МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

Нижегородский государственный университет им. Н.И. Лобачевского

Элементы двумерной графики в Java. Часть I

Учебно-методическое пособие

Рекомендован методической комиссией института информационных технологий, математики и механики для студентов ННГУ, обучающихся по направлению 02.03.01 «Математика и компьютерные науки»

Нижний Новгород
2015

УДК 004.92(075.8)
ББК 32.97
М15

М15 Макаров Е.М. ЭЛЕМЕНТЫ ДВУМЕРНОЙ ГРАФИКИ В JAVA. ЧАСТЬ I: Учебно-методическое пособие. – Нижний Новгород: Нижегородский госуниверситет, 2015. – 22 с.

Рецензент: д.ф.-м.н., профессор **Н.И. Жукова**

Пособие содержит сведения, необходимые для написания на языке Java программ, использующих двумерную графику (часть курса «Компьютерная геометрия и геометрическое моделирование»). В части I описываются объектно-ориентированное программирование, создание графического пользовательского интерфейса и графика в терминах пикселей. Пособие предназначено для студентов-математиков третьего курса механико-математического факультета.

Ответственный за выпуск:
председатель методической комиссии
института информационных технологий, математики и механики ННГУ
к.ф.-м.н., доцент **О.А. Кузенков**

УДК 004.92(075.8)
ББК 32.97

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 в одинаковым именем. Такое 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 конструк-

тор, 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

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

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

```
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, ... { ... }
```

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

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

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.

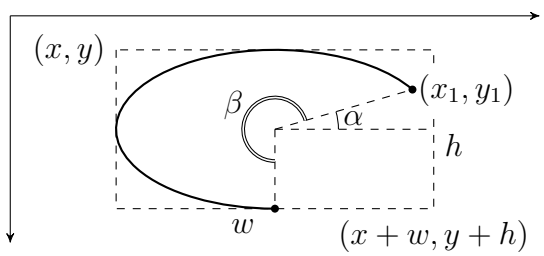

Рис. 1. Дуга эллипса, представляемая классом `Arc2D.Double`

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`.

Упражнение 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.

Упражнение 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;
```

Упражнение 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.

Упражнение 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. 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.)

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 подклассом `C`, или интерфейсом, который реализует `C`. На рис.2 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. 2 можно переписать следующим образом.

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

[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. 2. Пример локального класса

```
}
```

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 программа `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 было сказано 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 аннотации `public`), а также вложенные классы. Тем не менее, рекомендуется помещать каждый класс верхнего уровня в отдельный файл.

Для компиляция файла нужно указать его имя. Например, пусть класс `graphics.shapes.Test` с методом `main` находится в файле `C:\Users\Username\java\graphics\shapes\Test.java`. Скомпилировать файл можно следующей командой.

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

В Linux каталоги разделяются обычной косой чертой `/`. При успешном завершении компиляции генерируется файл `Test.class`, содержащий байт-код. Запустить его можно с помощью программы `java`. Однако важно заметить, что аргументом этой программы является полное имя класса (включая содержащие его пакеты), а не имя файла. Поскольку имена пакетов будут интерпретированы как имена подкаталогов, запускать `java` нужно из родительского каталога внешнего пакета.

```
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 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 Windwows 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
```

Упражнение 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. 3.

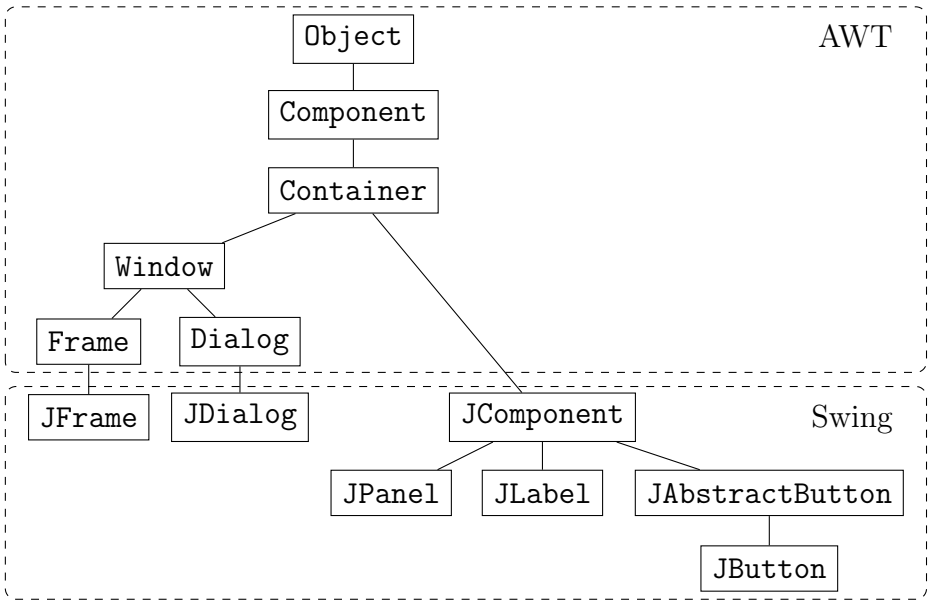

Рис. 3. Некоторые классы, соответствующие элементам интерфейса

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 создания 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. 4. 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*,

