D` (отрезок), `Rectangle2D` (прямоугольник), `Ellipse2D` (эллипс), `Arc2D` (дуга эллипса), а также `QuadCurve` и `CubicCurve` (кривые Безье второй и третьей степени, соответственно). Они устроены по той же схеме, что и `Point2D`: каждый из них содержит статические вложенные классы `Float` и `Double`, которые расширяют класс, в который они вложены. Все эти классы, кроме `Point2D` и его подклассов, реализуют интерфейс `Shape`.

Кривые Безье n-го порядка — это способ задания параметрических кривых, в котором каждая координата выражается многочленом n-й степени от параметра. Кривая задается своей начальной и конечной точками, а также $n - 1$ *контрольными точками*; все вместе они называются *опорными точками*. Отрезок, соединяющий начальную и первую контрольную точки касается кривой в начальной точке и, аналогично, отрезок между последней контрольной и конечной точками касается кривой в конечной точке. Кроме того, длины этих отрезков определяют «скорость» параметризуемой точки в начале и конце кривой. Кривые Безье первого порядка являются отрезками, второго порядка — параболами, но кривые третьего порядка не обязательно являются графиками гладких функций: они могут иметь самопересечения и особые точки. На рис. 4.1 показаны квадратичная кривая с контрольной точкой (x_c, y_c) и кубическая кривая с контрольными точками (x_{c1}, y_{c1}) и (x_{c2}, y_{c2}), а также аргументы конструкторов соответствующих классов.

`Path2D.Double()` (конструктор)	Создает пустую фигуру
`void moveTo(double x, double y)`	Перемещает текущую точку в (x, y)
`void lineTo(double x, double y)`	Соединяет текущую точку с (x, y) отрезком
`void quadTo(double x1, double y1, double x2, double y2)`	Соединяет текущую точку с (x_2, y_2) с помощью квадратичной кривой Безье с контрольной точкой (x_1, y_1)
`void curveTo(double x1, double y1, double x2, double y2, double x3, double y3)`	Соединяет текущую точку с (x_3, y_3) с помощью кубической кривой Безье с контрольными точками (x_1, y_1) и (x_2, y_2)
`void append(Shape s, boolean connect)`	Добавляет фигуру `s`. Если `connect == true`, соединяет текущую точку и `s` отрезком
`void closePath()`	Соединяет отрезком текущую точку с точкой из последнего `moveTo`
`boolean contains(double x, double y)`	Возвращает `true`, если точка (x, y) лежит внутри фигуры
`Rectangle2D getBounds2D()`	Возвращает (не обязательно наименьший) прямоугольник, содержащий фигуру

Рис. 4.3. Некоторые методы `Path2D` и `Path2D.Double`

Аргументы конструкторов описанных классов показаны на рис. 4.2. Все аргументы, кроме `type` у класса `Arc2D.Double`, имеют тип `double`. Аргументы конструкторов `Line2D.Double`, `Rectangle2D.Double`, `Ellipse2D.Double` и `Arc2D.Double` имеют тот же смысл, что и для соответствующих методов класса `Graphics`, показанных на рис. 3.1 (где используется `Oval` вместо `Ellipse`). Аргумент `type` конструктора `Arc2D.Double` имеет тип `int` и должен быть равен одной из констант: `Arc2D.OPEN`, `Arc2D.CHORD` или `Arc2D.PIE`, как описано в упражнении 1.4.

Наконец, `java.awt.geom` содержит абстрактный класс `Path2D`, также имеющий вложенные подклассы `Float` и `Double`. Основные методы `Path2D` и `Path2D.Double` приведены на рис. 4.3. Эти классы позволяют собирать фигуры, описанные выше, в составные фигуры. Конструктор `Path2D.Double()` создает пустую фигуру. Далее, у каждой фигуры есть текущая точка, которую можно передвигать или соединять с новой текущей точкой отрезком или кривой. Наконец, с помощью метода `append` к данной фигуре можно добавлять другую. Если второй аргумент метода равен `true`, то начальная точка добавляемой фигуры соединяется отрезком с текущей точкой. Начальная точка фигуры определяется первым вызовом `moveTo`; в случае присоединения отрезком он заменяется на `lineTo`.

У `Graphics` есть подкласс `java.awt.Graphics2D`, который существенно расширяет возможности рисования. Оба эти класса являются абстрактными, и в реальности графический контекст, передаваемый методу `paintComponent`, является объектом некоторого подкласса `Graphics2D`. Чтобы получить доступ к методам `Graphics2D`, нужно привести тип графического контекста к этому классу.

```
@Override
protected void paintComponent(Graphics g) {
  super.paintComponent(g);
  Graphics2D g2 = (Graphics2D)g;
  ...
}
```

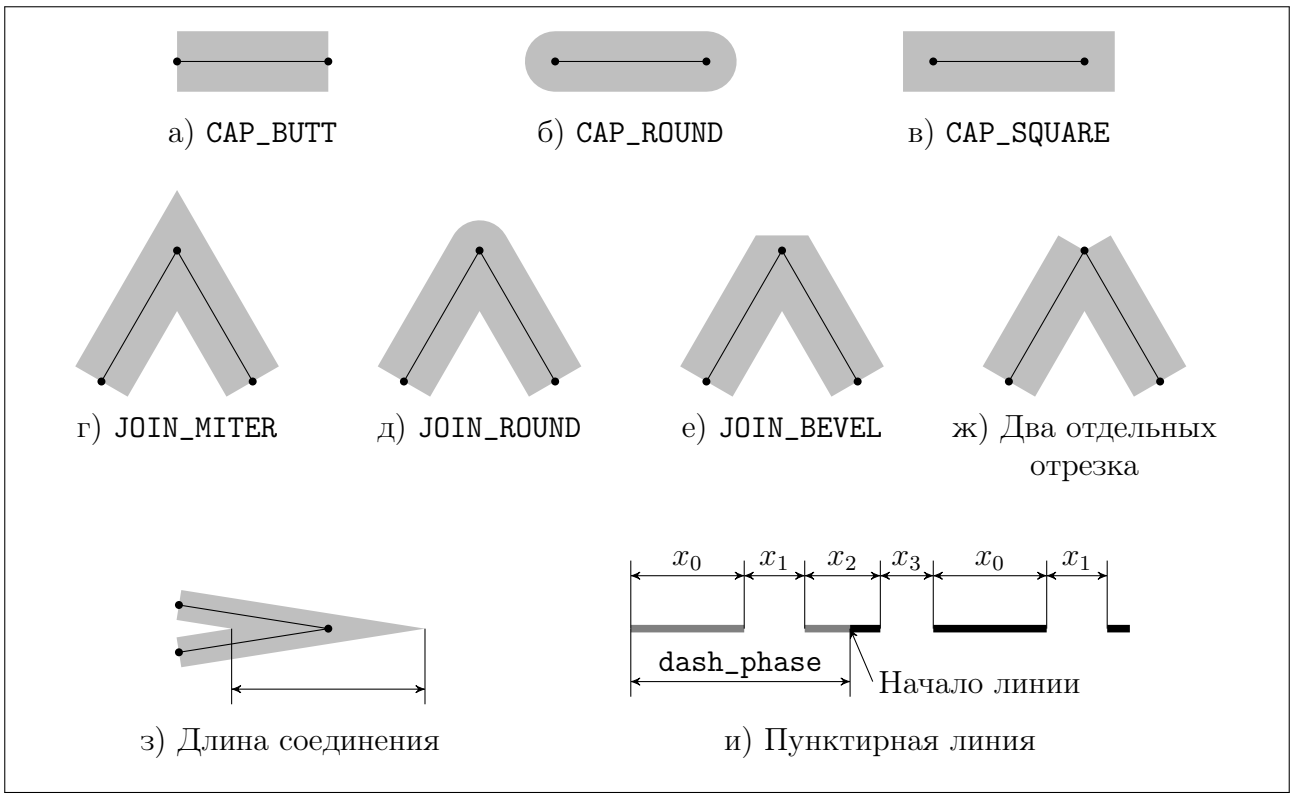


Рис. 4.4. Стили линий и соединений

Объекты любого класса, реализующего `Shape`, можно нарисовать или закрасить текущим цветом с помощью методов `draw(Shape)` и `fill(Shape)` класса `Graphics2D`. Разумеется, можно использовать и методы рисования, объявленные в `Graphics`.

4.2. Стиль линии

Класс `Graphics2D` может контролировать толщину и стиль линий. Для этого нужно создать объект класса `java.awt.BasicStroke` и передать его методу `setStroke` класса `Graphics2D`. Конструктор `BasicStroke` принимает от 0 до 6 аргументов.

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

Первый аргумент задает ширину линии. Второй и третий аргументы задают стили окончания и соединения линий, соответственно. Они имеют важное значение для широких линий. Возможные значения, являющиеся статическими константами класса `BasicStroke`, и их эффекты показаны на рис. 4.4. Стиль соединения используется только для сегментов одной и той же фигуры, полученных с помощью методов `lineTo`, `quadTo` и `curveTo` и `closePath`. Если же нарисовать два отдельных отрезка, оканчивающихся в одной точке, то соединение может выглядеть неприглядно, как на на рис. 4.4, ж). Поэтому в замкнутой фигуре для рисования последнего отрезка важно использовать метод `closePath`, а не `lineTo`.

Если два отрезка соединяются под очень острым углом и используется стиль соединения `JOIN_MITER`, то может получиться достаточно длинный «клин», как показано на рис. 4.4, з). Чтобы предотвратить этот нежелательный эффект, используется параметр

miterlimit. Если отношение длины соединения к ширине линии больше miterlimit, то вместо стиля JOIN_MITER используется JOIN_BEVEL.

Последние два аргумента описывают характер пунктира. Пятый аргумент dash является массивом, в котором содержатся длины последовательных отрезков и пропусков. Указанный рисунок повторяется циклически. Последний аргумент определяет длину начального фрагмента, который будет пропущен. На рис. 4.4, и) показан пунктир, задаваемый следующим массивом.

```
float[] dash = new float[] {x0, x1, x2, x3};
```

Класс BasicStroke имеет конструкторы с 1, 3, 4 и 6 параметрами, а также конструктор по умолчанию (без параметров). По умолчанию определяется непрерывная линия шириной 1.0 со стилем окончания CAP_SQUARE, стилем соединения JOIN_MITER и значением miterlimit, равным 10.0.

Класс Graphics2D имеет несколько параметров, отвечающих за качество рисования. Их значение устанавливается методом setRenderingHint, первым аргументом которого является имя параметра, а вторым — его значение. Наиболее полезным параметром является сглаживание (antialiasing). Оно заключается в закрашивании пикселей на границе фигуры цветом, являющимся промежуточным между цветом фигуры и цветом фона, в результате чего граница фигуры выглядит более гладко. Режим сглаживания включается командой

```
g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                    RenderingHints.VALUE_ANTIALIAS_ON);
```

Следует отметить, что данные параметры являются лишь рекомендацией для реализации, поэтому они могут учитываться не всегда.

Упражнения

4.1. Измените программу из упражнения 3.2 так, чтобы она создавала один объект класса Path2D.Double, представляющий всю фигуру на рис. 3.2, и рисовала его с помощью метода draw.

4.2. Вызов shape1.append(shape2, true) добавляет фигуру shape2 к shape1 и соединяет отрезком текущую точку shape1 и начальную точку shape2. Выясните экспериментально, какая точка является начальной в Rectangle2D.Double, Ellipse2D.Double и Arc2D.Double.

4.3. Найдите минимальный угол пересечения (в градусах), при котором еще используется стиль соединения JOIN_MITER, если используется конструктор BasicStroke по умолчанию.

4.4. Меняя параметр dash_phase конструктора BasicStroke несколько раз в секунду, можно добиться того, что пунктир будет двигаться. Раздел 7 описывает, как использовать таймер Swing для создания анимации. Используя эту информацию, напишите программу, рисующую движущимся пунктиром треугольник со скругленными углами, как на рис .4.5.

