# Глава 7. Объектно-ориентированное программирование
## Язык Python

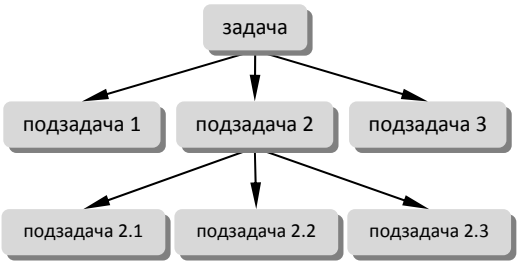
## § 46. Что такое ООП?

Как вы знаете, работа первых компьютеров сводилась к вычислениям по заданным формулам различной сложности. Число переменных и массивов в программе было невелико, так что программист мог легко удерживать в памяти все взаимосвязи между ними и детали алгоритма.

С каждым годом производительность компьютеров росла, и человек «поручал» им все более и более трудоёмкие задачи. Компьютеры следующих поколений стали использоваться для создания сложных информационных систем (например, банковских) и моделирования процессов, происходящих в реальном мире. Новые задачи требовали более сложных алгоритмов, объем программ вырос до сотен тысяч и даже миллионов строк, число переменных и массивов измерялось в тысячах.

Программисты столкнулись с проблемой сложности, которая превысила возможности человеческого разума. Один человек уже не способен написать надёжно работающую серьезную программу, так как не может «охватить взором» все её детали. Поэтому в разработке большинства современных программ принимает участие множество специалистов. При этом возникает новая проблема – нужно разделить работу между ними так, чтобы каждый мог работать независимо от других, а потом готовую программу можно было бы собрать вместе из готовых блоков, как из кубиков.

Как отмечал известный нидерландский программист Эдсгер Дейкстра, человечество еще в древности придумало способ управления сложными системами: «разделяй и властвуй». Это означает, что исходную систему нужно разбить на подсистемы (выполнить *декомпозицию*) так, чтобы работу каждой из них можно было рассматривать и совершенствовать независимо от других.




Для этого в классическом (процедурном) программировании используют метод проектирования «сверху вниз»: сложная задача разбивается на части (подзадачи и соответствующие им *алгоритмы*), которые затем снова разбиваются на более мелкие подзадачи и т.д. Однако при этом задачу «реального мира» приходится переформулировать, представляя все данные в виде переменных, массивов, списков и других структур данных. При моделировании больших систем объем этих данных увеличивается, они становятся плохо управляемыми, и это приводит к большому числу ошибок. Так как любой алгоритм может обратиться к любым глобальным (общедоступным) данным, повышается риск случайного недопустимого изменения каких-то значений.

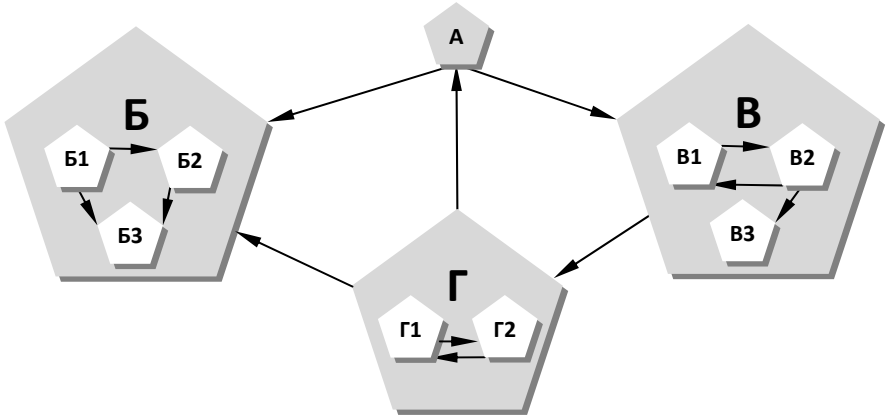
В конце 60-х годов XX века появилась новая идея – применить в разработке программ тот подход, который использует человек в повседневной жизни. Люди воспринимают мир как множество *объектов* – предметов, животных, людей – это отмечал еще в XVII веке французский математик и философ Рене Декарт. Все объекты имеют внутреннее устройство и состояние, свойства (внешние характеристики) и поведение. Чтобы справиться со сложностью окружающего мира, люди часто игнорируют многие свойства объектов, ограничиваясь лишь теми, которые необходимы для решения их практических задач. Такой прием называется *абстракцией*.

