Министерство общего и профессионального образования
Российской Федерации
РОСТОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Ю.А. Кирютенко В.А. Савельев

# Объектно-ориентированное программирование и язык Smalltalk.

## Ядро графики: классы Point, Rectangle, Form

Ростов-на-Дону
1997

**Ю.А. КирютенкоВ.А. Савельев**

# Объектно-ориентированное программирование и язык Smalltalk.

# Ядро графики: классы Point, Rectangle, Form

**Аннотация**

Методическая разработка посвящена современному направлению в программировании — объектно-ориентированной методологии программирования и языку Smalltalk и является продолжением вышедших ранее методических разработок по объектно-ориентированному программированию и языку Smalltalk, посвященных общим концепциям и синтаксису языка, интерфейсу пользователя и среде программирования, классам Collection, Magnitude, Stream и их подклассам, протоколам поддержки всех объектов и всех классов системы. В данной методической разработке описываются классы, обеспечивающие работу с графическими объектами системы. Всюду далее при описании классов языка Smalltalk мы будем иметь ввиду их реализацию в языке Smalltalk/V for DOS фирмы Digitalk (США).

Методическая разработка предназначена для студентов 3—5 курсов механико-математического факультета и слушателей ФПК.

Печатается в соответствии с решением кафедры математического анализа Ростовского государственного университета, протокол №4 от 14 октября 1997 года.

Настоящие методические указания набраны в системе LaTeX с использованием кириллических шрифтов семейства «Литературная» (AOParaGraph) и математических шрифтов семейств Computer Modern и Math Symbol (American Mathematical Society).

© 1997, Ю.А. КирютенкоВ.А. Савельев

# 1 Классы поддержки графики

Графические возможности Smalltalk/V основываются на битовой (растровой) графике. Весь пользовательский интерфейс среды программирования Smalltalk/V также реализован средствами битовой графики.

Основным в битовой графике является понятие битовой карты. Битовая карта — это линейный массив бит. Каждый бит может принимать значение 1 или 0, причем 1 представляет белый цвет, а 0 — черный. Поскольку экран терминала представляет собой прямоугольник, состоящий из пикселов (наименьших отдельно окрашиваемых элементов), то для организации двумерного представления битовой карты используется экземпляр класса Form (Форма). Помимо битовой карты, объект-форма содержит также информацию о ширине и высоте прямоугольного участка, в который должна «отображаться» битовая карта, это, в свою очередь позволяет манипулировать битовой картой как двумерным массивом бит. Каждому биту битовой карты ставится в соответствие пиксел на экране. Экран компьютера представлен в системе в виде специального объекта-формы — экземпляра класса DisplayScreen (ЭкранТерминала). Класс DisplayScreen является подклассом класса Form.

Все операции битовой графики сводятся к перемещениям групп бит битовой карты из одного положения в другое. Все эти операции можно было бы реализовать путем посылки соответствующих сообщений объекту-форме. Однако, учитывая достаточно сложный характер графических операций все они реализованы в классе BitBlt (сокращение от Bits Block Transfer — Передача Блока Бит).

В операциях битовой графики необходимо адресовать как единичные биты, так и целые области бит. Экземпляр класса Point (Точка) предназначен для адресации отдельных бит объекта-формы, тогда как экземпляр класса Rectangle (Прямоугольник) используется для адресации прямоугольной области бит в объекте-форме.

Таким образом, все что связано с графикой системы Smalltalk/V сосредоточено в классах Point, Rectangle, Form и BitBlt. В последующих разделах приводится описание как каждого из перечисленных выше классов, так и некоторых других, тесно с ними связанных. Начнем рассмотрение с классов поддержки графики.

## 1.1 Класс Point (Точка)

Экземпляр класса Point имеет две переменные x и y, которые обычно обозначают положение пикселя в форме относительным левого верхнего угла формы (или другого заданного начала координат). В соответствии с соглашением,

координата $x$ увеличиваясь перемешается вправо, а координата $y$ — вниз, что соответствует привычному для нас размещению текста на странице и направлению просмотра экрана дисплея. Это соглашение отличается от «правой» системы координат, принятой в математике, когда $y$ увеличивается при движении снизу вверх.

Точка обычно создается с помощью бинарного сообщения , посылаемого экземпляру класса Integer. Аргументом такого сообщения также является целое число. Первое целое число (получатель) задает x-координату точки, второе (аргумент) — y-координату. Например, результат вычисления выражения

```
200 @ 150
```

будет точкой с $x$- и $y$-координатами равными, соответственно, 200 и 150. Кроме этого, протокол класса Point поддерживает сообщение x: xInteger y: yInteger, создающее точку вида (xInteger, yInteger). Так выражение

```
Point x: 200 y: 150
```

при вычислении определит ту же самую точку, что и выражение 200 @ 150.

Протокол экземпляра класса Point поддерживает сообщения доступа и сравнения.

Класс Point — Протокол экземпляра

---

x — возвращает $x$-координату точки-приемника.

x: aNumber — устанавливает $x$-координату точки-приемника равной аргументу aNumber.

y — возвращает $y$-координату точки-приемника.

y: aNumber — устанавливает $y$-координату точки-приемника равной аргументу aNumber.

$<$ aPoint — возвращает true, если точка-приемник лежит выше и левее точки-аргумента aPoint, иначе возвращает false.

$<=$ aPoint — возвращает true, если точка-приемник лежит не ниже и не правее точки-аргумента aPoint, иначе возвращает false.

$>$ aPoint — возвращает true, если точка-приемник лежит ниже и правее точки-аргумента aPoint, иначе возвращает false.

$>=$ aPoint — возвращает true, если точка-приемник лежит не выше и не левее точки-аргумента aPoint, иначе возвращает false.

**max: aPoint** — возвращает нижний правый угол прямоугольника, заданного точкой-приемником и точкой-аргументом **aPoint**.

**min: aPoint** — возвращает верхний левый угол прямоугольника, заданного точкой-приемником и точкой-аргументом **aPoint**.

**between: aPoint and: bPoint** — возвращает **true**, если точка-приемник >= **aPoint** и точка-приемник <= **bPoint**, иначе возвращает **false**.

---

Вот несколько выражения, использующих сообщения их этого протокола.

| выражение | результат |
|---|---|
| (10 @ 100) x | 10 |
| (10 @ 100) y | 100 |
| (10 @ 100) x: 50 | 50 @ 100 |
| (10 @ 100) y: 50 | 10 @ 50 |
| (45 @ 230) < (175 @ 270) | true |
| (45 @ 230) < (175 @ 200) | false |
| (45 @ 230) > (175 @ 200) | false |
| (175 @ 270) > (45 @ 230) | true |
| (45 @ 230) max: (175 @ 200) | 175 @ 230 |
| (45 @ 230) min: (175 @ 200) | 45 @ 200 |
| 1 @ 2 between: 0 @ 2 and: 2 @ 2 | true |

Арифметические операции класса **Number** переносятся на экземпляры класса **Point**, как покоординатные операции, то есть их результатом является новый экземпляр класса **Point**, координаты которого получены применением указанной операции к соответствующим координатам точки-приемника и, если есть, точки-аргумента. Допускается также, чтобы в этих операциях вместо аргумента-точки использовался аргумент-число **aNumber** (экземпляр класса **Number**), который понимается как точка с $x$- и $y$-координатами равными числу **aNumber**. Отсечение и округление, использующее те же имена сообщений, что и класс **Number**, также включены в протокол экземпляра класса **Point**.

Класс **Point** — Протокол экземпляра

---

*** scale** — возвращает новую точку, которая имеет координаты равные произведению координат точки-приемника на аргумент **scale**, который может быть числом или точкой; если **scale** — точка, то происходит перемножение соответствующих координат точки-приемника и точки **scale**.

\+ **delta** — возвращает новую точку, которая имеет координаты равные сумме одноименных координат точки-приемника и аргумента **delta**.

− **delta** — возвращает новую точку, которая имеет координаты равные разности между одноименными координатами точки-приемника и аргумента **delta**.