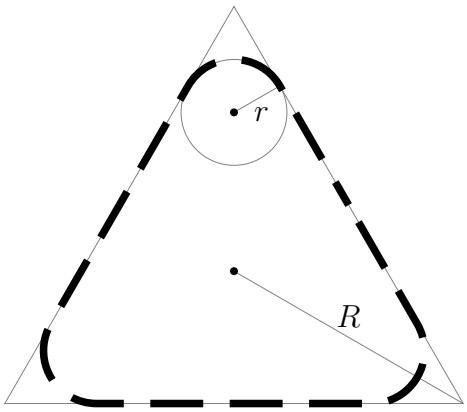


Рис. 4.5. К упражнению 4.4 (движущийся пунктир)

Используйте следующие размеры в пикселях.

$R = 200$	радиус описанной окружности
$r = 40$	радиус дуги, вписанной в угол
$d_0 = 40$	длина отрезков пунктира
$d_1 = 20$	длина пробелов между отрезками пунктира
$w = 10$	ширина линии
$v = 40$	скорость движения пунктира (в пикселях в секунду)

Для того, чтобы пунктир был однородным, нужно описать всю фигуру как один объект класса `Path2D.Double` и затем нарисовать ее методом `draw` класса `Graphics2D`, предварительно установив нужный стиль линии. Допускается наличие одного места (начала и конца фигуры), где длина отрезков пунктира может меняться.

5. Аффинные отображения

5.1. Определения и свойства

Этот раздел содержит напоминание фактов об аффинных отображениях на плоскости, которые необходимы для преобразования координат при рисовании. Дополнительную информацию об аффинных пространствах можно найти в [2, гл. 4].

Определение 1. Система координат на плоскости задается репером (O, e_1, e_2), состоящим из точки O, называемой началом координат, и неколлинеарных векторов e_1, e_2, то есть базиса. Будем обозначать реперы буквой $\mathcal{F}$, возможно, с индексами или штрихами. Координатами точки P в репере $\mathcal{F} = (O, e_1, e_2)$ называются координаты вектора $\overrightarrow{OP}$ в базисе (e_1, e_2). Будем обозначать координаты P в $\mathcal{F}$ через $[P]\mathcal{F}$.

Координаты будем рассматривать в виде вектора-столбца, но иногда записывать в виде строки для экономии места.

Теорема 2. Пусть $\mathcal{F} = (O, e_1, e_2)$ и $\mathcal{F}' = (O', e_1', e_2')$ — два репера. Тогда существует единственная матрица $C_{\mathcal{F}'}^{\mathcal{F}}$ размера 2×2, такая что

$$[P]\mathcal{F} = C_{\mathcal{F}'}^{\mathcal{F}}[P]\mathcal{F}' + [O']\mathcal{F} \tag{1}$$

для произвольной точки P.

Матрица $C_{\mathcal{F}'}^{\mathcal{F}}$ называется матрицей перехода от базиса (e_1, e_2) к (e_1', e_2'). Обозначения $C_{\mathcal{F}'}^{\mathcal{F}}$ и $[P]\mathcal{F}'$ в (1) выбраны по аналогии с физическими размерностями. Например, $2{,}54\frac{\text{см}}{\text{дюйм}}$ — это коэффициент перевода дюймов в сантиметры, поэтому $7{,}62\,\text{см} = 2{,}54\frac{\text{см}}{\text{дюйм}} \cdot 3\,\text{дюйма}$.

Следующий факт доказывается непосредственной проверкой.

Утверждение 3. Имеет место эквивалентность.

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix} \begin{pmatrix} x' \\ y' \end{pmatrix} + \begin{pmatrix} x_0 \\ y_0 \end{pmatrix} \iff \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} & x_0 \\ c_{21} & c_{22} & y_0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix}$$

Следствие 4. В условиях теоремы 2

$$\begin{pmatrix} [P]\mathcal{F} \\ 1 \end{pmatrix} = \begin{pmatrix} C_{\mathcal{F}'}^{\mathcal{F}} & [O']\mathcal{F} \\ 0 & 1 \end{pmatrix} \begin{pmatrix} [P]\mathcal{F}' \\ 1 \end{pmatrix}.$$

Определение 5. Блочная 3×3 матрица $\begin{pmatrix} C_{\mathcal{F}'}^{\mathcal{F}} & [O']\mathcal{F} \\ 0 & 1 \end{pmatrix}$ называется матрицей перехода от репера $\mathcal{F}$ к реперу $\mathcal{F}'$ и обозначается $D_{\mathcal{F}'}^{\mathcal{F}}$. Если $[P]\mathcal{F} = (x, y)$, то $(x, y, 1)$ называются однородными координатами точки P в $\mathcal{F}$. В более общем случае, однородными координатами P называются также произвольная тройка (kx, ky, k), где $k \neq 0$.

Если не возникает путаницы, будем использовать $[P]\mathcal{F}$ также для обозначения однородных координат. Таким образом, следствие 4 переписывается как

$$[P]\mathcal{F} = D_{\mathcal{F}'}^{\mathcal{F}}[P]\mathcal{F}'. \tag{2}$$

Определение 6. Отображение f точек на плоскости называется аффинным по отношению к реперам $\mathcal{F}$ и $\mathcal{F}'$, если

$$[P]\mathcal{F} = [f(P)]\mathcal{F}'. \tag{3}$$

Пусть f будет аффинным отображением по отношению к $\mathcal{F} = (O, e_1, e_2)$ и $\mathcal{F}' = (O', e_1', e_2')$. В силу сохранения координат f переводит концы векторов e_i, отложенных от точки O, в концы e_i', отложенных от O'. Таким образом, репер $\mathcal{F}'$ можно назвать образом $\mathcal{F}$ при отображении f и обозначить через $f(\mathcal{F})$. Следовательно, можно говорить про аффинное отображение по отношению к исходному реперу $\mathcal{F}$. На самом деле, можно показать, что если отображение аффинно по отношению к одному реперу, то оно аффинно по отношению к любому реперу.

Будем обозначать аффинные отображения через f, возможно, с индексами или штрихами.

Чтобы связать координаты точки P и ее образа в исходном репере $\mathcal{F}$, заметим, что

$$[f(P)]\mathcal{F} \overset{(2)}{=} D^{\mathcal{F}}_{f(\mathcal{F})}[f(P)](f(\mathcal{F})) \overset{(3)}{=} D^{\mathcal{F}}_{f(\mathcal{F})}[P]\mathcal{F}. \tag{4}$$

Определение 7. Матрица $D^{\mathcal{F}}_{f(\mathcal{F})}$ называется матрицей f в $\mathcal{F}$ и обозначается $[f]\mathcal{F}$.

Таким образом, матрица $D^{\mathcal{F}}_{\mathcal{F}'}$ перехода от $\mathcal{F}$ к $\mathcal{F}'$ используется двумя способами: во-первых, она преобразует координаты точки в $\mathcal{F}'$ в координаты *той же* точки в $\mathcal{F}$ (см. (2)), а во-вторых, она преобразует координаты точки в $\mathcal{F}$ в координаты *ее образа* в $\mathcal{F}$ при аффинном отображении, переводящем $\mathcal{F}$ в $\mathcal{F}'$ (см. (4)).

Если $[f]\mathcal{F} = \begin{pmatrix} C & x_0 \\ 0 & 1 \end{pmatrix}$, то действие f на точку P заключается в следующем: сначала к радиус-вектору P применяется линейное отображение с матрицей C, а затем получившийся вектор откладывается от точки с координатами x_0. Если $x_0 = 0$, то отображение f можно назвать линейным, хотя оно действует не на векторы, а на точки.

Теорема 8.

1. $D^{\mathcal{F}_1}_{\mathcal{F}_3} = D^{\mathcal{F}_1}_{\mathcal{F}_2} D^{\mathcal{F}_2}_{\mathcal{F}_3}$.
2. $(D^{\mathcal{F}_1}_{\mathcal{F}_2})^{-1} = D^{\mathcal{F}_2}_{\mathcal{F}_1}$.
3. $[f_2 \circ f_1]\mathcal{F} = ([f_2]\mathcal{F})([f_1]\mathcal{F})$.
4. Если f — взаимно-однозначное отображение, то $[f^{-1}]\mathcal{F} = ([f]\mathcal{F})^{-1}$.
5. $[f]\mathcal{F}' = (D^{\mathcal{F}}_{\mathcal{F}'})^{-1}([f]\mathcal{F})(D^{\mathcal{F}}_{\mathcal{F}'})$.

Докажем пункт 5. Пусть $D = D^{\mathcal{F}}_{\mathcal{F}'}$, $[f]\mathcal{F} = A$ и $[f]\mathcal{F}' = A'$. Рассмотрим следующие преобразования координат.

$$[P]\mathcal{F}' \overset{D}{\longmapsto} [P]\mathcal{F} \overset{A}{\longmapsto} [f(P)]\mathcal{F} \overset{D^{-1}}{\longmapsto} [f(P)]\mathcal{F}',$$

Значит, $[f(P)]\mathcal{F}' = D^{-1}AD[P]\mathcal{F}'$ для всех P, откуда $A' = D^{-1}AD$, что и требовалось доказать.

Пусть $f_1(\mathcal{F}_1) = \mathcal{F}_2$ и $f_2(\mathcal{F}_2) = \mathcal{F}_3$. Матрицу композиции $f = f_2 \circ f_1$ в $\mathcal{F}_1$ можно представить двумя способами. С одной стороны $[f]\mathcal{F}_1 = ([f_2]\mathcal{F}_1)([f_1]\mathcal{F}_1)$. С другой стороны

$$[f]\mathcal{F}_1 = D^{\mathcal{F}_1}_{\mathcal{F}_3} = D^{\mathcal{F}_1}_{\mathcal{F}_2} D^{\mathcal{F}_2}_{\mathcal{F}_3} = [f_1]\mathcal{F}_1[f_2]\mathcal{F}_2$$

Приравнивая эти выражения, получаем

$$[f_2]\mathcal{F}_1\ [f_1]\mathcal{F}_1 = [f_1]\mathcal{F}_1\ [f_2]\mathcal{F}_2.$$

Итак, записывать матрицы последовательных отображений можно справа налево, причем оба отображения рассматриваются в первом репере, или слева направо, но при этом второе отображение рассматривается во втором репере.

Если $\mathcal{F} = (O, e_1, e_2)$ — ортонормированный репер, то поворот вокруг O на угол α, масштабирования вдоль e_1 и e_2 с коэффициентами λ и μ, соответственно, а также параллельный перенос на вектор с координатами (x, y) являются аффинными отображениями со следующими матрицами в $\mathcal{F}$.

$$R(\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 9 & 0 & 1 \end{pmatrix} \quad S(\lambda,\mu) = \begin{pmatrix} \lambda & 0 & 0 \\ 0 & \mu & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad T(x,y) = \begin{pmatrix} 1 & 0 & x \\ 0 & 1 & y \\ 0 & 0 & 1 \end{pmatrix} \tag{5}$$

Необходимо подчеркнуть, что поворот имеет матрицу, приведенная выше, если базисные векторы имеют одинаковую длину и образуют угол 90°, а направление поворота совпадает с направлением кратчайшего поворота, переводящего e_1 в e_2. Если перевод e_1 в e_2 осуществляется поворотом по часовой стрелке (это часто имеет место в компьютерной системе координат), а поворот производится против часовой стрелке, как принято в математике, то необходимо поменять знак перед $\sin\alpha$.

Если λ или μ в $S(\lambda, \mu)$ отрицательны, то кроме масштабирования отображение включает в себя симметрию относительно соответствующей оси координат. В частности, обозначим симметрии относительно осей Ox и Oy через

$$S_x = S(1, -1), \qquad S_y = S(-1, 1). \tag{6}$$

Матрицы поворота и масштабирования с центром в произвольной точке с координатами (x, y) можно получить с помощью теоремы 8, пункт 5.

$$R_{(x,y)}(\alpha) = T(-x,-y)R(\alpha)T(x,y) \qquad S_{(x,y)}(\lambda,\mu) = T(-x,-y)S(\lambda,\mu)T(x,y) \tag{7}$$

5.2. Преобразование координат из мировых в экранные

В компьютерной графике обычно пользуются двумя системами координат. Мировая система координат описывает реальные физические объекты, и единицами измерения в ней служат, например, метры или сантиметры. Экранная система координат связана с конкретным устройством или окном, и единицей измерения в ней служат пиксели. Требуется найти способ пересчета координат из мировой системы в экранную.