**Абстракция** – это выделение существенных характеристик объекта, отличающих его от других объектов.

Для разных задач существенные свойства могут быть совершенно разные. Например, услышав слово «кошка», многие подумают о пушистом усатом животном, которое мурлыкает, когда его гладят. В то же время ветеринарный врач представляет скелет, ткани и внутренние органы

кошки, которую ему нужно лечить. В каждом из этих случаев применение абстракции дает разные модели одного и того же объекта, поскольку различны цели моделирования.

Как применить принцип абстракции в программировании? Поскольку формулировка задач, решаемых на компьютерах, все более приближается к формулировкам реальных жизненных задач, возникла такая идея: представить программу в виде множества объектов (моделей), каждый из которых обладает своими свойствами и поведением, но его внутреннее устройство скрыто от других объектов. Тогда решение задачи сводится к моделированию взаимодействия этих объектов. Построенная таким образом модель задачи называется *объектной*. Здесь тоже идет проектирование «сверху вниз», только не по алгоритмам (как в процедурном программировании), а *по объектам*. Если нарисовать схему такой декомпозиции, она представляет собой граф, так как каждый объект может обмениваться данными со всеми другими:




Здесь А, Б, В и Г – объекты «верхнего уровня»; Б1, Б2 и Б3 – подобъекты объекта Б и т.д.

Для решения задачи «на верхнем уровне» достаточно определить, *что* делает тот или иной объект, не заботясь о том, *как* именно он это делает. Таким образом, для преодоления сложности мы используем *абстракцию*, то есть сознательно отбрасываем второстепенные детали.

Если построена объектная модель задачи (выделены объекты и определены правила обмена данными между ними), можно поручить разработку каждого из объектов отдельному программисту (или группе), которые должны написать соответствующую часть программы, то есть определить, *как именно* объект выполняет свои функции. При этом конкретному разработчику не обязательно держать в голове полную информацию обо всех объектах, нужно лишь строго соблюдать соглашения о способе обмена данными (*интерфейсе*) «своего» объекта с другими.

Программирование, основанное на моделировании задачи реального мира как множества взаимодействующих объектов, принято называть объектно-ориентированным программированием (ООП). Более строгое определение мы дадим немного позже.

### ? *Контрольные вопросы*

1. Почему со временем неизбежно изменяются методы программирования?
2. Что такое декомпозиция, зачем она применяется?
3. Что такое процедурное программирование? Какой вид декомпозиции в нём используется?
4. Какие проблемы в программировании привели к появлению ООП?
5. Что такое абстракция? Зачем она используется в обычной жизни?
6. Объясните, как связана абстракция с моделированием.
7. Какой вид декомпозиции используется в ООП?
8. Какие преимущества дает объектный подход в программировании?
9. Что такое интерфейс? Приведите примеры объектов, у которых одинаковый интерфейс и разное устройство.

## § 47. Объекты и классы

Как мы увидели в предыдущем параграфе, для того, чтобы построить объектную модель, нужно

- выделить взаимодействующие объекты, с помощью которых можно достаточно полно описать поведение моделируемой системы;
- определить их *свойства*, существенные в данной задаче;
- описать *поведение* (возможные действия) объектов, то есть команды, которые объекты могут выполнить.

Этап разработки модели, на котором решаются перечисленные выше задачи, называется *объектно-ориентированным анализом* (ООА). Он выполняется до того, как программисты напишут самую первую строчку кода, и во многом определяет качество и надежность будущей программы.