// **scale** — возвращает новую точку, которая имеет координаты равные целому частному от деления координат точки-приемника на аргумент **scale** (с округлением в сторону отрицательной бесконечности).

\\\\ **scale** — возвращает новую точку, которая имеет координаты равные остатку от деления координат точки-приемника на аргумент **scale**.

**abs** — возвращает новую точку, которая имеет координаты равные абсолютным значениям координат точки-приемника.

**dotProduct: aPoint** — возвращает скалярное произведение точки-приемника на точку-аргумент **aPoint**.

**negated** — возвращает новую точку, которая имеет координаты противоположные по знаку значениям координат точки-приемника.

**rounded** — возвращает новую точку, которая имеет координаты равные округленным координатам точки-приемника.

**transpose** — возвращает новую точку, $x$-координата которой равна $y$-координате точки-приемника, а ее $y$-координата — $x$-координате приемника.

**truncated** — возвращает новую точку, которая имеет координаты равные координатам точки-приемника, усеченным до ближайшего целого не превосходящего их по абсолютной величине.

---

Чтобы лучше разобраться с описанными операциями с точками, посмотрите на следующие примеры.

| выражение | результат |
|---|---|
| (45 @ 230) + (175 @ 300) | 220 @ 530 |
| (45 @ 230) + 5 | 50 @ 235 |
| (45 @ 230) - (175 @ 300) | -130 @ -70 |
| (3 @ 22) - 10 | -7 @ 12 |
| (10 @ 20 ) * (3 @ 2) | 30 @ 40 |

| | |
|---|---|
| (160 @ 240) // 50 | 3 @ 4 |
| (160 @ 240) // (50 @ 30) | 3 @ 8 |
| (160 @ 240) \\ (50 @ 50) | 10 @ 40 |
| (-20 @ -30) abs | 20 @ 30 |
| ((45 @ 230) - (175 @ 300) abs | 130 @ 70 |
| (-2 @ 3) negated | 2 @ -3 |
| (2 @ 4) dotProduct: (5 @ 6) | 34 |
| (120.5 @ 220.7) rounded | 121 @ 221 |
| (120.5 @ 220.7) truncated | 120 @ 220 |
| (2 @ 4) transpose | 4 @ 2 |

Если для приложений приведенного протокола недостаточно, его нетрудно дополнить нужными методами. Например, можно определить метод, вычисляющий расстояние между двумя точками.

```
dist: aPoint
  "Возвращает расстояние между точкой-приемником
   и точкой-аргументом \cd{aPoint}."
   ↑ ((x - aPoint x) squared + (y - aPoint y) squared) sqrt
```

Тогда, вычисляя выражение (0@0) dist: (3@4), получим число 5.0.

## 1.2 Класс **Rectangle (Прямоугольник)**

Прямоугольники представляют прямоугольные области пикселей. Прямоугольник, как объект системы Smalltalk, является экземпляром класса Rectangle и имеет две переменные экземпляра origin и corner, которые задают две точки, определяющие прямоугольник: origin — верхний левый угол прямоугольника, corner — нижний правый угол прямоугольника. Ширина (width) и высота (height) прямоугольника могут быть вычислены следующим образом:

```
width  := corner x - origin y
height := corner y - origin y
```

Ширина и высота прямоугольника отражают количество бит по горизонтали и вертикали, соответственно, в прямоугольной области. Точка, задаваемая в виде width @ height называется размером (extent) прямоугольника. Размер прямоугольника можно вычислить следующим образом:

```
extent := corner - origin
```

Точки и прямоугольники используются вместе для поддержки графических операций. Поэтому создавать новый прямоугольник позволяют и протокол класса Point и протокол класса Rectangle.

Класс Point — Протокол экземпляра

---

corner: aPoint — возвращает прямоугольник с точкой-приемником в качестве origin и точкой-аргументом в качестве corner.

extent: aPoint — возвращает прямоугольник с точкой-приемником в качестве origin, с шириной и высотой равными координатам точки-аргумента aPoint.

---

В качестве иллюстрации рассмотрим два выражения, создающих одинаковые прямоугольники:

```
1 @ 1 corner: 100 @ 100.
1 @ 1 extent:  99 @ 99.
```

Кроме двух сообщений класса Point, создающих прямоугольники, протокол класса Rectangle поддерживает еще два сообщения для создания своих экземпляров. Эти сообщения определяют либо координаты противоположных углов, либо верхний левый угол и размеры области.

Класс Rectangle — Протокол класса

---

origin: originPoint corner: cornerPoint — возвращает прямоугольник с верхним левым и нижним правым углами, задаваемыми аргументами originPoint и cornerPoint.

origin: originPoint extent: extentPoint — возвращает прямоугольник с верхней левой точкой равной аргументу originPoint, с шириной и высотой равными координатам точки-аргумента extentPoint.

---

В качестве примера, приведем еще два выражения, создающих прямоугольники, одинаковые с теми, которые созданы чуть выше с помощью сообщений из протокола класса Point:

```
Rectangle origin: 1 @ 1 corner: 100 @ 100.
Rectangle origin: 1 @ 1 extent:  99 @ 99.
```

Класс **Rectangle** поддерживает протокол для определения параметров объекта прямоугольника и для их изменения, там где это возможно.

Класс **Rectangle** Протокол экземпляра

---

**bottom** — возвращает положение нижней стороны прямоугольника ($y$-координату точки **corner**).

**center** — возвращает точку, представляющую геометрический центр прямоугольника.

**corner** — возвращает точку, представляющую правый нижний угол прямоугольника.

**corner: aPoint** — изменяет в прямоугольнике-приемнике точку, представляющую правый нижний угол.

**extent** — возвращает размер прямоугольника в виде точки **width @ height**.

**extent: aPoint** — изменяет размеры прямоугольника в соответствии с точкой-аргументом, которая рассматривается как **width @ height**.

**height** — возвращает высоту прямоугольника.

**height: anInteger** — изменяет высоту прямоугольника, делая ее равной значению аргумента **anInteger**.

**left** — возвращает положение левой стороны приемника сообщения ($x$-координату точки **origin**).

**origin** — возвращает точку, представляющую левый верхний угол прямоугольника.

**origin: originPoint corner: cornerPoint** — изменяет верхний левый и нижний правый углы прямоугольника, делая их равными точкам-аргументам **originPoint** и **cornerPoint**, соответственно.

**origin: originPoint extent: extentPoint** — изменяет верхний левый угол прямоугольника и его размеры, устанавливая верхний левый угол равным точке **originPoint**, а ширину и высоту равными координатам точки **extentPoint**.

**right** — возвращает положение правой стороны прямоугольника ($x$-координату точки **corner**).

**top** — возвращает положение верхней стороны прямоугольника ($y$-координату точки **origin**).

**width** — возвращает ширину прямоугольника.

**width: anInteger** — изменяет ширину прямоугольника, делая ее равной значению аргумента **anInteger**.

---

Для демонстрации результатов посылки сообщений прямоугольникам создадим три прямоугольника **BoxA, BoxB** и **BoxC**:

```
BoxA := 50 @ 50 corner:  200 @ 200.
BoxB := 120 @ 120 corner: 260 @ 240.
BoxC := 100 @ 300 corner:  300 @ 400
```

Эти прямоугольники далее используются во всех выражениях этого раздела. Обратите внимания, что прямоугольники печатаются в виде **originPoint corner: cornerPoint**.

| выражение | результат |
|---|---|
| BoxA top | 50 ($y$-координата точки origin) |
| BoxA bottom | 200 ($y$-координата точки corner) |
| BoxB left | 120 ($x$-координата точки origin) |
| BoxB right | 260 ($x$-координата точки corner) |
| BoxA center | 125 @ 125 (геометрический центр) |
| BoxC width | 200 (ширина прямоугольника) |
| BoxC heigth | 100 (высота прямоугольника) |
| BoxC origin | 100 @ 300 |
| BoxC corner | 300 @ 400 |

---

Следующий протокол класса **Rectangle** позволяет, как результат вычислений, создавать новые прямоугольники. При этом используются арифметические операции с точками, определяющими прямоугольник.