Пример 5.1. В мировой системе координат, где единицами измерения служат метры, задана фигура, показанная на рис. 5.1. Требуется изобразить ее в окне, имеющем высоту h пикселей, границу шириной p пикселей и разрешение r пикселей на сантиметр. Изображение должно быть выполнено в масштабе m метров в одном сантиметре и располагаться в левом нижнем углу окна, то есть левый нижний угол фигуры с мировыми координатами $(x_w, y_w) = (3, 1)$ должен иметь экранные координаты $(x_s, y_s) = (h - p, p)$. Описанное расположение изображено на рис. 5.2.

Пусть $\mathcal{W}$ и $\mathcal{S} = (O, e_1, e_2)$ — реперы, задающие мировую и экранную системы координат, соответственно. Тогда длины векторов re_1 и re_2 равны 1 см. Далее, пусть P —

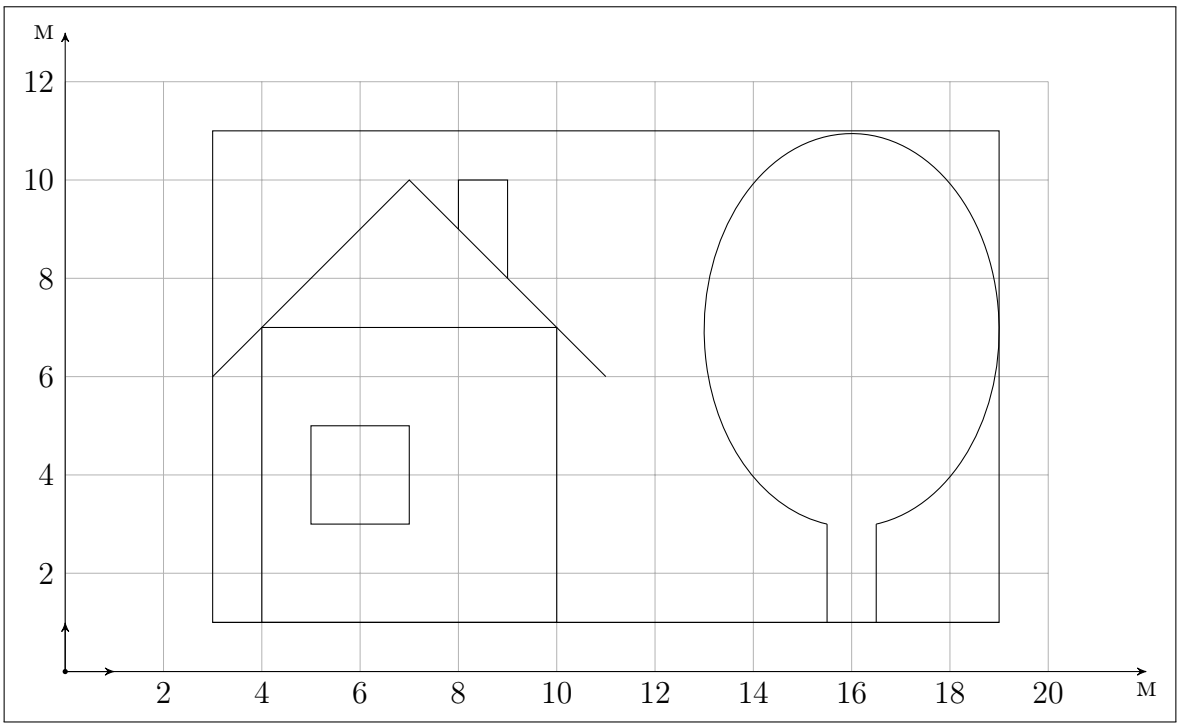


Рис. 5.1. Фигура в мировых координатах

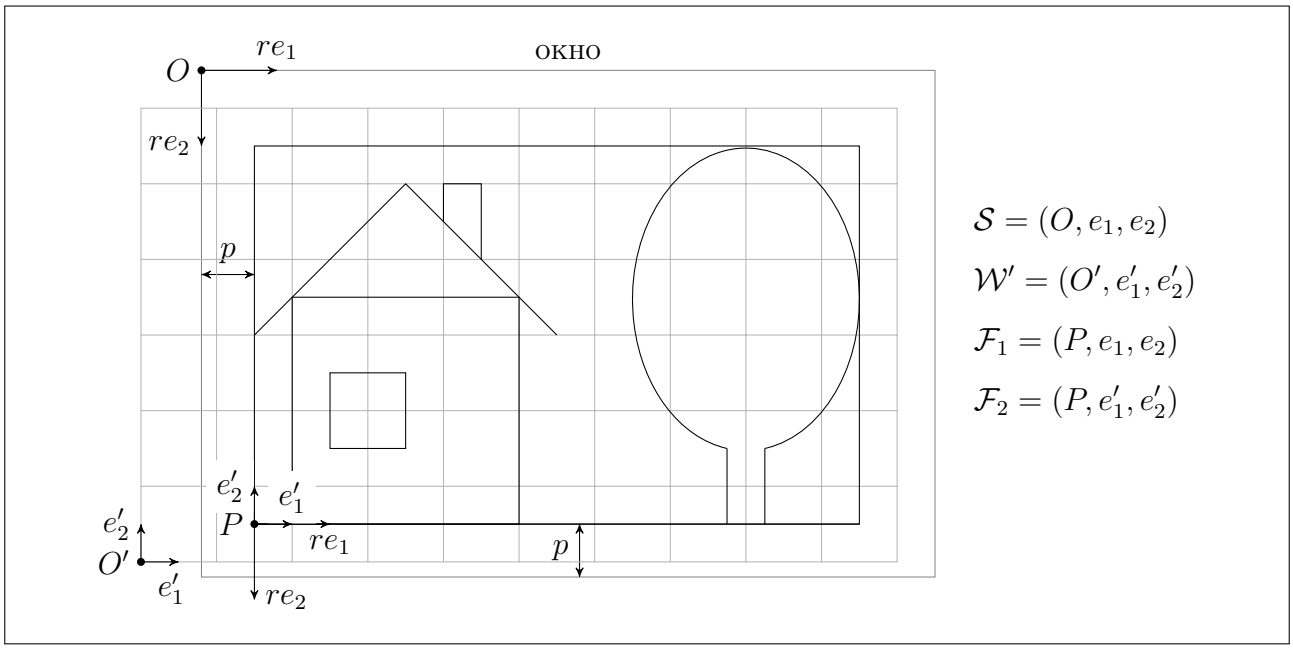


Рис. 5.2. Преобразование координат из мировых в экранные: пример 5.1

точка с экранными координатами (x_s, y_s). Рассмотрим репер $\mathcal{W}' = (O', e'_1, e'_2)$, где $|e'_1| = |e'_2| = 1/m$ см и $[P]\mathcal{W}' = (x_w, y_w)$. Тогда аффинное отображение, переводящее $\mathcal{W}$ в $\mathcal{W}'$, переводит исходную фигуру в требуемое изображение в окне. По определению аффинного отображения координаты любой точки в $\mathcal{W}$ и ее образа в $\mathcal{W}'$ совпадают.

Теперь нужно определить, как преобразуются координаты между $\mathcal{W}'$ и $\mathcal{S}$. Для этого найдем матрицу перехода $D^{\mathcal{S}}_{\mathcal{W}'}$ от $\mathcal{S}$ к $\mathcal{W}'$ и представим ее в виде произведения матриц переходов. Рассмотрим два дополнительных репера $\mathcal{F}_1 = (P, e_1, e_2)$ и $\mathcal{F}_2 = (P, e'_1, e'_2)$ и

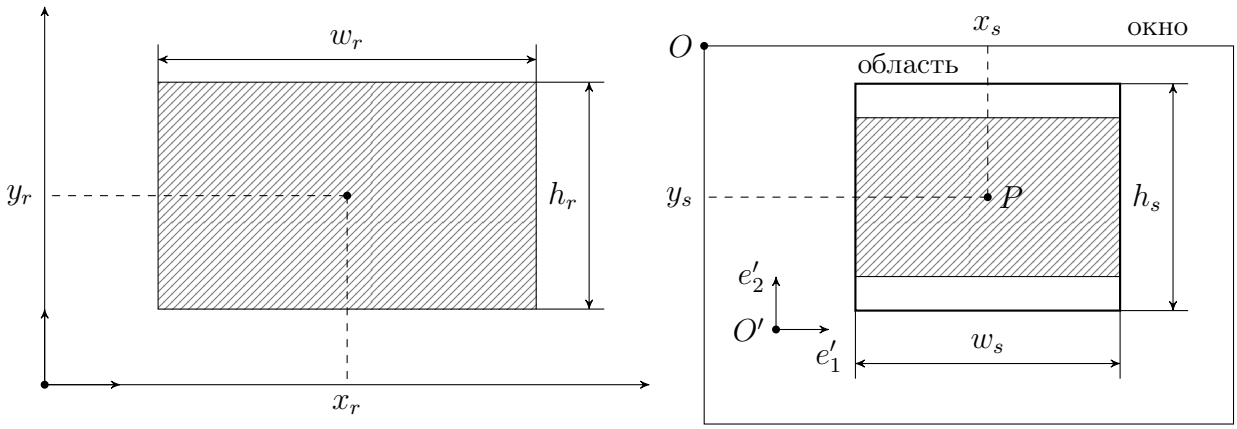


Рис. 5.3. Преобразование координат из мировых в экранные: пример 5.2

представим

$$D^{\mathcal{S}}_{\mathcal{W}'} = D^{\mathcal{S}}_{\mathcal{F}_1} D^{\mathcal{F}_1}_{\mathcal{F}_2} D^{\mathcal{F}_2}_{\mathcal{W}'}.$$

Матрицы переходов можно найти по определению, но о них удобно думать как о матрицах отображений. Напомним, что по определению, если f есть аффинное отображение, переводящее репер $\mathcal{F}$ в $\mathcal{F}'$, то матрица перехода от $\mathcal{F}$ к $\mathcal{F}'$ есть матрица отображения f в репере $\mathcal{F}$:

$$D^{\mathcal{F}}_{\mathcal{F}'} = [f]\mathcal{F}.$$

Значит, $D^{\mathcal{S}}_{\mathcal{F}_1}$ и $D^{\mathcal{F}_2}_{\mathcal{W}'}$ есть матрицы параллельных переносов на векторы (x_s, y_s) и $(-x_w, -y_w)$, соответственно. Что же касается отображения, переводящего $\mathcal{F}_1$ в $\mathcal{F}_2$, оно увеличивает базисные векторы с 1 пикселя до r/m пикселей и осуществляет симметрию по отношению к горизонтальной оси. Следовательно, это масштабирование с центром в точке P и коэффициентами r/m и $-r/m$. Общая матрица перехода есть

$$\begin{aligned} D^{\mathcal{S}}_{\mathcal{W}'} = D^{\mathcal{S}}_{\mathcal{F}_1} D^{\mathcal{F}_1}_{\mathcal{F}_2} D^{\mathcal{F}_2}_{\mathcal{W}'} = T(x_s, y_s) S(r/m, -r/m) T(-x_w, -y_w) = \\ = \begin{pmatrix} 1 & 0 & x_s \\ 0 & 1 & y_s \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} r/m & 0 & 0 \\ 0 & -r/m & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & -x_w \\ 0 & 1 & -y_w \\ 0 & 0 & 1 \end{pmatrix}. \end{aligned} \tag{8}$$

Почему для нахождения отображения, переводящего $\mathcal{S}$ в $\mathcal{W}'$, мы рассматриваем вспомогательные реперы $\mathcal{F}_1$ и $\mathcal{F}_2$? Можно было бы сразу перевести точку O в O' с помощью параллельного переноса, а затем сделать масштабирование с центром в O'. Координаты вектора этого переноса берутся в базисе $\mathcal{S}$, поэтому они равны $[O']\mathcal{S}$. Однако координаты O' в $\mathcal{S}$ требуется вычислить (что достаточно просто: см. упражнение 5.14), в то время как координаты P в обоих реперах даны в условии.