```
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. 4. Класс с методом `main()`

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. 4 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 [2].

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. 5.

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. 4, 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 методов `getMinimalSize()`,

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

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

`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. 4 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 метод `getPreferredSize()` возвращать нужный размер. Во-первых, можно вызвать `setPreferredSize(size)`; тогда 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. 4. 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. 5, измеряются в пикселах.

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.

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. 6. Вместе с классом `Drawing` на рис. 4 он образует программу, которая выводит на экран окно заданного размера с белым фоном.

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

Упражнение 2.2. Сделайте `DrawingPanel` единственным классом, поместив в него методы `createAndShowGUI()` и `main()`.

Упражнение 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, `JComponent` содержит метод `paintComponent( Graphics g)`, который прорисовывает компонент. Поскольку метод является защищенным (`protected`), он доступен в подклассах `JComponent` и может быть переопределен в

```
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. 6. Подкласс `JPanel`

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 и который возвращается методом `getClipBounds()`. Метод `paintComponent()` может использовать эту информацию и рисовать только внутри

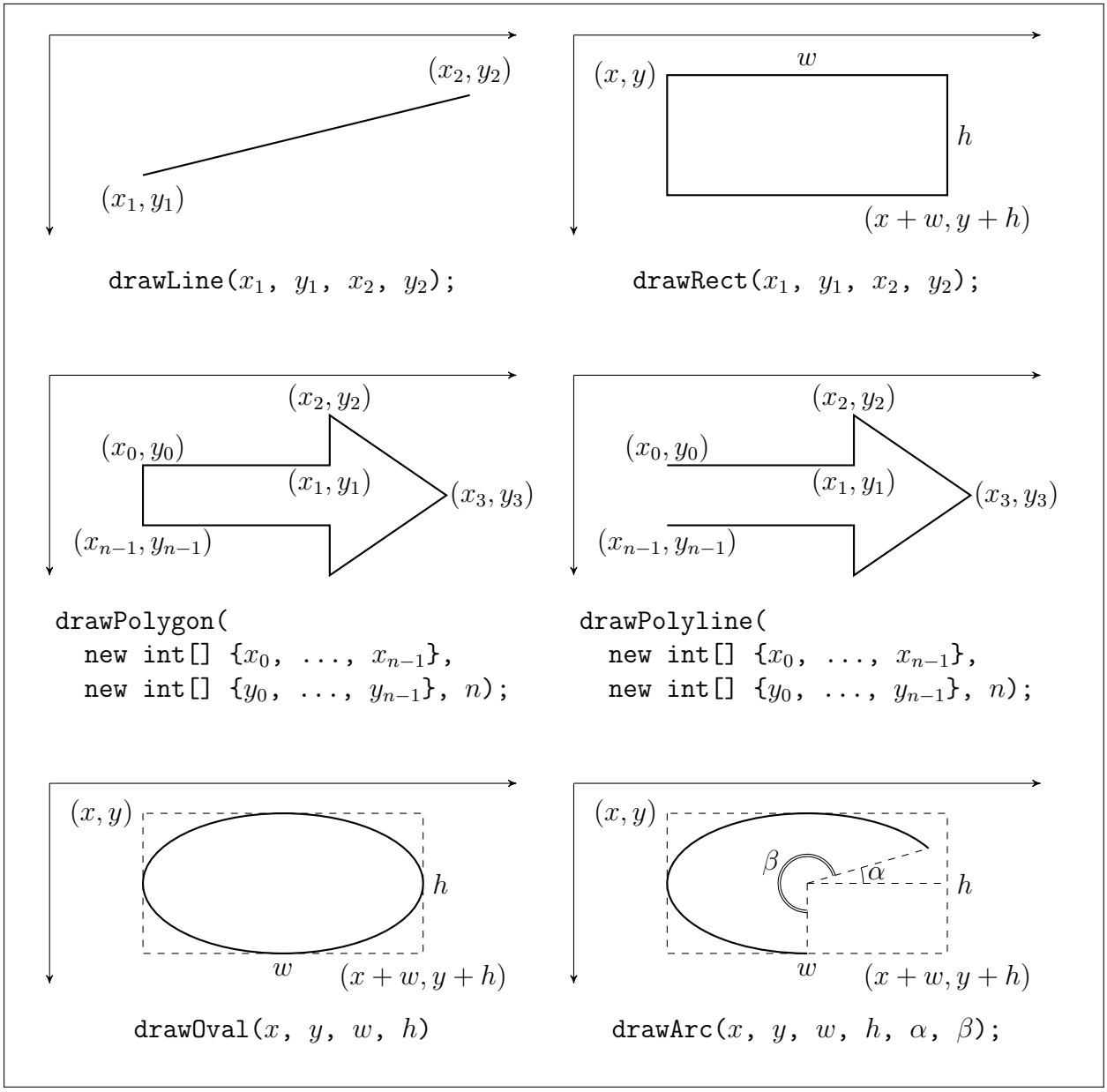

Рис. 7. Некоторые методы для рисования в классе `Graphics`

данного прямоугольника для большей эффективности.

Класс `Graphics` содержит методы для рисования отрезков, прямоугольников, многоугольников, эллипсов и их дуг. Они показаны на рис. 7. 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. 7, в классе `Graphics` есть методы `fillRect()`, `fillPolygon()`, `fillOval()` и `fillArc()`, которые закрашивают соответствующие фи-

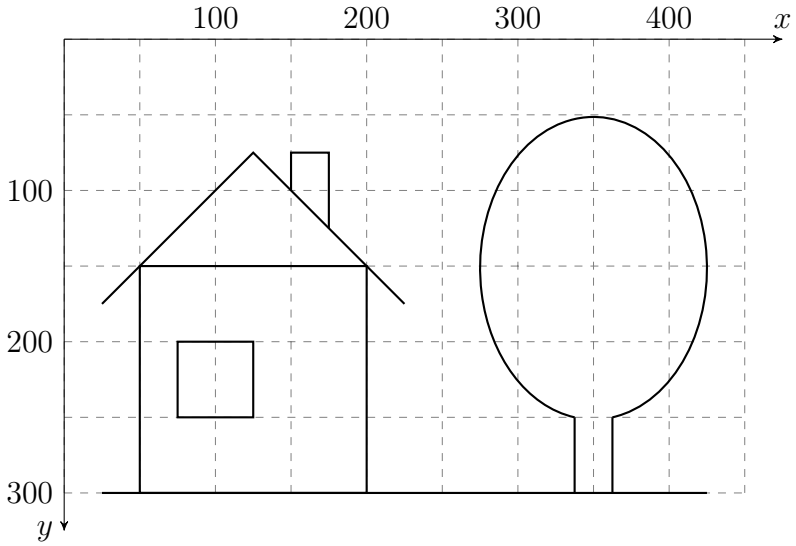

Рис. 8. Пример рисунка

гуры текущим цветом. Последний метод закрашивает сектор эллипса (фигуру, ограниченную дугой и двумя радиусами).

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

Упражнение 3.1. Дополните программу на рис. 4 и 6 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.

Упражнение 3.2. Напишите программу, рисующую картинку, изображенную на рис. 8 (без осей координат и координатной решетки).

Упражнение 3.3. Класс `JComponent`, являющийся родительским для всех компонентов Swing, не имеет представителя пользовательского интерфейса. Поэтому метод `paintComponent()` класса `JComponent` ничего не делает. Однако `JComponent` можно также использовать как поверхность для рисования. Замените `JPanel` на `JComponent` в классе на рис. 6, 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.

Список литературы

1. Портянкин И. Swing. Эффектные пользовательские интерфейсы. — 2-е изд. — Лори, 2011.

2. Хорстманн К. С., Корнелл Г. Java. Библиотека профессионала : в 2 т. — 9-е изд. — М. : ООО «И.Д. Вильямс», 2014.

3. Шилдт Г. Java 8. Полное руководство : в 2 т. — М. : ООО «И.Д. Вильямс», 2015.

4. Шилдт Г. Swing. Руководство для начинающих. — М. : ООО «И.Д. Вильямс», 2007.