Класс **Rectangle** — Протокол экземпляра

---

**expandBy: delta** — возвращает прямоугольник, который больше приемника на аргумент **delta**, который может быть прямоугольником, точкой или числом.

**insetBy: delta** — возвращает прямоугольник, который вложен в приемник с отступом от его границы на аргумент **delta**, который может быть прямоугольником, точкой или числом.

**intersect: aRectangle** — возвращает прямоугольник, который является пересечением прямоугольником, приемником сообщения, с прямоугольником-аргументом **aRectangle**.

**nonIntersections: aRectangle** — возвращает упорядоченный набор (экземпляр класса **OrderedCollection**) прямоугольников, объединение которых представляет область прямоугольника-приемника, расположенную вне прямоугольника-аргумента **aRectangle**.

**merge: aRectangle** — возвращает наименьший прямоугольник, который содержит и прямоугольник-приемник и прямоугольник-аргумент **aRectangle**.

---

В следующей таблице приведены примеры, использующие сообщения из этого протокола. Обратите внимание на последний пример.

| выражение | результат |
|---|---|
| BoxC expandBy: 10 | 90 @ 290 corner: 310 @ 410 |
| BoxC expandBy: 10 @ 20 | 90 @ 280 corner: 310 @ 420 |
| BoxC insetBy: 10 | 110 @ 310 corner: 290 @ 390 |
| BoxC insetBy: 10 @ 20 | 110 @ 320 corner: 290 @ 380 |
| BoxA intersect: BoxB | 120 @ 120 corner: 200 @ 200 |
| BoxB merge: BoxC | 100 @ 120 corner: 300 @ 400 |
| BoxA nonIntersections: BoxB | OrderedCollection(<br>50 @ 50 corner: 200 @ 120<br>50 @ 120 corner: 120 @ 200) |

Протокол тестирования класса **Rectangle** включает в себя сообщения для определения того, содержится ли некоторая точка внутри границ данного прямоугольника, пересекаются два прямоугольника или нет.

Класс **Rectangle** Протокол экземпляра

---

**containsPoint: aPoint** — проверяет, содержится ли внутри приемника сообщения точка-аргумент **aPoint**.

**intersects: aRectangle** — проверяет, пересекается ли прямоугольник, приемник сообщения, с прямоугольником-аргументом **aRectangle**.

---

Продолжим рассмотрение вышеупомянутых прямоугольников **BoxA, BoxB, BoxC.**

| выражение | результат |
|---|---|
| BoxA containsPoint: 350 @ 150 | false |
| BoxC containsPoint: 200 @ 320 | true |
| BoxA intersects: BoxB | true |
| BoxA intersects: BoxC | false |

Подобно координатам точки, координаты прямоугольника могут быть округлены или усечены до ближайшего целого числа. Прямоугольник может быть перемещен на указанную величину, перемещен в заданную позицию и его координаты могут быть масштабированы или сдвинуты на некоторую заданную величину.

Класс **Rectangle** — Протокол экземпляра

---

**rounded** — изменить координаты прямоугольника-приемника, округляя их до ближайшего целого.

**moveBy: aPoint** — сдвинуть прямоугольник на вектор **aPoint**.

**moveTo: aPoint** — сдвинуть прямоугольник так, чтобы его верхний левый угол совпал с аргументом **aPoint**.

**scaleBy: scale** — возвращает прямоугольник, представляющий собой приемник, у которого переменные экземпляра равняются отмасштабированными переменным экземпляра приемника сообщения на аргумент **scale**, который является или точкой или числом.

**scaleTo: aRectangle** — возвращает прямоугольник, размер которого пропорционален прямоугольнику, приемнику сообщения, в отношении, определяемом аргументом **aRectangle**.

**translateBy: delta** — возвращает прямоугольник, который сдвинут по отношению к приемнику на аргумент **delta**, который является или точкой или числом.

**truncated** — изменяет координаты прямоугольника, усекая их до ближайшего целого.

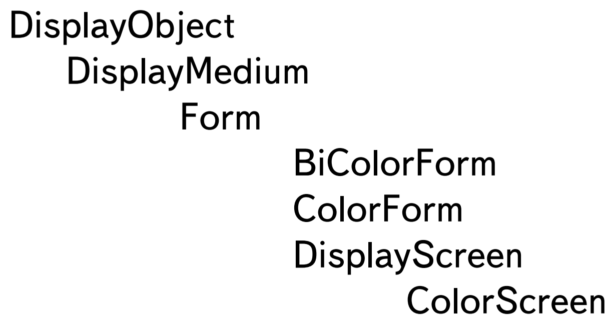
---

Применим эти сообщения к нашим прямоугольникам BoxA, BoxB и BoxC. Если использование в примерах сообщений предыдущих протоколов не изменяло эти прямоугольники, то первые два выражения следующей таблицы делают это.

| выражение | результат |
|---|---|
| BoxA moveBy: 50 @ 50 | 100 @ 100 corner: 250 @ 250 |
| BoxA moveTo: 200 @ 300 | 200 @ 300 corner: 350 @ 450 |
| BoxA scaleBy: 2 | 400 @ 600 corner: 700 @ 900 |
| BoxA translateBy: -100 | 100 @ 200 corner: 250 @ 350 |

## 2 Класс Form (Форма)

Класс Form (Форма) является подклассом в иерархии, определяемой суперклассом DisplayObject.




Классы DisplayMedium (СредаОтображения) и DisplayObject (ОтображаемыйОбъект) не определяют никаких переменных, являются абстрактными классами и не создают экземпляров. Они предназначены для группирования взаимосвязанных методов так, чтобы последние могли наследоваться подклассами. Подробнее о них мы поговорим позже.

В свою очередь, как мы видим, класс Form имеет несколько подклассов, среди который сразу отметим и кратко опишем класс DisplayScreen (Экран). В системе существует обычно один экземпляр класса DisplayScreen, он имеет имя

Display, которое является глобальной переменной системы. Display используется для того, чтобы обрабатывать общие запросы пользователя и имеет дело с физически существующим в данной вычислительной системе дисплеем. Каждый раз, при смене терминала или используемого графического режима, содержимое этой переменной инициализируется заново, чтобы отражать текущие характеристики экрана. С переменной Display можно работать также как и с любой другой формой, однако при копировании группы бит в объект-форму, связанную с глобальной переменной Display, все изменения немедленно отобразятся на экране. Другими словами, содержимое формы Display отражает содержимое экрана терминала.

Кроме сообщений, посылаемых экземплярам, которые наследуется из суперклассов DisplayObject, DisplayMedium и Form, класс DisplayScreen обеспечивает вывод содержимого экрана на принтер и переопределяет несколько сообщений специфической обработки форм, представляющих экран. Протокол сообщений посылаемых классу DisplayScreen позволяет настроить работу системы на существующем терминале и, самое главное, запустить систему Smalltalk/V посредством посылки классу DisplayScreen сообщения initSystem.

Бывает необходимость существования в системе нескольких экземпляров класса DisplayScreen, например, когда программируется полноэкранная мультипликация (с двойной буферизацией). Обычно, полноэкранная мультипликация используется крайне редко, вместо нее выполняется мультипликация внутри небольшой прямоугольной области экрана, при которой скрытый буфер используется для хранения битов следующего изображения, а каждое новое изображение получается копированием этого буфера на выделенную прямоугольную область.