Итак, для преобразования координат из мировой системы координат $\mathcal{W}$ в экранную $\mathcal{S}$ можно найти репер $\mathcal{W}'$, в котором фигура с исходными координатами имеет требуемое положение и размер, а затем преобразовать $\mathcal{S}$ в $\mathcal{W}'$ последовательностью простых аффинных отображений, таких как параллельный перенос и масштабирование. Матрицы этих отображений нужно записать слева направо.

Пример 5.2. В мировых координатах задан прямоугольник, который требуется изобразить в прямоугольной области окна. Центр изображения нужно поместить в центр области, размер изображения сделать максимальным так, чтобы все изображение помещалось в области, и при этом отношение ширины прямоугольника к его высоте должно сохраняться. Пусть размер прямоугольника есть $w_r \times h_r$ в мировых координатах и его центр имеет мировые координаты (x_r, y_r). Аналогично, пусть размер области окна есть $w_s \times h_s$ пикселей, а ее центр P имеет экранные координаты (x_s, y_s) (рис. 5.3).

Пусть экранные координаты задаются репером $\mathcal{S} = (O, e_1, e_2)$. Вычислим отношение ширины к высоте у прямоугольника и области окна: $\rho_r = w_r/h_r$, $\rho_s = w_s/h_s$. Если $\rho_r > \rho_s$, то прямоугольник является более вытянутым по горизонтали, чем окно, поэтому коэффициент масштабирования будет определяться горизонтальными размерами. Рассмотрим такой репер $\mathcal{W}' = (O', e'_1, e'_2)$, что прямоугольник с исходными координатами имеет в нем ту же ширину, что и область окна, то есть $w_r|e'_1| = w_s|e_1|$, и при этом центр прямоугольника совпадает с центром P области, то есть $[P]\mathcal{W}' = (x_r, y_r)$. Поскольку ширина прямоугольника в $\mathcal{W}'$ совпадает с шириной области, а $\rho_r > \rho_s$, высота прямоугольника будет меньше высоты области, то есть изображение будет полностью помещаться в области окна. Как и раньше, рассмотрим вспомогательные реперы $\mathcal{F}_1 = (P, e_1, e_2)$ и $\mathcal{F}_2 = (P, e'_1, e'_2)$. Тогда $D^{\mathcal{S}}_{\mathcal{F}_1} = T(x_s, y_s)$ и $D^{\mathcal{F}_2}_{\mathcal{W}'} = T(-x_r, -y_r)$. Чтобы отобразить $\mathcal{F}_1$ в $\mathcal{F}_2$, нужно увеличить базисные векторы в $\dfrac{|e'_1|}{|e_1|} = \dfrac{w_s}{w_r}$ раз и сделать симметрию относительно горизонтальной оси, поэтому $D^{\mathcal{F}_1}_{\mathcal{F}_2} = S(\lambda, -\lambda)$, где $\lambda = w_s/w_r$. В итоге

$$D^{\mathcal{S}}_{\mathcal{W}'} = T(x_s, y_s)S(\lambda, -\lambda)T(-x_r, -y_r). \tag{9}$$

Если $\rho_r \le \rho_s$, то коэффициент масштабирования λ будет определяться отношением вертикальных размеров, то есть $\lambda = h_s/h_r$, а в остальном матрица перехода будет той же.

Упражнения

Если не указано противное, реперы в следующих задачах считаются ортонормированными. Обозначения матриц аффинных отображений см. в (5), (6) и (7).

5.1. Пусть E — единичная матрица, A и B — матрицы 2×2, а x и y — векторы-столбцы высотой 2. Проверьте, что

$$\begin{pmatrix} A & x \\ 0 & 1 \end{pmatrix} \begin{pmatrix} B & y \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} AB & Ay + x \\ 0 & 1 \end{pmatrix}.$$

В частности

$$\begin{pmatrix} E & x \\ 0 & 1 \end{pmatrix} \begin{pmatrix} A & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} A & x \\ 0 & 1 \end{pmatrix} \quad \text{и} \quad \begin{pmatrix} A & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} E & x \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} A & Ax \\ 0 & 1 \end{pmatrix}.$$

Таким образом, матрицы переноса и следующего за ним линейного отображения объединяются в одну матрицу, а если линейное отображение следует за переносом, то матрица отображения применяется к вектору переноса.

Выведите отсюда, что $S(\lambda, \mu)T(a, b) = T(\lambda a, \mu b)S(\lambda, \mu)$.

5.2. Докажите, что если A — обратимая матрица, то

$$\begin{pmatrix} A & x \\ 0 & 1 \end{pmatrix}^{-1} = \begin{pmatrix} A^{-1} & -A^{-1}x \\ 0 & 1 \end{pmatrix}.$$

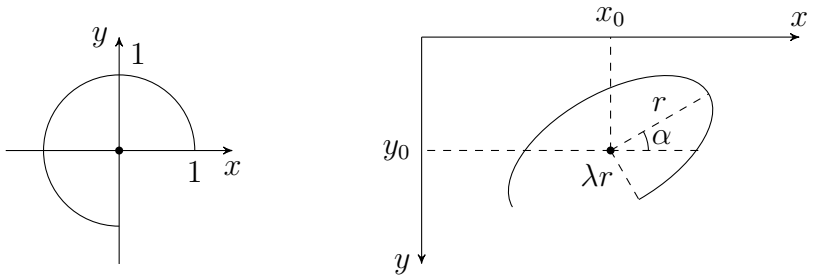


Рис. 5.4. К упражнению 5.6

5.3. Отображение с матрицей $\begin{pmatrix} 1 & a \\ b & 1 \end{pmatrix}$ называется сдвигом (shear). Выясните геометрический смысл коэффициентов a и b. Каков образ квадрата со сторонами, параллельными осям координат и с противоположными углами в точках $(0,0)$ и $(1,1)$ при данном отображении?

5.4. Дан график функции $\sin x$ на отрезке $[-\pi/2, \pi/2]$. Напишите матрицы аффинных отображений, переводящих этот график в графики следующих функций:
а) $\cos x$ на $[0, \pi]$,
б) $\arcsin x$ на $[-1, 1]$,
в) $\arccos x$ на $[-1, 1]$.

5.5. Требуется найти матрицу отображения, которое осуществляет поворот на угол α, за которым следует масштабирование с коэффициентом λ вдоль повернутой оси Ox и коэффициентом μ вдоль повернутой оси Oy. В каком порядке нужно перемножать матрицы $R(\alpha)$ и $S(\lambda, \mu)$?

5.6. В мировых координатах задана дуга окружности с центром в начале координат, радиусом 1 и угловой величиной 270°, как показано на рис. 5.4. Напишите матрицу, переводящую мировые координат точек окружности в экранные координаты точек дуги следующего эллипса. Центр эллипса имеет экранные координаты (x_0, y_0), большая полуось составляет r пикселей, отношение малой полуоси к большой равно λ, и эллипс повернут на угол α против часовой стрелки.

5.7. Студент решает упражнение 5.6 и знает, что искомая матрица представляется в виде произведения $T(x_0, y_0)$, $R(\pm\alpha)$, $S(\pm r, \pm r)$ и $S(1, \lambda)$, однако не уверен, в каком порядке следует перемножать эти матрицы. Если точки с полученными после умножения на матрицу координатами изобразить на экране, где примерно будет находиться полученное изображение и как оно будет выглядеть в каждом из следующих вариантов? (Изображение может находиться за пределами экрана.)
1) $T(x_0, y_0)S(r, -r)S(1, \lambda)R(\alpha)$;
2) $T(x_0, y_0)S(1, \lambda)R(\alpha)S(r, -r)$;
3) $T(x_0, y_0)R(\alpha)S(r, -r)S(1, \lambda)$;
4) $S(r, -r)R(\alpha)S(1, \lambda)T(x_0, y_0)$.

5.8. Пусть l_1 и l_2 будут прямые, получающиеся из оси Ox поворотом на угол α вокруг начала координат и переносом на вектор (a, b), соответственно, и пусть S_1 и S_2 будут матрицы симметрий относительно l_1 и l_2. Используя теорему 8(5), докажите следующие равенства.

1) $S_1 = R(\alpha)S_xR(-\alpha) = R(2\alpha)S_x$;
2) $S_1S_x = R(2\alpha)$;
3) $S_2 = T(a,b)S_xT(-a,-b) = T(0,2b)S_x$;
4) $S_2S_x = T(0,2b)$.

5.9. Пусть S_l будет матрицей симметрии относительно прямой l, пересекающей ось Oy в точке $(0,b)$ и образующей угол α с осью Ox.
1) Докажите, что $S_l = T(0,b)R(\alpha)S_xR(-\alpha)T(0,-b)$.
2) Найдите такие числа a и b, что $S_l = T(a,b)R(2\alpha)S_x$.

5.10. Найдите числа a и b, для которых
1) $R_{(x,y)}(\alpha) = T(a,b)R(\alpha)$;
2) $S_{(x,y)}(\lambda,\mu) = T(a,b)S(\lambda,\mu)$;
3) $R_{(x,y)}(-\alpha)R(\alpha) = T(a,b)$.

5.11. Найдите матрицы следующих отображений.
1) Масштабирование с коэффициентами λ и μ в направлении векторов $(1,0)$ и $(a,1)$, соответственно;
2) масштабирование с коэффициентами λ и μ в направлении векторов $(1,b)$ и $(0,1)$, соответственно;
3) масштабирование с коэффициентами λ и μ с центром в точке с координатами (x_0,y_0) в направлении векторов с координатами (x_1,y_1) и (x_2,y_2) (результат можно представить в виде произведения матриц);
4) проекция на ось Ox параллельно вектору $(a,1)$;
5) проекция на ось Oy параллельно вектору $(1,b)$.

5.12. В данном упражнении считается, что f, f_1 и f_2 — аффинные отображения, а $\mathcal{F}$, $\mathcal{F}_1$ и $\mathcal{F}_2$ — произвольные реперы.
1) Пусть $f(\mathcal{F}_1) = \mathcal{F}_2$. Докажите, что $[f]\mathcal{F}_1 = [f]\mathcal{F}_2$.
2) Пусть $f(\mathcal{F}_1) = \mathcal{F}_2$, $D_1 = D^{\mathcal{F}}_{\mathcal{F}_1}$ и $D_2 = D^{\mathcal{F}}_{\mathcal{F}_2}$. Докажите, что $[f]\mathcal{F} = D_2D_1^{-1}$, а $[f]\mathcal{F}_1 = [f]\mathcal{F}_2 = D_1^{-1}D_2$.
3) Пусть $f_1(\mathcal{F}_1) = \mathcal{F}_2$. Докажите, что $[f_2 \circ f_1]\mathcal{F}_1 = [f_1 \circ f_2]\mathcal{F}_2$.

5.13. Вычислите явный вид матрицы $D^{\mathcal{S}}_{\mathcal{W}'}$ из (8) в примере 5.1 и выпишите явные формулы преобразования мировых координат (x',y') в экранные координаты (x,y).

5.14. Пользуясь упражнением 5.1, покажите, что матрицу $D^{\mathcal{S}}_{\mathcal{W}'}$ из упражнения 5.13 можно представить в виде

$$T(x_s - rx_w/m, y_s + ry_w/m)S(r/m, -r/m).$$

Покажите отсюда, что $[O']\mathcal{S} = (x_s - rx_w/m, y_s + ry_w/m)$.

5.15. Когда аффинное отображение действует в одном аффинном пространстве, его матрица переводит координаты точки в координаты образа этой точки в том же репере (4). В более общем случае аффинное отображение f может действовать из одного пространства с репером $\mathcal{F}$ в другое с репером $\mathcal{F}'$. В этом случае матрицей f по отношению к $\mathcal{F}$ и $\mathcal{F}'$ называется $D^{\mathcal{F}'}_{f(\mathcal{F})}$. Будем обозначать ее через $[f](\mathcal{F},\mathcal{F}')$. Рассуждения, аналогичные (4), показывают, что

$$[f(P)]\mathcal{F}' = D^{\mathcal{F}'}_{f(\mathcal{F})}[f(P)](f(\mathcal{F})) = [f](\mathcal{F},\mathcal{F}')[P]\mathcal{F}.$$

1) Покажите, что $[f_2](\mathcal{F}_2,\mathcal{F}_3)[f_1](\mathcal{F}_1,\mathcal{F}_2) = [f_2 \circ f_1](\mathcal{F}_1,\mathcal{F}_3)$.

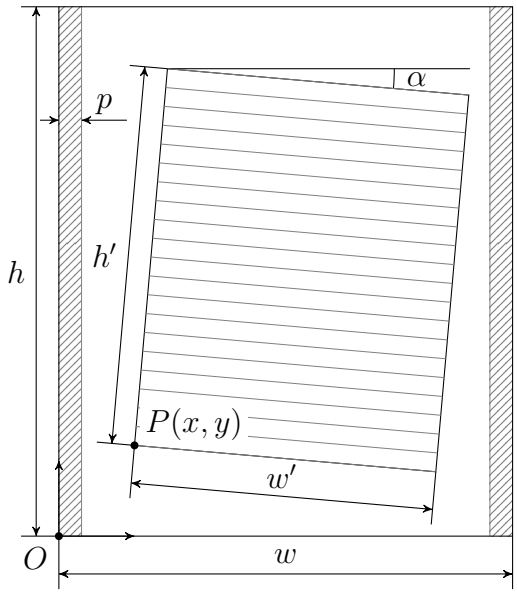


Рис. 5.5. К упражнению 5.17

2) В примере 5.1 матрица, переводящая мировые координаты в экранные, являлась произведением $T(x_s, y_s)S(r/m, -r/m)T(-x_w, -y_w)$, где все три множителя рассматривались как матрицы перехода. Пусть $\mathcal{W}$ и $\mathcal{S}$ — реперы, задающие мировую и экранную системы координат, соответственно. Найдите реперы $\mathcal{F}_1$ и $\mathcal{F}_2$ и также аффинные отображения f_1, f_2 и f_3, такие что $[f_1](\mathcal{W}, \mathcal{F}_1) = T(-x_w, -y_w)$, $[f_2](\mathcal{F}_1, \mathcal{F}_2) = S(r/m, -r/m)$ и $[f_3](\mathcal{F}_2, \mathcal{S}) = T(x_s, y_s)$. Таким образом, матрицу, переводящую мировые координаты в экранные, можно рассматривать не как матрицу перехода, а как матрицу аффинного отображения $f_3 \circ f_2 \circ f_1$.

5.16. Прямоугольная область в экранных координатах задана координатами левого верхнего угла (x_s, y_s), шириной w_s и высотой h_s. Прямоугольник в мировых координатах задан координатами левого нижнего угла (x_r, y_r), шириной w_r и высотой h_r. Разрешение экрана есть r пикселей на сантиметр. Выпишите матрицу преобразования координат в виде произведения матриц переноса и масштабирования, если

а) прямоугольник требуется изобразить в каждом из четырех углов, а также в центре экранной области в масштабе одна мировая единица в одном сантиметре,

б) прямоугольник должен занять всю экранную область, при этом отношение ширины к высоте не обязательно сохраняется.

5.17. На рис. 5.5 изображена ксерокопия прямоугольного изображения размером $w' \times h'$ на странице размером $w \times h$. Изображение повернуто на угол α по часовой стрелке. На странице задана обычная декартова система координат с началом в левом нижнем углу, и координаты левого нижнего угла изображения есть (x, y).

Требуется изменить изображение так, чтобы оно занимало всю ширину страницы, кроме полей шириной p с обеих сторон, было центрировано по высоте и его нижняя граница была горизонтальна. Напишите матрицу этого отображения в системе координат страницы.

5.18. Отношение высоты к ширине листа бумаги любого формата серии A (например, A4) равно $\sqrt{2} \approx 1{,}414$. Это позволяет в точности разместить две уменьшенные и повернутые

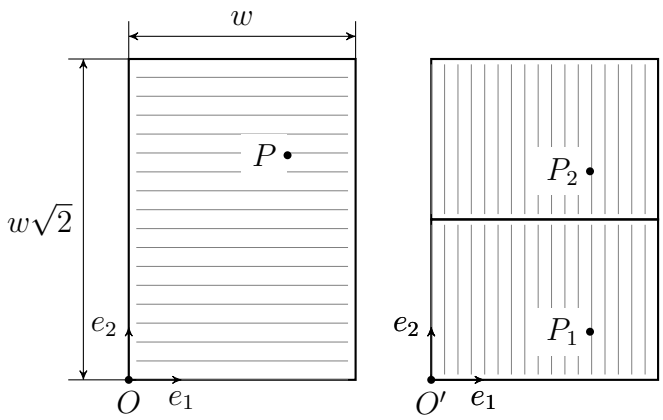


Рис. 5.6. К упражнению 5.18

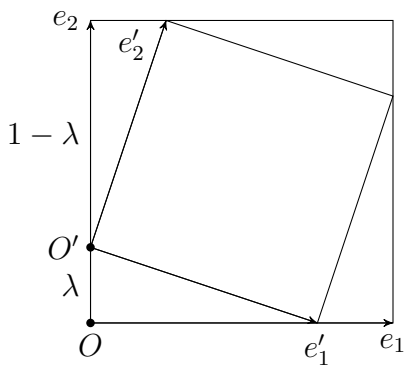


Рис. 5.7. К упражнению 5.20

на 90° копии страницы на одной странице, как показано на рис. 5.6. Пусть при таких преобразованиях точка P переходит в P_1 и P_2, и пусть заданы реперы $\mathcal{F} = (O, e_1, e_2)$ и $\mathcal{F}' = (O', e_1, e_2)$ с началами в левых нижних углах страниц. Напишите матрицу, преобразующую $[P]\mathcal{F}$ в $[P_i]\mathcal{F}'$, $i = 1, 2$, если ширина исходной страницы есть $w|e_1|$.

5.19. Требуется напечатать на принтере содержимого экранного окна программы на Java. Разрешение экрана и принтера — r и r' пикселей на сантиметр, соответственно. Начала системы координат окна находится в левом верхнем углу окна, а принтера — в левом нижнем углу страницы. Левый верхний угол окна должен иметь координаты (x, y) в системе координат принтера. Напишите матрицу, переводящую координаты точки в окне в координаты ее образа на странице.

5.20. На рис. 5.7 изображен квадрат, вписанный в другой квадрат со стороной 1. Известно, что $OO' = \lambda$. Рассмотрим реперы $\mathcal{F} = (O, e_1, e_2)$ и $\mathcal{F}' = (O', e_1', e_2')$ и аффинное отображение f, переводящее больший квадрат в меньший, причем $f(O) = O'$. Найдите $[f]\mathcal{F} = D^{\mathcal{F}}_{\mathcal{F}'}$ тремя способами.

1) Представьте f в виде $f_3 \circ f_2 \circ f_1$ и найдите $([f_3]\mathcal{F})([f_2]\mathcal{F})([f_1]\mathcal{F})$.
2) Выберите подходящие реперы $\mathcal{F}_1$ и $\mathcal{F}_2$ и найдите $D^{\mathcal{F}}_{\mathcal{F}_1} D^{\mathcal{F}_1}_{\mathcal{F}_2} D^{\mathcal{F}_2}_{\mathcal{F}'}$.
3) Выберите три неколлинеарные точки и запишите по столбцам их однородные координаты в $\mathcal{F}$, а также однородные координаты в $\mathcal{F}$ их образов при отображении f. Обозначим получившиеся 3×3 матрицы через X и Y. Тогда $[f]\mathcal{F}$ должна удовлетворять уравнению $([f]\mathcal{F})X = Y$.

6. Аффинные отображения в Java

Аффинные отображения в Java представляются объектами класса `java.awt.geom.AffineTransform`. Можно создавать отображения разных типов, получать их композицию, применять их к точкам и фигурам. Кроме того, в каждом объекте подкласса `Graphics2D` содержится аффинное отображение, применяемое к данному графическому контексту. Поэтому существует два способа рисовать объекты в мировой системе координат. Можно создать объект `AffineTransform`, описывающий преобразование координат из мировой системы в экранную, применить его к фигуре и нарисовать результат. Альтернативно можно задать нужное отображение в графическом контексте и затем рисовать фигуры, заданные непосредственно в мировых координатах. Однако с последним методом связана небольшая трудность. Дело в том, что при рисовании линии Java создает фигуру, описывающую контур линии, и затем закрашивает эту фигуру нужным цветом. Отображение, являющееся частью графического контекста, применяется в том числе к этой фигуре. В результате масштабирование, которое может присутствовать в отображении, влияет в том числе на толщину линий. Для компенсации этого эффекта нужно с помощью метода `setStroke` установить ширину линии, являющуюся обратной величиной к коэффициенту масштабирования.

На рис. 6.1 приведены конструкторы класса `AffineTransform`, а также так называемые статические фабричные методы для создания определенного класса отображений. В описании методов используются обозначения из (5) и (7).

Обратите внимание, что элементы матрицы передаются в конструктор по столбцам. Вероятно, это сделано, чтобы к линейному отображению было легко добавить вектор переноса, координаты которого находятся в последнем столбце.

Если необходимо отображение, осуществляющее поворот на угол, кратный 90°, рекомендуется пользоваться методом `getQuadrantRotateInstance()`. Это ускоряет вычисление и увеличивает точность, так как координаты образов можно получить без тригонометрических функций. Перегруженный метод `getRotateInstance(double x, double y)` возвращает поворот на угол $-\pi \le \theta \le \pi$, переводящий вектор $(1, 0)$ в вектор, коллинеарный (x, y). Этот угол также возвращается функцией `Math.atan2(y, x)`.

Метод `transform(p1, p2)` применяет отображение к точке `p1` и записывает результат в `p2`, а также возвращает его. Если `p2` есть `null`, то создается новый объект того же типа, что и `p1`, то есть `Point2D.Double` или `Point2D.Float`; в противном случае поля `p2` переписываются. Метод работает корректно, даже если `p1` и `p2` — это (указатель на) один и тот же объект; в этом случае его поля переписываются. У данного метода существуют перегруженные версии, принимающие массивы координат точек типа `double` и `float`.

Применить отображение к фигуре, содержащейся в объекте класса, реализующего интерфейс `Shape`, можно с помощью метода `createTransformedShape` или конструктора класса `Path2D.Double`. Здесь используется факт, что такие фигуры, в том числе дуги эллипсов, внутренне представляются (возможно, с некоторой погрешностью) как последовательности кривых Безье первого, второго и третьего порядков. Для того же, чтобы применить аффинное отображение к кривой Безье, достаточно применить его только к опорным точкам; при этом кривые переводятся в кривые, определяемые образами опорных точек.