Определение класса Form в системе Smalltalk/V имеет вид:

```
DisplayMedium subclass: #Form
    instancesVariableNames:
        'bits width height offset byteWidth deviceType '
    classVariableNames:
        'WhiteMask PrinterMode BlackMask
        DarkGreyMask LightGreyMask GreyMask '
    poolDictionaries: 'CharacterConstants '
```

Основное предназначение объекта-формы (экземпляра класса Form) — организация двухмерного представления для битовой карты (bits). Два рассмотренных ранее класса, Rectangle и Point, широко используются при работе с изображениями. Экземпляр класса Point (точка) содержит значения координат x и y, и используется для ссылок на положение пикселя в форме. Прямоугольник, определяя две точки — верхний левый угол (origin) и нижний правый угол

(corner), используется для определения прямоугольный области, занимаемой экземпляром класса Form: origin определяется по переменной offset (они равны), а corner = offset + (width @ height). Рассмотрим подробнее все переменные объекта-формы и их предназначение. Начнем с переменных экземпляра:

**bits (биты)** — содержит битовую карту, экземпляр класса Bitmap.

**width (ширина)** — содержит горизонтальный размер формы в пикселах.

**height (высота)** — содержит вертикальный размер формы в пикселах.

**offset (смещение)** — указывает вектор смещения формы при выводе формы-приемника; по умолчанию вектор смещения равен 0@0.

**byteWidth (ширинаВБайтах)** — содержит горизонтальный размер формы в байтах. Используется для эффективной реализации операций вывода формы.

**deviceType (типУстройства)** — задает тип устройства, к которому принадлежит форма. При создании объекта-формы эта переменная принимает значение 0, указывая на то, что форма расположена в обычной оперативной памяти. При создании объекта класса DisplayScreen (подкласса класса Form), переменная устанавливается в 1, указывая на то, что созданная форма рассматривается как буфер экрана терминала, при этом его адрес и размеры зависят от типа графического адаптера и текущего графического режима. Если же форма является экземпляром класса BiColorForm или ColorForm, подклассов класса Form, то эта переменная принимает значения 2 или 3, соответственно. Как и предыдущая переменная непосредственно при программировании не используется.

Теперь обратимся к переменным класса.

**WhiteMask** — содержит форму маски белого цвета.

**BlackMask** — содержит форму маски черного цвета.

**DarkGreyMask** — содержит форму маски темно-серого цвета.

**GreyMask** — содержит форму маски серого цвета.

**LightGreyMask** — одержит форму маски светло-серого цвета.

**PrinterMode** — содержит строку (экземпляр класса aString) для переключения принтера в графический режим (например, Esc K).

Все эти переменные, кроме последней, — ссылаются на специальные формы, которые называются масками (mask) и используются при отображении формы на другую форму (например, на экран компьютера). Форма-маска имеет фиксированную ширину и высоту, равную 16 пикселам. Если размер формы-маски меньше чем размер прямоугольника отображаемой формы, форма-маска последовательно сдвигается, покрывая весь исходный прямоугольник. При этом на каждом шаге происходит комбинирование бит формы-маски и отображаемой формы (формы-источника) по правилу логического "И". Использование формы-маски приводит к эффекту «полутоновый текстуры», поэтому форму-маску иначе еще называют полутоном (halftone). По умолчанию используется белый полутон (nil-маска), все биты которой равны 1. Следовательно, в этом случае биты формы-источника при их комбинировании с полутоном не изменяются.

Встроенные в систему полутона и являются переменными класса в классе **Form**. Чтобы их получить достаточно послать классу **Form** соответствующее сообщение. Вот их полный список:

| выражение | возвращаемый полутон |
|---|---|
| **Form white** | белый полутон (**nil**-маска) |
| **Form black** | черный |
| **Form gray** | серый |
| **Form darkGray** | темно-серый |
| **Form lightGray** | светло-серый |

Для создания объекта-формы существует ряд сообщений, посылаемых классу **Form**. Ниже приведено описание некоторых из них:

Класс **Form** — Протокол класса

**fromDisplay** Возвращает объект-форму, размеры которой совпадают с размером экрана, а содержимое формы копируется из прямоугольной области экрана.

**fromDisplay: aRectangle** Возвращает объект-форму, размеры которой совпадают с размером прямоугольника **aRectangle**. Содержимое формы копируется из прямоугольной области экрана, описываемой прямоугольником **aRectangle**.

**fromUser** Предлагает пользователю указать прямоугольную область на экране. Затем возвращает объект-форму, содержимое которой скопировано из указанной области экрана.

**fromUserSize: aPoint** Предлагает пользователю указать прямоугольную область на экране. Начальный размер области задается точкой **aPoint**. Возвращает объект-форму, содержимое которой скопировано из указанной области экрана.

**width: wInteger height: hInteger** Возвращает белую форму (все биты битовой карты установлены в 1). Ширина формы задается целым числом **wInteger**, высота — целым числом **hInteger**.

**new** Возвращает форму с единственной инициализированной значением 0 переменной **deviceType**, что означает форму, находящуюся в основной памяти.

---

Для создания объекта-формы можно также воспользоваться упомянутым сообщением **new**, посылаемым классу **Form**, а затем с помощью описанных ниже сообщений, инициализировать все оставшиеся переменные экземпляра созданного

Класс **Form** Протокол экземпляра

---

**extent: aPoint** Изменить ширину и высоту формы. Ширина и высота задаются $x$ и $y$ координатами точки **aPoint**, соответственно.

**width: w height: h** Изменить ширину формы на значение целого числа **w**, а высоту — на значение целого числа **h**. Создать битовую карту соответствующего размера из единиц.

**width: w height: h initialByte: aByte** Изменить ширину и высоту формы на значения **w** и **h**, соответственно. Все байты битовой карты инициализировать значением **aByte** (когда **aByte** равен **16rFF**, получается белая форма, а когда 0 — черная).

**copy: aRectangle from: aForm to: aPoint rule: anInteger** Скопировать заданный прямоугольника **aRectangle** из заданной формы **aForm** в точку **aPoint** формы-получателя с использованием правила комбинирования **anInteger**.

---

Как и всякий класс, протокол класса Form содержит множество сообщений, которые позволяют получить и установить значение любой переменной экземпляра (get-методы и set-методы). В качестве примера опишем только те, которые имеют отношение к переменной bits или к правилам ее использования.

Класс Form Протокол экземпляра

---

bitmap — возвращает битовую карту формы-приемника (значение переменной экземпляра bits).

at: aPoint — возвращает бит (0 или 1) в позиции aPoint из битовой карты формы.

at: aPoint put: aBit — устанавливает бит в позиции aPoint из битовой карты формы-приемника в соответствии с аргументом aBit (в 0 или в 1).

byteValueAt: a Point put: aByte — заменяет байт в позиции aPoint битовой карты на байт aByte.

byteValueAtX: xInteger Y: yInteger — возвращает байт битовой карты в позиции, определяемой точкой xInteger @ yInteger.

compatibleMask — возвращает имя класса формы-маски наиболее пригодной для использования с формой-приемником.

---

# 3 Вывод формы на экран и принтер

Созданную форму можно увидеть, если вывести ее на экран. Для этого нужно использовать сообщения из протокола класса DisplayObject. Рассмотрим этот класс подробнее. Прежде всего отметим, что экземпляр класса Form — один из видов отображаемых на экране дисплея объектов. Существуют и другие. Способ, которым эти объекты реализуются, описывается в иерархии классов системы с суперклассом DisplayObject. Класс Form — один из подклассов в этой иерархии, добавляющий растровое представление изображения.

Любой отображаемый объект представляет собой изображение, которое имеет ширину, высоту, предполагаемое начало в точке 0@0 и смещение относительно этого начала (offset), согласно которому этот образ должен выводиться на экран. Все отображаемые объекты подобны в своей способности копировать себя на другое изображение, масштабировать и перемещать себя на экране. При этом они отличаются друг от друга способом создания изображения.

Протокол класса DisplayObject прежде всего поддерживает сообщения для работы с переменными изображения.

Класс DisplayObject — Протокол экземпляра

---

**width** Возвращает ширину той прямоугольной рамки, которая ограничивает приемник.

**height** Возвращает высоту той прямоугольной рамки, которая ограничивает приемник.

**extent** Возвращает экземпляр класса Point (точку), представляющий ширину и высоту той прямоугольной рамки, которая ограничивает приемник.

**offset** Возвращает точку, представляющую величину, на которую приемник должен быть сдвинут, когда его изображение прорисовывается на экране или когда определяется его местоположение.