На рис. 6.2 приведены методы класса `AffineTransform`, меняющие отображение, а также аналогичные методы `Graphics2D`, действующие на отображение, содержащееся в графическом контексте. Таким образом, есть два способа получить объект `AffineTransform`, содержащий поворот, масштабирование или перенос: воспользоваться фабричным мето-

Конструкторы	
`new AffineTransform()`	единичная матрица
`new AffineTransform(double[] a)`	$\begin{pmatrix} a[0] & a[2] & a[4] \\ a[1] & a[3] & a[5] \\ 0 & 0 & 1 \end{pmatrix}$, если `a.length` ≥ 6 $\begin{pmatrix} a[0] & a[2] & 0 \\ a[1] & a[3] & 0 \\ 0 & 0 & 1 \end{pmatrix}$, если $4 \leq$ `a.length` < 6
`new AffineTransform(double m00, double m10, double m01, double m11, double m02, double m12)`	$\begin{pmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ 0 & 0 & 1 \end{pmatrix}$
Фабричные методы (возвращают `AffineTransform`)	
`getRotateInstance(double theta)`	$R(\theta)$
`getRotateInstance(double theta, double x, double y)`	$R_{(x,y)}(\theta)$
`getRotateInstance(double x, double y)`	$R(\mathtt{atan2}\,(y,x))$
`getQuadrantRotateInstance(int n)`	$R(n\pi/2)$
`getScaleInstance(double x, double y)`	$S(x,y)$
`getTranslateInstance(double x, double y)`	$T(x,y)$
Другие методы	
`AffineTransform createInverse()`	возвращает обратное преобразование
`Point2D transform(Point2D p1, Point2D p2)`	применяет отображение к p_1, возвращает результат и записывает его в p_2
`Shape createTransformedShape(Shape s)`	применяет отображение к фигура s
`new Path2D.Double(Shape s, AffineTransform A)`	применяет отображение A к фигура s и записывает результат в новую фигуру

Рис. 6.1. Некоторые методы класса `AffineTransform`. Обозначения см. в (5) и (7).

дом на рис. 6.1 или создать тождественное отображение с помощью конструктора, а затем применить к нему нужный метод на рис. 6.2.

На рис. 6.3 показан пример подкласса `JPanel`, рисующего квадрат со стороной 100 пикселей и левым нижним углом в центре панели.

В результате вызова большинства методов на рис. 6.2 новая матрица умножается на текущую матрицу справа; таким образом, когда отображение будет применяться к точкам, новая матрица будет умножаться первой. Исключением из этого правила является метод `AffineTransform.preConcatenate()`, который умножает свой аргумент на матрицу текущего отображения слева.

Пусть $\mathcal{S}$ — репер, задающий экранную систему координат и пусть с графическим контекстом `g2` ассоциировано аффинное отображение с матрицей A. Эту матрицу можно рассматривать как матрицу перехода $D^{\mathcal{S}}_{\mathcal{W}}$ от $\mathcal{S}$ к некоторому реперу $\mathcal{W}$, задающему мировую систему координат. Тогда `g2.transform(A1)` ассоциирует с `g2` новую мировую систему координат с репером $\mathcal{W}'$, где $D^{\mathcal{W}}_{\mathcal{W}'} = A_1$.

Таким образом, можно считать, что в графическом контексте содержится произведение преобразований, отображающих мировые координаты в экранные. Это произведение

Методы `AffineTransform`	Методы `Graphics2D`	Действие
`A.concatenate(A1)`	`g2.transform(A1)`	$A := AA_1$
`A.preConcatenate(A1)`		$A := A_1A$
`A.rotate(theta)`	`g2.rotate(theta)`	$A := AR(\theta)$
`A.rotate(theta, x, y)`	`g2.rotate(theta, x, y)`	$A := AR_{(x,y)}(\theta)$
`A.quadrantRotate(n)`		$A := AR(n\pi/2)$
`A.scale(x,y)`	`g2.scale(x,y)`	$A := AS(x, y)$
`A.translate(x,y)`	`g2.translate(x,y)`	$A := AT(x, y)$
	`g2.getTransform()`	возвращает A
	`g2.setTransform(A1)`	$A := A_1$

Рис. 6.2. Методы `AffineTransform` и `Graphics2D` для изменение отображения. Переменная `A` имеет тип `AffineTransform`, `g2` имеет тип `Graphics2D` и содержит отображение A.

```
public class DrawingPanel extends JPanel {

  @Override
  public void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2 = (Graphics2D) g;
    g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                        RenderingHints.VALUE_ANTIALIAS_ON);
    g2.translate(getWidth() / 2, getHeight() / 2);
    float r = 50;
    g2.scale(r, -r);
    g2.setStroke(new BasicStroke(1 / r));
    g2.draw(new Rectangle2D.Double(0, 0, 2, 2));
  }
}
```

Рис. 6.3. Применение аффинных отображений к графическому контексту

можно рассматривать как стек (на самом деле, конечно, хранится лишь результат произведения), к которому новые преобразования можно добавлять только справа. Эти преобразования будут первыми применяться к мировым координатам. В классе `Graphics2D` нет методов, которые добавляют преобразования слева, и делать этого не следует. Вместо этого Java может самостоятельно добавить преобразование слева для правильного отображения компонента на требуемом устройстве, например, на принтере.

По этой же причине нельзя полагаться на то, что контекст, переданный в аргументе метода `paintComponent()`, содержит вначале тождественное отображение. Несмотря на то, что класс `Graphics2D` содержит метод `setTransform()` для установки аффинного отображения, его нельзя использовать вместо `transform()` и аналогичных методов на рис. 6.2, потому что `setTransform()` может переписать переданное отображение. Как правило, устанавливать можно только отображение, которое было ранее получено с помощью метода `getTransform()`. Так, в разделе 7 описывается метод иерархического моделирования, который использует следующий порядок рисования.

```
// Получить текущее отображение
AffineTransform saveAT = g2.getTransform();
// Добавить новое отображение
g2.transform(...);
// Нарисовать объект
g2.draw(...);
// Восстановить исходное отображение
g2.setTransform(saveAT);
```

Иногда желательно установить мировую систему координат, где единицами измерения служат сантиметры, дюймы или их части. Для этого есть два способа. Следующий фрагмент устанавливает в графическом контексте `g2` аффинное преобразование, соответствующее системе координат с началом в левом верхнем углу, где ось Ox направлена вправо, ось Oy направлена вниз и единичный отрезок составляет $1/72$ дюйма.

```
GraphicsConfiguration gc = g2.getDeviceConfiguration();
g2.setTransform(gc.getDefaultTransform());
g2.transform(gc.getNormalizingTransform());
```

Здесь метод `getDeviceConfiguration()` возвращает структуру данных, описывающую характеристики текущего графического устройства. Метод `getDefaultTransform()` возвращает преобразование A, установленное в устройстве по умолчанию (обычно оно тождественное). Наконец, метод `getNormalizingTransform()` возвращает преобразование, композиция которого с A дает описанную систему координат. Единица измерения, равная $1/72$ дюйма, называется типографским пунктом и используется для измерения шрифтов во многих программах компьютерной верстки и, в частности, в языке PostScript описания страниц, разработанном компанией Adobe.

Второй способ заключается в вызове метода

```
Toolkit.getDefaultToolkit().getScreenResolution();
```

который возвращает разрешение экрана как целое количество пикселей на дюйм. Однако оба эти способа зависят от того, насколько правильно драйверы и операционная система сообщают Java разрешение экрана, и ни один из них не гарантирует правильного масштабирования на всех устройствах.

Упражнения

6.1. Напишите программу, которая использует оба описанных выше метода для установки мировой системы координат с единичным отрезком, равным 1 см. Нарисуйте горизонтальную линию длиной 25 мировых единиц и измерьте ее на экране физической линейкой. Является ли длина линии приблизительно 25 см? Какой из двух методов дает лучшее приближение?

6.2. Напишите программу, которая создает объект `Path2D.Double`, содержащий приближение к графику $\sin x$ на отрезке $[-\pi/2, \pi/2]$ в виде ломаной с большим числом (от 50 до 100) отрезков. График должен использовать естественные координаты. Затем, применив требуемое отображение к графическому контексту, нарисуйте этот график в центре окна в масштабе π единиц на дюйм. Также нарисуйте графики других функций, указанных в упражнении 5.4.

```
class House extends Path2D.Double {

  public void lineToRelative(double x, double y) {
    Point2D p = getCurrentPoint();
    lineTo(p.getX() + x, p.getY() + y);
  }

  public House() {
    moveTo(3, 1); lineTo(19, 1); // Земля
    append(new Rectangle2D.Double(4, 1, 6, 6), false); // Дом
    moveTo(3, 6); lineTo(7, 10); lineTo(11, 6); // Крыша
    append(new Rectangle2D.Double(5, 3, 2, 2), false); // Окно
    // Крона дерева
    double alpha = Math.toDegrees(Math.asin(1.0/6));
    append(new Arc2D_Double(16.5, 3, 3, 4, -90+alpha, 270-alpha),
           false);
    // Ствол дерева
    moveTo(15.5, 1); lineToRelative(0, 2);
    moveTo(16.5, 1); lineToRelative(0, 2);
    // Труба
    moveTo(8, 9); lineToRelative(0, 1);
    lineToRelative(1, 0); lineToRelative(0, -2);
  }
}
```

Рис. 6.4. К упражнению 6.4. Класс, определяющий фигуру на рис. 5.1. Программа использует класс `Arc2D_Double`, описанный в упражнении 1.4

6.3. Напишите программу, рисующую дугу эллипса из упражнения 5.6. Сначала программа создает объект

```
Shape arc = new Arc2D.Double(-1, -1, 2, 2, 90, 270, Arc2D.OPEN);
```

Параметры конструктора `Arc2D.Double` описаны в упражнении 1.4. Вспомните, что углы отсчитываются против часовой стрелки, если считать, что ось Oy направлена вниз, поэтому можно считать, что углы отсчитываются по часовой стрелке в обычной математической системе координат, где ось Oy направлена вверх. Затем требуемое отображение применяется к `arc` и полученная фигура рисуется. Поскольку используется масштабирование с разными коэффициентами, отображение лучше не устанавливать в графическом контексте, чтобы ширина линии не зависела от направления. Параметры x_0, y_0, r, λ и α задайте константами.

6.4. Класс, задающий фигуру на рис. 5.1, показан на рис. 6.4. Используя его, напишите программу, описанную в примере 5.2 и упражнении 5.16. В качестве прямоугольной области возьмите все окно за исключением границы в p пикселей. Программа должна содержать по одному методу, возвращающему аффинное отображение, для каждого варианта расположения рисунка в области. Метод `paintComponent()` должен применять к рисунку преобразование, полученное от одного из этих методов, и рисовать результат. Убедитесь, что фигура правильно перерисовывается при изменении размера окна.

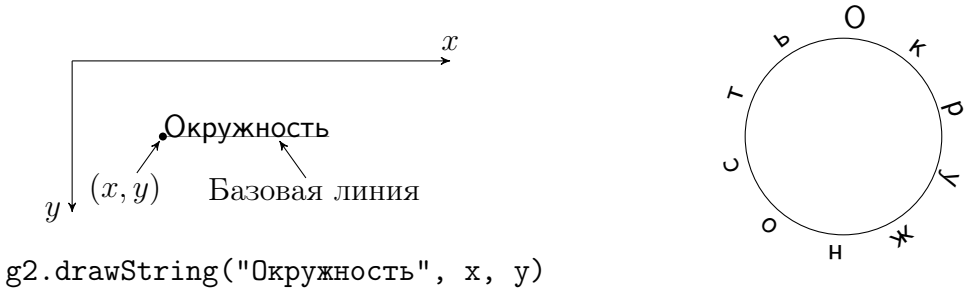


Рис. 6.5. К упражнению 6.5

6.5. Одной из характеристик графического контекста является текущий шрифт. Его можно установить, например, следующей командой.

```
g2.setFont(new Font(Font.SANS_SERIF, Font.PLAIN, 25));
```

Первым аргументом конструктора класса `java.awt.Font` является название шрифта. В этом классе имеется несколько предопределенных названий, включая `SERIF` (обычный, с засечками), `SANS_SERIF` (рубленный, без засечек) и `MONOSPACED` (моноширинный). Вторым аргументом является стиль, который включает варианты `BOLD` (жирный), `ITALIC` (курсив) и `PLAIN` (прямой). Третий аргумент задает размер в мировых единицах. Таким образом, после вызова `g2.getNormalizingTransform()` можно задавать размер шрифта в пунктах.