**offset: aPoint** Устанавливает смещение offset приемника равным аргументу aPoint.

**boundingBox** Возвращает прямоугольную область с началом в точке offset, которая представляет границы приемника.

---

Как уже отмечалось, основное предназначение класса DisplayObject обеспечивать вывод объектов на экран. Именно к таким относятся следующие два сообщения.

Класс DisplayObject — Протокол экземпляра

---

**displayAt: aPoint** — вывести приемник на экран в точке с координатами aPoint.

**display** — вывести приемник на экран в точке с координатами 0@0.

---

Вот несколько простых примеров. Текст примера описывает изображение, которое появиться на экране в результате выполнения ниже приводимых выражений.

**Пример 3.1.** Белый прямоугольник шириной 100 пикселов и высотой 50 пикселов, верхний левый угол которого совпадает с верхней левой точкой экрана.

```
|f|
f := Form width: 100 height: 50.
f displayAt: 0 @ 0
```

**Пример 3.2.** Белый прямоугольник шириной 100 пикселов и высотой 50 пикселов, верхний левый угол которого совпадает с точкой экрана 20 @ 10.

```
|f|
f := Form width: 100 height: 50.
f offset: 20 @ 10.
f displayAt: 0 @ 0
```

**Пример 3.3.** Тот же самый прямоугольник, что и в примере 2, но создан с помощью других сообщений, которые позволяют указать явно байт заполнения формы.

```
|f|
f := (Form new width: 100 height: 50 initialByte: 16rFF).
f offset: 20 @ 10.
f displayAt: 0 @ 0
```

**Пример 3.4.** Прямоугольник шириной 100 пикселов и высотой 50 пикселов, верхний левый угол которого совпадает с верхней левой точкой экрана. Прямоугольник состоит из белых и черных вертикальных полосок, каждая шириной в 4 пиксела.

```
|f|
f := (Form new width: 100 height: 50 initialByte: 16rF0).
f displayAt: 0 @ 0
```

Форму можно вывести не только на экран, но и на печатающее устройство. Эта возможность обеспечивается самим классом Form.

Класс Form — Протокол экземпляра

---

outputToPrinter — выводит содержимое формы на графическое печатающее устройство повернутым на 90 градусов по часовой стрелке.

outputToPrinterUpright — выводит содержимое формы на текущее графическое устройство вертикально.

---

**Пример 3.5.** Прямоугольник из примера 3.4 будет напечатан точно так же, как он виден на экране, если выполнить следующие выражения.

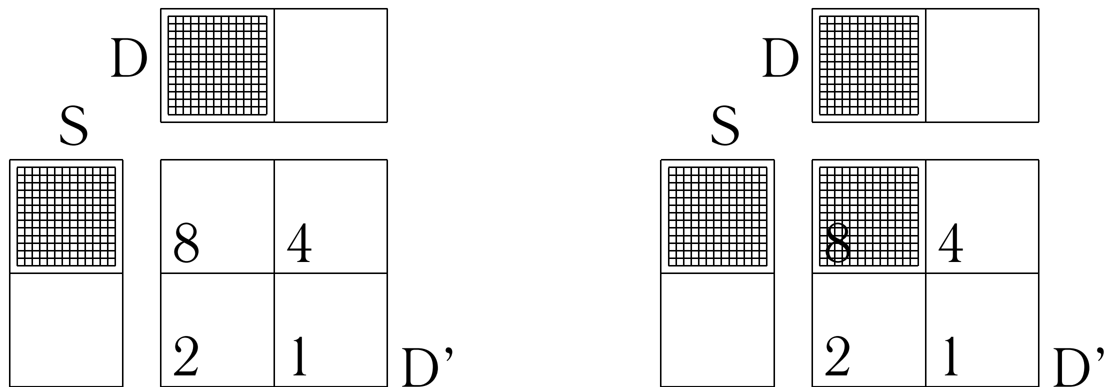



Рис. 1: Модель для определения числа, соответствующего правилу комбинирования, и правило **"ИЛИ"**.

```
|f|
f := (Form new width: 100 height: 50 initialByte: 16rF0).
f outputToPrinterUpright
```

Протокол сообщений класса DisplayObject не ограничивается перечисленными простыми сообщениями. Он значительно богаче, но чтобы понять более сложные сообщения, нужно ввести новые важные понятия.

## 3.1 Правила комбинирования

Все предыдущие примеры при выводе формы на экран основывались только на форме-источнике и форме-маске, которая пока всегда была по умолчанию белой маской и потому не меняла формы источника, а результат всегда сохранялся непосредственно на экране. Напомним, что экран — это тоже форма, которая имеет собственное содержание, которое до сих пор не принималось во внимание. Однако, содержимое формы-цели (в предыдущих примерах — экрана) может также приниматься во внимание при определении результата вывода формы-источника. Для этого вводится комбинационное правило (правило комбинирования), которое задает способ комбинирования бит формы-источника (после их логического перемножения на биты маски!) и бит формы-цели (целевой формы). Напомним, что белому пикселу соответствует единичный бит, а черному — нулевой. Существуют различные способы комбинирования бит формы-источника и формы-цели. Каждое такое правило должно определять будет результат белым или черным в следующих четырех случаях: форма-источник является белой или черной и форма-цель является белой или черной. Различные комбинационные правила можно получить, посылая соответствующие сообще-

ния классу Form. Результатом посылки сообщения является целое число, определяющее способ комбинирования.

Класс Form — Протокол класса

---

**andRule** — возвращает целое число 1, соответствующее комбинаторному правилу “под”. По этому правилу биты формы-источника логически перемножаются с битами формы-цели.

**under** — совпадает с правилом Form andRule.

**over** — возвращает целое число 3, соответствующее комбинаторному правилу «поверх». По этому правилу биты формы-источника полностью замещают соответствующие биты формы-цели.

**erase** — возвращает целое число 4, соответствующее комбинаторному правилу «удаление». По этому правилу биты целевой формы устанавливаются в 0, если соответствующие биты исходной формы установлены в 1.

**reverse** — возвращает целое число 6, соответствующее комбинационному правилу «инверсия». По этому правилу биты формы-источника и формы-цели комбинируются по правилу «исключающего или» (XOR)

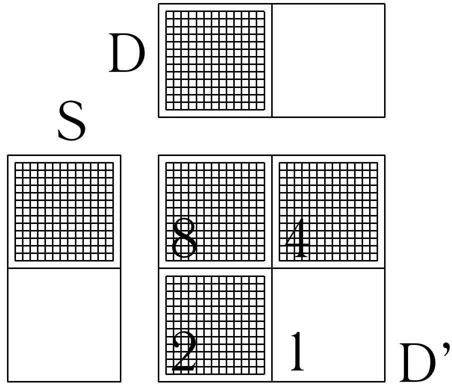
**orRule** — возвращает целое число 7, соответствующее комбинаторному правилу «или». По этому правилу биты формы-источника логически складываются с битами формы-цели.

---

Во всех приведенных ранее примерах комбинационное правило явно не указывалось. Как уже отмечалось, по умолчанию всегда используется комбинационное правило Form over.

Как легко подсчитать всего существует 16 правил комбинирования. Но почему именно этим правилам присвоены именно эти целые числа? Чтобы это объяснить, прежде всего напомним, что белому биту соответствует 1, а на наших поясняющих рисунках ему соответствует незакрашенный квадратик формы, черному биту — 0, при этом соответствующий квадратик закрашен. Теперь рассмотрим рисунок (Рис. 1), в левой части которого изображены: слева — форма-источник (S), сверху — форма-цель (D), между этими формами внизу справа — квадрат D’ с пронумерованными (степенями двойки) четырьмя ячейками, соответствующими четырем возможным случаям, с которыми мы сталкиваемся при комбинировании бит источника S и цели D. Например, ячейка с номером 2 соответствует случаю, когда источник был белый, а цель — черная.

andRule: 1; D'= S and D




over: 1+2=3; D'= S

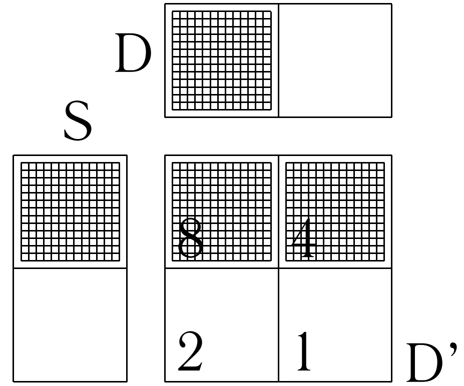
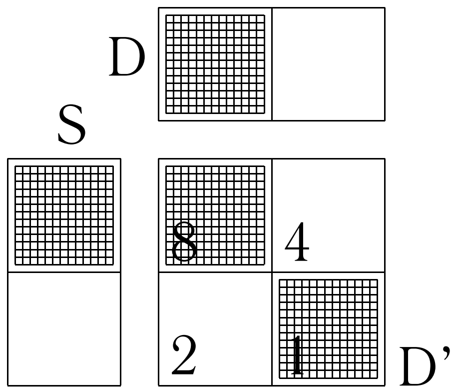



Рис. 2: Правило **“ПОД”** и **“ПОВЕРХ”**.

reverse: 2+4=6; D'= S xor D




erase: 4. D'= (not S) and D

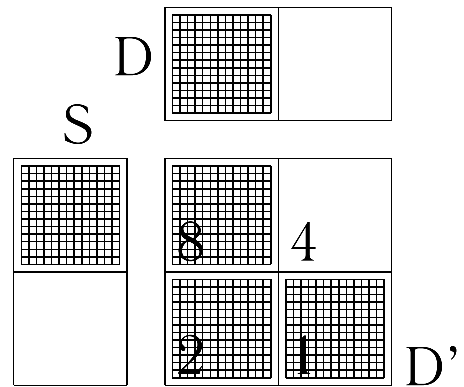



Рис. 3: Правила **“ИНВЕРСИЯ”** и **“УДАЛЕНИЕ”**.

Соответствующим образом закрашивая или нет четыре ячейки в квадрате D' (то есть делая их черными или белыми), можно описать любое правило комбинирования. Сумма чисел в незакрашенных ячейках квадрата D' и определяет данное комбинационное правило. Например, по правилу orRule, графическое пояснение к которому приведено в правой части рисунка 1, результат должен быть белым, когда или источник или цель (или оба) являются белыми, поэтому надо оставить в форме D' незакрашенными ячейки с номерами 4, 2, 1. Целое число для определения этого правила получается как сумма номеров соответствующих ячеек, то есть 4+2+1=7.

Приведенные в разделе рисунки 2 и 3 иллюстрируют остальные четыре комбинаторных правила из протокола класса Form. На рисунках, дополнительно, каждое из правил описывается (в соответствии с комбинаторной диаграммой) целым числом, обозначающим это правило, и логической формулой операций над битами.

## 3.2 Отсекающий прямоугольник

Кроме правила комбинирования между битами формы-источника и формы-цели, можно еще указать отсекающий прямоугольник (clippingBox) или, по-другому, прямоугольник отсечения, позволяющий задать границы прямоугольного участка в форме-цели, биты которого, в независимости от других параметров, смогут комбинироваться с битами формы-источника при выводе. Часто желательно, чтобы изображение появилось или изменилось только в некоторой части большого образа, именно прямоугольник отсечения гарантирует, что все элементы изображения появятся или изменятся только в пределах этой области. Например, для каждой панели в любом окне всегда задан отсекающий прямоугольник, который вычисляется как при открытии окна, так и при его перемещении или изменении размеров. Поэтому нет необходимости принимать какие-либо специальные меры контроля за тем, чтобы выводимая в панель информация не попала за ее пределы. По умолчанию в качестве прямоугольника отсечения всегда принимается экран.

## 3.3 Полностью контролируемый вывод на экран

Класс DisplayObject содержит еще несколько отображающих сообщений, которые используют правило комбинирования и отсекающий прямоугольник. Первое сообщение протокола — самое общее, оно явно устанавливает все параметры, необходимые для операции вывода изображения на экран. В следующих двух сообщениях этого протокола явно устанавливается только часть параметров. Неуказанные параметры устанавливаются по умолчанию:

Display — среда отображения (аргумент aDisplayMedium), которая в этом случае указывает на глобальную переменную системы Smalltalk/V);

0 @ 0 — позиция размещения изображения на экране (аргумент aDisplayPoint);

0@0 extent: Display extent — прямоугольник отсечения (аргумент aRectangle);

Form over — комбинаторное правило (аргумент ruleInteger);

nil или Form white — белая маска (аргумент aForm).

Класс DisplayObject Протокол экземпляра

---

displayOn: aDisplayMedium
at: aDisplayPoint
clippingBox: aRectangle
rule: ruleInteger
mask: aForm
— отображает образ-приемник на среду отображения aDisplayMedium в положении aDisplayPoint в соответствии с правилом комбинирования ruleInteger и полутоновой маской aForm. При этом, отображаемый объект ограничен областью, которая указывается отсекающим прямоугольником aRectangle.

displayAt: aPoint clippingBox: aRectangle Отображает образ-приемник на экран в точке aPoint, с использованием отсекающего прямоугольника aRectangle.

displayAt: aPoint rule: anInteger Отображает образ-приемник на экране в точке aPoint, с использованием правила комбинирования anInteger.

---

Давайте вернемся к примерам по отображению форм на экран и рассмотрим примеры применения выше перечисленных сообщений.

**Пример 3.6.** Прямоугольник шириной 100 пикселов и высотой 50 пикселов, верхний левый угол которого совпадает с верхней левой точкой экрана. Прямоугольник состоит из черных и белых вертикальных полосок, каждая шириной в 4 пиксела (инвертированный прямоугольник из примера 3.4).

```
|f|
f := (Form new width: 100 height: 50 initialByte: 16rF0).
f displayAt: 0 @ 0 rule: Form reverse
```

**Пример 3.7.** Прямоугольник шириной 30 пикселов и высотой 30 пикселов, верхний левый угол которого совпадает с точкой экрана 10 @ 10. Прямоугольник раскрашен также, как и прямоугольник из примера 3.4.

```
|f|
f := (Form new width: 100 height: 50 initialByte: 16rF0).
f displayAt: 0 @ 0 clippingBox: ((10 @ 10) corner: (40 @ 40))
```

**Пример 3.8.** Прямоугольник шириной 120 пикселов и высотой 140 пикселов, верхний левый угол которого совпадает с точкой экрана 480 @ 10. Прямоугольник состоит из черных и серых вертикальных полосок. Еще раз обратите внимание на то, что прямоугольник отсечения связывается с формой-целью (с экраном), а не с формой-источником.

```
|f|
f := (Form new width: 150 height: 150 initialByte: 16rF0).
f displayOn: Display
  at: 480 @ 0
  clippingBox: ((100 @ 10) corner: (600 @ 450))
  rule: Form over
  mask: Form gray
```

# 4 Класс DisplayMedium

Класс DisplayMedium (СредаОтображения), с некоторыми сообщениями которого мы уже познакомились, является подклассом класса DisplayObject и представляет объекты, на которые могут копироваться изображения-источники. Кроме сообщений, унаследованных из суперкласса, класс DisplayMedium обеспечивает протокол для окраски изображения и размещения границ вокруг прямоугольников, содержащих изображение. Другими словами этот класс представляет изображения, которые могут «окрашиваться» (заполняться серым оттенком) и окаймляться (то-есть вкладываться в прямоугольник так, что граница этого прямоугольника может «окрашиваться»).

Цвета окраски — это формы-маски, которые известны системе, благодаря классу Form:

black — растр состоит из 0 (черный),

white — растр состоит из 1 (белый),

gray — растр состоит из смеси нулей и единиц (серый).