Метод `void drawString(String str, float x, float y)` в классе `Graphics2D` печатает строку `str` в точке с координатами (x, y). Точка находится непосредственно слева от строки на ее *базовой линии*, как показано на рис. 6.5. Аффинное отображение в графическом контексте действует на печатаемые символы. Так, направление текста совпадает с направлением оси Ox мировой системы координат, а буквы направлены в ту сторону от базовой линии, которая противоположна оси Oy. Таким образом, при аффинном отображении по умолчанию буквы расположены обычным образом (а не перевернуты).

Напишите программу, рисующую окружность радиуса r и буквы слова «Окружность», равномерно распределенные вокруг окружности. Вертикальная ось букв должна быть направлена к центру окружности, как показано на рис. 6.5. Расстояние от центра до базовой линии букв равно $1{,}1r$. Используйте метод `drawString`, где второй и третий аргументы равны нулю.

6.6. Пользуясь аффинным отображением из упражнения 5.20, напишите программу, рисующую картину на рис. 6.6.

6.7. В [1] описан следующий способ рисования фрактала, похожего на папоротник. Четыре аффинных отображения задаются своими матрицами.

$$\begin{pmatrix} 0{,}00 & 0{,}00 & 0{,}00 \\ 0{,}00 & 0{,}16 & 0{,}00 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0{,}85 & 0{,}04 & 0{,}00 \\ -0{,}04 & 0{,}85 & 1{,}60 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 0{,}20 & -0{,}26 & 0{,}00 \\ 0{,}23 & 0{,}22 & 1{,}60 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} -0{,}15 & 0{,}28 & 0{,}00 \\ 0{,}26 & 0{,}24 & 0{,}44 \\ 0 & 0 & 1 \end{pmatrix}$$

Пусть f_i— аффинное отображение с i-й матрицей, $i = 1, 2, 3, 4$. Последовательность точек задается рекуррентным отношением: P_0 имеет координаты $(0, 0)$, а $P_{n+1} = f(A_n)$, где f есть f_i со вероятностью p_i. Вероятности заданы следующим образом.

i	1	2	3	4
p_i	0,01	0,85	0,07	0,07

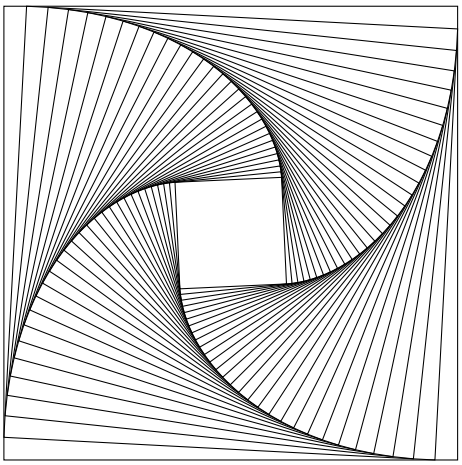


Рис. 6.6. К упражнению 6.6

Рис. 6.7. К упражнению 6.7

При большом n, например, при $n = 50\,000$, точки $P_0, \ldots, P_n$ формируют изображение папоротника.

Известно, что координаты (x, y) каждой точки P_i удовлетворяют неравенствам:

$$-2{,}2 < x < 2{,}7 \qquad 0 \le y < 10.$$

Требуется нарисовать изображение так, чтобы оно было максимального размера, но полностью помещалось в окне и с каждой стороны оставались поля шириной не менее 10 пикселей. Начальный размер окна — 300×300 пикселей, но он может меняться пользователем во время исполнения.

Примените преобразование мировых координат в экранные к графическому контексту и создайте четыре объекта `AffineTransform`, содержащие описанные выше отображения. Для получения случайных чисел используйте метод `Math.random()`, который возвращает числа от 0 до 1. Рисуйте точки в виде закрашенных квадратов со стороной 1 пиксель. Обратите внимание, что для определения размера такого квадрата в мировых координатах следует учесть масштабирование, примененное к графическому контексту.

7. Программирование анимации

7.1. Использование таймера

В библиотеке Java есть два таймера: `javax.swing.Timer` и `java.util.Timer`. Второй из них появился позже и имеет бóльшую функциональность. Однако для приложений Swing первого таймера обычно достаточно, поэтому мы будем использовать его.

Как описано в §1.10, конструктор `Timer` принимает два аргумента: время ожидания перед первым срабатыванием и между последующими срабатываниями в миллисекундах, а также объект класса, реализующего интерфейс `java.awt.event.ActionListener`. Этот интерфейс объявляет единственный метод `void actionPerformed(ActionEvent e)`. Данный метод выступает в роли обработчика событий, то есть вызывается при каждом срабатывании таймера. Напомним, что методы являются открытыми (public) по умолчанию в определении интерфейса, но модификатор доступа необходимо указывать при его реализации (желательно также указывать аннотацию `@Override`).

Класс, реализующий интерфейс `ActionListener`, обычно используется только в одном месте, поэтому его можно сделать внутренним, локальным или анонимным. Последний вариант показан на рис. 7.2. Однако для описания обработчика событий не обязательно создавать новый класс. Например, можно сделать так, чтобы сам класс `DrawingPanel` реализовывал интерфейс `ActionListener`. Этот вариант показан на рис. 7.3.

Таймер может иметь несколько обработчиков событий. Они добавляются с помощью метода `addActionListener()`. Конструктор устанавливает одинаковую задержку перед первым срабатыванием и между последующими срабатываниями, но эти интервалы можно менять независимо с помощью методов `setInitialDelay()` и `setDelay()`, соответственно.

Предположим, таймер нужно остановить после десяти срабатываний. Это можно сделать в переопределенном методе `actionPerformed()` с помощью метода `stop()`. Однако класс-обработчик событий должен быть определен до того, как его объект передается в конструктор класса `Timer`. Как в таком случае метод `actionPerformed()` может использовать объект класса `Timer`, который еще не создан? Эту трудность можно решить несколькими способами. Во-первых, таймер можно сделать полем класса `DrawingPanel`.

`Timer(int delay, ActionListener listener)`	Конструктор
`void addActionListener(ActionListener listener)`	Добавляет метод, вызываемый при срабатывании
`boolean isRunning()`	Возвращает `true`, если таймер работает
`void setDelay(int delay)`	Устанавливает интервал (в миллисекундах) между последовательными срабатываниями
`void setInitialDelay(int initialDelay)`	Устанавливает интервал (в миллисекундах) перед первым срабатыванием
`void setRepeats(boolean flag)`	Если `flag` есть `false`, делает так, чтобы таймер сработал только один раз
`void start()`	Запускает таймер
`void stop()`	Останавливает таймер

Рис. 7.1. Основные методы класса `Timer`

```
public class DrawingPanel extends JPanel {
  public DrawingPanel() {
    ActionListener performer = new ActionListener() {
      @Override
      public void actionPerformed(ActionEvent e) {
        // Действие при срабатывании таймера
      }
    };

    new Timer(1000, performer).start();
  }
  ...
}
```

Рис. 7.2. Анонимный класс как обработчик событий

```
public class DrawingPanel extends JPanel implements
 ActionListener {
  @Override
  public void actionPerformed(ActionEvent e) {
    // Действие при срабатывании таймера
  }

  public DrawingPanel() {
    new Timer(1000, this).start();
  }
  ...
}
```

Рис. 7.3. Панель как обработчик событий

Внутренний или анонимный класс, реализующий `ActionListener`, имеет доступ к полям объемлющего класса, включая закрытые. Во-вторых, конструктору `Timer` можно передать `null` в качестве второго аргумента. После того, как объект `Timer` создан, можно определить обработчик событий и добавить его к таймеру методом `addActionListener`. Этот вариант показан на рис. 7.4. Как указано в §1.10, локальный класс, реализующий `ActionListener`, не может использовать переменные, объявленные в окружающем методе, если они меняют свое значение. В данном случае переменная `t` не меняет значения и соответственно объявлена `final`. Начиная с Java 8, модификатор `final` в такой ситуации необязателен.

Параметр метода `actionPerformed(ActionEvent e)` обычно не используется, однако `e.getSource()` возвращает источник события, которым в данном случае является таймер. Это дает еще один способ получить доступ к таймеру внутри обработчика событий. Заметьте, что метод `getSource()` возвращает результат типа `Object`, а не `Timer`, поэтому нужно использовать приведение типа.

Как описано в §2.2, любой доступ к компонентам Swing должен производиться из потока диспетчеризации событий, а не из главного потока. Таймер Swing приспособлен к

```
public class DrawingPanel extends JPanel {
  public DrawingPanel() {
    final Timer t = new Timer(1000, null);
    ActionListener performer = new ActionListener() {
      private int timerFired = 0;

      @Override
      public void actionPerformed(ActionEvent e) {
        timerFired++;
        if (timerFired >= 10) t.stop();
      }
    };
    t.addActionListener(performer);
    t.start();
  }
  ...
}
```

Рис. 7.4. Остановка таймера

этому, так как обработчики событий исполняются в потоке диспетчеризации. Значит, в методе `actionPerformed()` можно вызывать метод `repaint()` для того, чтобы запланировать перерисовку компонента[6].

Лямбда-выражения. Начиная с Java 8, вместо анонимного класса можно использовать так называемые лямбда-выражения, которые значительно упрощают синтаксис по сравнению с анонимными классами. Так, конструктор `DrawingPanel` можно записать следующим образом.

```
public DrawingPanel() {
  new Timer(1000, e -> { /* Действие при срабатывании таймера */ }).start();
}
```

Компилятору знает, что второй аргумент конструктора `Timer` имеет тип `ActionListener`. Это так называемый функциональный интерфейс, то есть интерфейс с единственным абстрактным методом. Параметр этого метода имеет тип `ActionEvent`, значит, параметр `e` имеет тот же тип.

В общем случае лямбда-выражение имеет следующий вид:

```
(T1 x1, ..., Tn xn) -> /* тело лямбда-выражения */
```

где тело является либо выражением, либо блоком операторов, взятым в фигурные скобки. Эти скобки можно опустить, если тело состоит из единственного метода, возвращающего `void`, например, `System.out.println()`. Также можно опускать типы параметров и скобки вокруг них, если параметр один. Типом лямбда-выражения является функциональный интерфейс, и он должен определяться контекстом. Например, лямбда-выражение может использоваться в качестве правой части присваивания или аргумента метода, поскольку от выражения в таком месте ожидается конкретный тип.

[6]Ранее метод `repaint()` считался потокобезопасным, то есть его можно было вызывать из любого потока. Однако этот метод не обозначен как потокобезопасный в документации Java 8.