С помощью следующих сообщений вся среда или часть среды отображения (экземпляра класса DisplayMedium) может раскрашиваться в один из этих цветов. В приводимых сообщениях, начало (origin) прямоугольника aRectangle должно находиться в системе координат приемника. Последнее сообщения протокола — основное, с его помощью реализуются все остальные, при этом используется комбинаторное правило Form over.

Класс DisplayMedium — Протокол экземпляра

---

black — закрасить всю область приемника черным (black).

black: aRectangle — закрасить область приемника, заключенную внутри прямоугольной области aRectangle, черным (black).

white — раскрасить всю область приемника белым (white).

white: aRectangle — закрасить область приемника, заключенную внутри прямоугольной области aRectangle, белым (white).

gray — закрасить всю область приемника серым (gray).

gray: aRectangle — закрасить область приемника, заключенную внутри прямоугольной области aRectangle, серым (gray).

fill: aRectangle
clippingBox: clipRectangle
rule: anInteger
mask: aForm
— закрасить область приемника, определенную аргументом aRectangle, внутри прямоугольника отсечения clipRectangle, заполняя ее формой-маской размером $16 \times 16$ бит, заданной аргументом aForm; при этом комбинаторное правило при копировании маски в образ-приемник задается аргументом anInteger.

---

**Пример 4.1.** Предположим, что pict — экземпляр класса Form, который имеет 100 пикселей в ширину и 100 пикселей в высоту, а box — экземпляр класса Rectangle с началом в точке 30@30, с шириной и высотой по 40 пикселей. Тогда протокол закраски черным цветом подобласти в pict, представленной объектом box, иллюстрируется следующей последовательностью выражений. Изображение, выводимое на экран, — это черный прямоугольник box размещенный внутри белого прямоугольника с началом в точке 0@0.

```
|pict box|
pict := Form width: 100 height: 100.
box := 30 @ 30 extent: 40 @ 40.
pict black: box.
pict display.
```

Класс **DisplayMedium** Протокол экземпляра

---

**border: aRectangle**
**clippingBox: clipRectangle**
**rule: anInteger**
**mask: aForm**
создает в заданном первым аргументом прямоугольнике **aRectangle** бордюр шириной 1, экран дисплея при этом ограничивается прямоугольником отсечения **clipRectangle**. Бордюр строится наложением заданной формы-маски **aForm** на форму-цели с помощью заданного правила комбинирования **anInteger**.

**border: aRectangle**
**rule: anInteger**
**mask: aForm**
создает в заданном прямоугольнике **aRectangle** бордюр шириной 1 по заданному правилу комбинирования **anInteger** с заданной маской **aForm**.

**border: aRectangle** Создает в заданном прямоугольнике **aRectangle** бордюр шириной 1 пиксел.

---

**Пример 4.2.** Если выполнить приведенные ниже выражения, то на экране в точке 0@0 отобразиться белый прямоугольник, внутри которого по периметру серого прямоугольника **box** и вне его, будет проведен черный бордюр (граница) толщиной в 1 пиксел.

```
|pict box|
 pict := Form width: 100 height: 100.
 box := 30@30 corner: 70@70.
 pict gray: box; border: box.
 pict display.
```

# 5 Использование цвета

Все о чем мы говорили до сих пор, касалось черно-белого вывода на экран и создания черно-белых форм и их серых оттенков разной интенсивности. Но

система позволяет работать и с цветом. Для этого используются формы и маски-формы, являющиеся экземплярами классов BiColorForm (ДвуцветнаяФорма) и ColorForm (ЦветнаяФорма). Сначала рассмотрим класс BiColorForm.

```
Form subclass: #BiColorForm
  instanceVariableNames: 'foreColor backColor '
  classVariableNames: ' '
  poolDictionaries: ' '
```

Определяемые этим классом две новые переменные экземпляра, ссылаются на экземпляры класса SmallInteger, которые описывают цвет символа foreColor и цвет фона backColor. По умолчанию переменная foreColor ссылается на 15 (белый символ), а backColor — на 0 (черный фон). Создать новую форму этого класса можно одним из следующих сообщений.

Класс BiColorForm — Протокол класса

---

new — создает новую форму с цветами по умолчанию.

color: aColor — создает новую форму-маску с цветом символа aColor и черным цветом фона.

foreColor: aColor backColor: bColor — создает новую форму-маску с цветом символа aColor и черным фона bColor.

---

Протокол сообщений экземпляра класса BiColorForm позволяет узнать установленные в форме цвета и переопределить их, или полностью изменить структуру формы в соответствии с содержимым выделенной области экрана.

Класс BiColorForm — Протокол экземпляра

---

backColor — возвращает цвет фона.

backColor: aColor — устанавливает цвет фона приемника равным аргументу aColor.

foreColor — возвращает цвет символа.

foreColor: aColor — устанавливает цвет символа приемника равным аргументу aColor.

**foreColor: aColor backColor: bColor** — устанавливает цвет символа приемника равным аргументу **aColor**, а цвет фона приемника равным аргументу **bColor**.

**fromDisplay: aRectangle** — перенести в форму-приемник содержимое экрана, заданное прямоугольником **aRectangle**.

---

Чтобы правильно обрабатывать экземпляры своих подклассов, сам класс **Form** имеет несколько специальных сообщений, о которых мы еще не упоминали, как в протоколе методов класса, так и в протоколе методов экземпляра. Чтобы возвращаемые значения приводимых ниже сообщений были понятны, напомним, что экземпляр класса **Form**, а значит и экземпляры всех его подклассов имеют переменную **deviceType**. Ее значения для экземпляров разных подклассов класса **Form** различны. Как раньше уже отмечалось, оно равно 1, если это экранная форма (то есть экземпляра класса **DisplayScreen**), равно 2, если форма является экземпляром класса **BiColorForm**, равно 3, если она — экземпляр класса **ColorForm**, равна 0, если форма — экземпляр самого класса **Form**. Кроме того, помимо комбинационных правил, о которых уже шел разговор в черно-белом случае, для «цветной» ситуации есть свои дополнительные комбинационные правила.

Класс **Form** Протокол класса

---

**biColorForm** — возвращает число 2, указывая на двуцветную форму.

**colorForm** — возвращает число 3, указывая на цветную форму.

**exchangeColor** — возвращает число 19, указывая на комбинационное правило, по которому цвет изображения меняется на цвет фона и наоборот.

**changeColor** — возвращает число 18, указывая на комбинационное правило, по которому цвет изображения меняется на цвет фона.

**orThru** — возвращает число 17, указывая на комбинационное правило, по которому стираются (обнуляются) биты формы-цели, соответствующие единичным битам в форме-источнике, а затем применяется комбинационное правило **andRule** (**under**).

---

Второй подкласс класса **Form** — класс **ColorForm** не добавляет новых переменных к структуре экземпляра, определяемой его суперклассом, но в отличии от него самого и его подкласса **BiColorForm**, совершенно по-другому определяет

и использует переменную **bits**, поскольку для определения цвета пиксела одного бита уже не достаточно.

Создать экземпляр класса **ColorForm** позволяют следующие сообщения.

Класс **ColorForm** Протокол класса

---

**new** — возвращает новый экземпляр класса **ColorForm** с единственной инициализированной числом 3 переменной экземпляра **deviceType**.

**color: aColor** — возвращает форму-маску заданного цвета **aColor**.

---

Изменить нужным образом структуру экземпляра этого класса можно с помощью следующих сообщений.

Класс **ColorForm** Протокол экземпляра

---

**width: wInteger height: hInteger initialByte: aByte** — изменить ширину и высоту формы-приемника в соответствии с заданными аргументами **wInteger** и **hInteger**, инициализировать каждый его байт заданным значением **aByte**.

**width: wInteger height: hInteger initialColor: aColor** — изменить ширину и высоту формы-приемника в соответствии с заданными аргументами **wInteger** и **hInteger**, установить заданный аргументом **aColor** цвет формы.

---

Приведем примеры использования цветных форм и форм-масок.