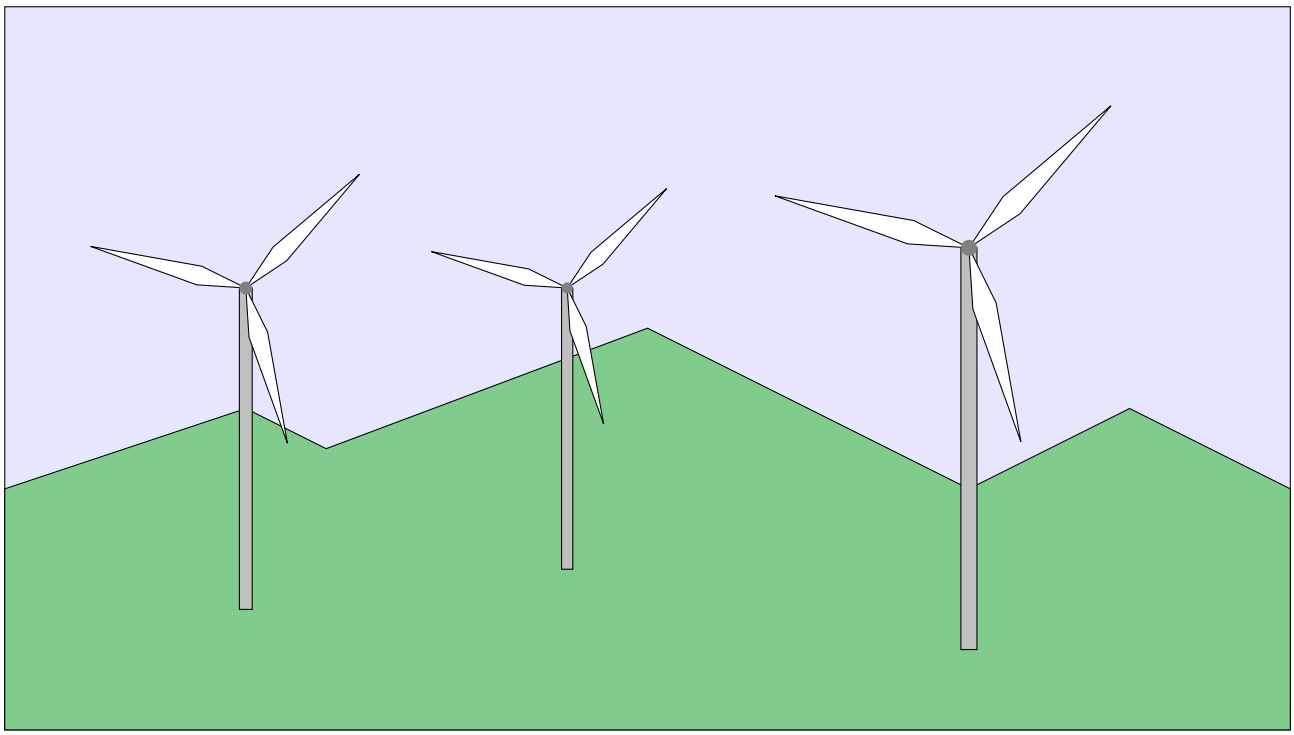


Рис. 7.5. Пейзаж

Если тело лямбда-выражения состоит из вызова единственного метода, то существует еще более компактная запись. Вместо

```
(T1 x1, ..., Tn xn) -> object.instanceMethod(x1, ..., xn)
```

можно написать `object::instanceMethod`, а вместо

```
(T1 x1, ..., Tn xn) -> Class.staticMethod(x1, ..., xn)
```

— `Class::staticMethod`.

7.2. Иерархическое моделирование

Идея иерархического моделирования заключается в том, что изображение состоит из нескольких объектов, каждый из которых рисуется в своей естественной системе координат. Перед рисованием каждого объекта к графическому контексту применяется аффинное преобразование, которое задает требуемое положение и размер объекта.

На рис. 7.5 изображен пейзаж, наблюдаемый из окна движущейся машины. Пейзаж постепенно сдвигается влево и циклически повторяется, а лопасти на ветрогенераторах вращаются. Все ветрогенераторы одинаковы, но к ним применяется масштабирование с разными коэффициентами для имитации перспективы. В данной программе не делается попытка сделать геометрически правильную перспективную проекцию трехмерных объектов, поэтому мировые координаты установок и их масштаб выбраны достаточно произвольно. Ветрогенератор рисуется в системе координат с началом в основании мачты. Перед началом рисования каждой установки к графическому контексту применяется нужное преобразование, а после завершения исходное преобразование восстанавливается. Аналогично лопасти рисуются в системе координат, начало которой расположено на вершине мачты, поэтому начало системы координат графического контекста перемещается в эту точку.

Константа	Пример	Описание	Единицы
w	16	ширина изображения	мировые единицы
h	9	высота изображения	мировые единицы
r	50	разрешение	пиксели на мировую единицу
f	30	кадровая частота	кадры в секунду
T	30	период сдвига изображения	секунды
ω	1/3	частота вращения лопастей	обороты в секунду

Рис. 7.6. Некоторые константы

```
AffineTransform saveAT = g2.getTransform();
g2.translate(x, y);
g2.scale(s, s);
g2.setColor(...); // Цвет мачты
g2.fill(new Rectangle2D.Double(-wt/2, 0, wt, ht));
g2.translate(0, ht);
g2.setColor(...); // Цвет лопастей
g2.rotate(alpha);
g2.fill(blades);
g2.setTransform(saveAT);
```

Рис. 7.7. Рисование ветрогенератора

Некоторые константы, используемые в программе, и примеры их значений показаны на рис. 7.6. (В настоящей программе этим константам нужно дать содержательные имена.) Размер панели следует установить в wr на hr пикселей, а чтобы не дать пользователю менять размеры окна, нужно вызвать

```
frame.setResizable(false);
```

в методе `createAndShowGUI()` на рис. 2.2 (здесь объект `frame` класса `JFrame` представляет главное окно программы).

Один циклический сдвиг занимает $N = fT$ кадров. Обработчик событий от таймера увеличивает счетчик кадров n и берет его остаток при делении на N, чтобы избежать переполнения. Затем обработчик запрашивает перерисовку с помощью вызова метода `repaint()`.

Метод `paintComponent()` устанавливает преобразование мировых координат в экранные, сдвигает мировую систему координат влево на nw/N единиц и рисует картину. Чтобы сдвиг был циклическим, система координат затем сдвигается на w единиц вправо и та же картина рисуется еще раз.

В программе необходимо задать размеры лопастей и высоту мачты, а также координаты основания и коэффициенты масштабирования для каждой установки. Пусть мачта имеет размеры $w_t \times h_t$, координаты основания (x, y) и коэффициент масштабирования s. Предположим также, что фигура, описывающая лопасти, находится в переменной `blades`, а угол поворота лопастей, вычисленный на основе номера кадра n и параметров ω, f есть α. Тогда код, рисующий установку, показан на рис. 7.7. Для создания фигуры `blades` можно задать одну лопасть в виде дельтоида, применить к нему повороты на 120° и 240°, как описано в разделе 6, и добавить полученные фигуры к исходной с помощью метода `Path2D.append()`.

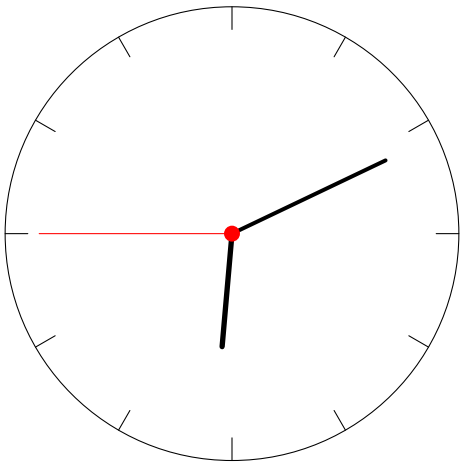


Рис. 7.8. К упражнению 7.5

Упражнения

7.1. Предположим, обработчик событий таймера вносит изменения в поля панели и вызывает метод `repaint()` для перерисовки. Можно ли можно ли сделать обработчик событий статическим вложенным классом панели?

7.2. Напишите программу, которая раз в секунду последовательно печатает сообщения «5», «4», «3», «2», «1», «Поехали!» на стандартный вывод и затем прекращает работу. Обратите внимание, что действия по созданию и запуску таймера нужно производить из потока диспетчеризации событий, поэтому метод `main()` должен быть похож на соответствующий метод из рис. 2.2[7].

7.3. Напишите программу, которая раз в секунду печатает текущее время на стандартный вывод. Часы, минуты и секунды можно получить соответственно с помощью методов экземпляра `getHour()`, `getMinute()` и `getSecond()` класса `java.time.LocalTime`. В свою очередь объект `LocalTime`, соответствующий текущему моменту, возвращается статическим методом `LocalTime.now()`. Время нужно запрашивать при каждом исполнении обработчика событий таймера, потому что срабатывание таймера в каждом интервале не гарантируется. Например, если программа была занята в течении нескольких интервалов, может быть инициировано лишь одно событие. Метод `main()` следует оформить, как описано в упражнении 7.2.

7.4. Напишите программу, рисующую пейзаж, которая описана в данном разделе.

7.5. Напишите программу, рисующую часы, как показано на рис. 7.8. Часы должны двигаться по панели, отражаясь от границ. О том, как определять текущее время, см. упражнение 7.3. Не используйте в тексте программы тригонометрические функции.

7.6. При иерархическом моделировании фрагмент, рисующий часть объекта, обычно вложен во фрагмент, рисующий сам объект. Соответственно, аффинные отображения должны сохраняться и восстанавливаться в правильном порядке. Так, пусть $\mathcal{W}$ — репер, задающий

[7]Таймер Swing не лучшим образом подходит для совершения регулярных действий вне контекста приложений с графическим интерфейсом. Для этого можно использовать `java.util.Timer` или классы из пакета `java.util.concurrent`.

```
// Запоминание мировой системы координат
AffineTransform at1 = g2.getTransform();
// Переход к системе координат, связанной с объектом
g2.transform(D1);
g2.draw(...); // Начало рисования объекта
AffineTransform at2 = g2.getTransform();
// Переход к системе координат, связанной с частью объекта
g2.transform(D2);
g2.draw(...); // Рисование части объекта
// Возврат к системе координат объекта
g2.setTransform(at2);
g2.draw(...); // Продолжение рисования объекта
// Возврат к мировой системе координат
g2.setTransform(at1);
```

Рис. 7.9. Вложенные преобразования координат

```
transformScope(g2, () -> {
  g2.transform(D1);
  g2.draw(...); // Начало рисования объекта
  transformScope(g2, () -> {
    g2.transform(D2);
    g2.draw(...); // Рисование части объекта
  });
  g2.draw(...); // Продолжение рисования объекта
});
```

Рис. 7.10. Вложенные преобразования координат с помощью лямбда-выражений

мировую систему координат, $\mathcal{W}_1$ — репер, связанный с объектом (например, ветрогенератором), а $\mathcal{W}_2$ — репер, связанный с частью объекта (например, лопастями). Эти реперы задают две матрицы перехода: $D_1 = D^{\mathcal{W}}_{\mathcal{W}_1}$ и $D_2 = D^{\mathcal{W}_1}_{\mathcal{W}_2}$. Тогда код, рисующий объект, может иметь вид, показанный на рис. 7.9.

К сожалению, контроль правильной вложенности аффинных отображений целиком лежит на программисте. Так, можно забыть восстановить исходное отображение по окончании рисования части объекта или восстановить отображения не в том порядке. Чтобы возложить контроль вложенности на компилятор, напишите метод `void transformScope( Graphics2D g2, Runnable r)`. (См. §2.2 о функциональном интерфейсе `Runnable`.) Метод `transformScope()` получает графический контекст и лямбда-выражение. Он запоминает текущее аффинное отображение в графическом контексте, исполняет лямбда-выражение и восстанавливает отображение. С помощью такого метода код на рис. 7.9 можно переписать, как показано на рис. 7.10. Перепишите программу из упражнения 7.4 с использованием метода `transformScope()`.

Список литературы

1. *Barnsley M. F.* Fractals everywhere. — 2nd ed. — Morgan Kaufmann, 2000.
2. *Кострикин А. И.* Введение в алгебру. ч. II. Линейная алгебра. — М. : Физматлит, 2000.
3. *Макаров Е. М.* Элементы двумерной графики в Java. Часть I. — Нижегородский государственный университет им. Н.И. Лобачевского, 2015.
4. *Портянкин И. А.* Swing. Эффектные пользовательские интерфейсы. — 2-е изд. — Лори, 2011.
5. *Хорстманн К. С.*, *Корнелл Г.* Java. Библиотека профессионала : в 2 т. : пер. с англ. — 9-е изд. — М. : ООО «И.Д. Вильямс», 2014.
6. *Шилдт Г.* Java 8. Полное руководство : в 2 т. : пер. с англ. — М. : ООО «И.Д. Вильямс», 2015.
7. *Шилдт Г.* Swing. Руководство для начинающих : пер. с англ. — М. : ООО «И.Д. Вильямс», 2007.

Евгений Маратович Макаров

Элементы двумерной графики в Java

Учебно-методическое пособие

Федеральное государственное автономное образовательное
учреждение высшего образования
«Национальный исследовательский Нижегородский
государственный университет им. Н. И. Лобачевского»
603950, Нижний Новгород, пр. Гагарина, 23.