**Пример 5.1.** Прямоугольник размером 100 x 100 пикселов с началом в точке 0@0 экрана, левая половина которого окрашена в цвет фона (4 — красный), а правая — в цвет символа (1 — синий). Почему так получилось? Объясним это, прокомментировав в порядке выполнения все действия, происходящие при выполнении выражения примера

```
|f|
f := (Form new width: 100 height: 100 initialByte: 16rFF).
f displayOn: f
    at: 0@0
    clippingBox: (0 @ 0 corner: 50 @ 100)
    rule: Form over
    mask: (Form black).
f displayOn: Display
    at: 0@0
```

```
clippingBox: (0 @ 0 corner: 100 @ 100)
rule: Form over
mask: (BiColorForm foreColor: 1 backColor: 4).
```

Итак, по порядку

1. f:= Form new width: 100 height: 100 initialByte: 16rFF — создается белая форма f размером 100 x 100 пикселов;

2. f displayOn: f at: 0@0 clippingBox: (0@0 corner: 50@100) rule: Form over mask: (Form black) — форма f копируется сама на себя, создавая в f экземпляр класса Form с левой половиной черного цвета и правой половиной белого цвета, что определяется наличием отсекающего прямоугольника и черной формы-маски.

3. f displayOn: Display at: 0@0 clippingBox: (0@0 corner: 100@100) rule: Form over mask: (BiColorForm foreColor: 1 backColor: 4) — форма f выводиться на экран, при этом переопределяется форма-маска, которая становится двуцветной формой с красным цветом фона и синим цветом символа, как результат на экране возникает прямоугольник черные пикселы которого окрашиваются красным цветом фона, а белые — синим цветом символа.

**Пример 5.2.** Тоже двухцветный прямоугольник размером 100 x 100 пикселов с началом в точке 0@0 экрана, только не из двух разноцветных половин, а «полосатый», состоящий из полосок шириной в 4 пиксела, одна полоса — цвета символа, там где в битовой карте формы стоят единицы, а другая — цвета фона, там где в битовой карте формы стоят нули.

```
|f|
f := (Form new width: 100 height: 100 initialByte: 16rF0).
f displayOn: Display
  at: 0@0
  clippingBox: (0 @ 0 corner: 100 @ 100)
  rule: Form over
  mask: (BiColorForm foreColor: 1 backColor: 4)
```

**Пример 5.3.** После выполнения следующих выражений, на экране появиться прямоугольник размером 100 x 100 пикселов, с началом в точке 0@0 экрана, полностью окрашенный в красный цвет.

```
|f|
f := (Form new width: 100 height: 100 initialByte: 16rFF).
f displayOn: Display
```

```
  at: 0@0
  clippingBox: Display boundingBox
  rule: Form over
  mask: (BiColorForm color: 4).
```

**Пример 5.4.** Вот еще один однотонный прямоугольник размером 100 x 100 пикселов, с началом в точке 0@0 экрана, полностью окрашенный в цвет, который определяется в процессе «смешивания» цветов foreColor и backColor, происходящего из-за присутствия формы-маски, состоящей из черных и белых пикселов.

```
|f|
f := (ColorForm width: 100 height: 100) foreColor: 5 backColor: 12.
f displayOn: Display
  at: 0@0
  clippingBox: (0 @ 0 corner: 100 @ 100)
  rule: Form over
  mask: (Form lightGray)
```

**Пример 5.5.** Прямоугольник размером 100 x 100 пикселов, отображается на экран в точке 200 @ 20, при этом то, что расположено в квадрате 20 @ 20 extent: 60 @ 60, будет отображать предыдущее содержимое экрана, а вокруг этого квадрата образуется желтая (цвета 14) непрозрачная рамка шириной в 20 пикселов.

```
|f g|
f := (BiColorForm width: 100 height: 100) foreColor: 14.
g := (Form new width: 100 height: 100 initialByte: 16rFF).
g displayOn: f
  at: 0@0
  clippingBox: (20 @ 20 extent: 60 @ 60)
  rule: Form over
  mask: (Form black).
f displayOn: Display
  at: 200 @ 20
  clippingBox: Display boundingBox
  rule: Form over
  mask: (Form white).
```

Этот пример похож на пример 5.1, только сначала черным цветом закрашивается не левая часть, а указанная внутренность прямоугольника. Эффект прозрачности черной части формы-источника f обеспечивается применяемым комбинационным правилом Form orThru, согласно которому в форме-цели изменяется только то, что при наложении попадает под белый бит (то есть под рамку).

Если применить правило Form over, то на экране возникнет просто сама форма f — черный квадрат с желтой рамкой вокруг него.

**Пример 5.6.** После выполнения выражений следующего примера, в квадрате экрана 300 x 300 с началом в точке 0@0 произойдет изменение всех цветов в соответствии с заданным цветом (10) и с предыдущим состоянием этой области экрана (из-за правила Form orRule). Кроме того, область экрана будет «слегка затенена» маской Form lightGray. Меняя первоначальный цвет формы f, меняя комбинационное правило, можно добиваться разнообразных эффектов.

```
|f|
f := (ColorForm new) width: 300 height: 300 initialColor: 10.
f displayOn: Display
    at: 0@0
    clippingBox: Display boundingBox
    rule: Form orRule
    mask: (Form lightGray).
```

# Список литературы


[1] Goldberg A., Robson D., **Smalltalk-80.** *The language.* — Addison-Wesley Publishing Company, 1988.

[2] Буч Г. *Объектно-ориентированное проектирование с примерами применения.* — М.: Конкорд, 1992.

[3] *Смолток. Объектно-ориентированная система программирования. Руководство пользователя, часть 1, 2, 3* — М.: Ин-т проблем информатики РАН, 1995

[4] Иванов А., Кремер Ю., *Язык Smalltalk: концепция объектно-ориентированного программирования* // КомпьютерПресс — 1992, № 4 — С. 21–31

[5] Фути К., Судзуки Н., *Языки программирования и схемотехника СБИС*: пер. с япон. — М.: «Мир», 1988

[6] Кирютенко Ю.А., Савельев В.А., *Объектно-ориентированное программирование и язык Smalltalk. Общие концепции и синтаксис.* — Ростов-на-Дону: УПЛ РГУ, 1995

[7] Кирютенко Ю.А., Савельев В.А., *Объектно-ориентированное программирование и язык Smalltalk. Интерфейс пользователя и среда программирования.* — Ростов-на-Дону: УПЛ РГУ, 1995

[8] Кирютенко Ю.А., Савельев В.А., *Объектно-ориентированное программирование и язык Smalltalk. Класс Collection и его подклассы. Часть 1, 2.* — Ростов-на-Дону: УПЛ РГУ, 1996

[9] Кирютенко Ю.А., Савельев В.А., *Объектно-ориентированное программирование и язык Smalltalk. Класс Magnitude и его подклассы.* — Ростов-на-Дону: УПЛ РГУ, 1997

[10] Кирютенко Ю.А., Савельев В.А., *Объектно-ориентированное программирование и язык Smalltalk. Протокол поддержки всех объектов системы.* — Ростов-на-Дону: УПЛ РГУ, 1997

[11] Кирютенко Ю.А., Савельев В.А., *Объектно-ориентированное программирование и язык Smalltalk. Протокол поддержки классов.* — Ростов-на-Дону: УПЛ РГУ, 1997

# Список иллюстраций

1 Модель для определения числа, соответствующего правилу комбинирования, и правило “ИЛИ”. . . . . 21
2 Правило “ПОД” и “ПОВЕРХ”. . . . . 23
3 Правила “ИНВЕРСИЯ” и “УДАЛЕНИЕ”. . . . . 23

# Содержание

**1 Классы поддержки графики 3**
1.1 Класс Point (Точка) . . . . . 3
1.2 Класс Rectangle (Прямоугольник) . . . . . 7

**2 Класс Form (Форма) 13**

**3 Вывод формы на экран и принтер 18**
3.1 Правила комбинирования . . . . . 21
3.2 Отсекающий прямоугольник . . . . . 24
3.3 Полностью контролируемый вывод на экран . . . . . 24

**4 Класс DisplayMedium 26**

**5 Использование цвета 28**