Рассмотрим объектно-ориентированный анализ на примере простой задачи. Пусть нам необходимо изучить движение автомобилей на шоссе, например, для того, чтобы определить, достаточна ли его пропускная способность. Как построить объектную модель этой задачи? Прежде всего, нужно разобраться, что такое объект.

**Объектом** можно назвать то, что имеет четкие границы и обладает *состоянием* и *поведением*.

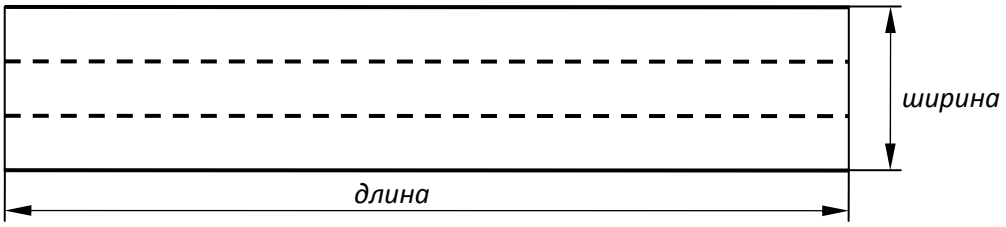
Состояние объекта определяет его возможное поведение. Например, лежачий человек не может прыгнуть, а незаряженное ружье не выстрелит.

В нашей задаче объекты – это дорога и двигающиеся по ней машины. Машин может быть несколько, причем все они с точки зрения нашей задачи имеют общие свойства. Поэтому нет смысла описывать отдельно каждую машину: достаточно один раз определить их общие черты, а потом просто сказать, что все машины ими обладают. В ООП для этой цели вводится специальный термин – *класс*.

**Класс** – это множество объектов, имеющих общую структуру и общее поведение.

Например, в рассматриваемой задаче можно ввести два класса – *Дорога* и *Машина*. По условию дорога одна, а машин может быть много.

Будем рассматривать прямой отрезок дороги, в этом случае объект «дорога» имеет два свойства, важных для нашей задачи: длину и число полос движения. Эти свойства определяют *состояние* дороги. «*Поведение*» дороги может заключаться в том, что число полос уменьшается, например, из-за ремонта покрытия, но в нашей простейшей модели объект «дорога» не будет изменяться.




Схематично класс *Дорога* можно изобразить в виде прямоугольника с тремя секциями: в верхней записывают название *класса*, во второй части – свойства, а в третьей – возможные действия, которые называют *методами*. В нашей модели дороги два свойства и ни одного метода.

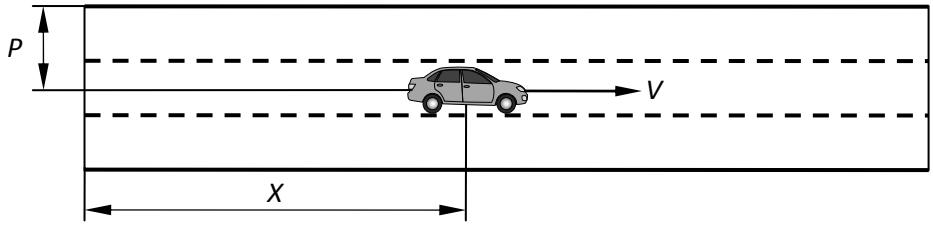
| ***Дорога*** |
|---|
| длина<br>ширина |
| |

Теперь рассмотрим объекты класса *Машина*. Их важнейшие свойства – координаты и скорость движения. Для упрощения будем считать, что

- все машины одинаковы;
- все машины движутся по дороге слева направо с постоянной скоростью (скорости разных машин могут быть различны);
- по каждой полосе движения едет только одна машина, так что можно не учитывать обгон и переход на другую полосу;
- если машина выходит за правую границу дороги, вместо нее слева на той же полосе появляется новая машина.

Не все эти допущения выглядят естественно, но такая простая модель позволит понять основные принципы метода.

За координаты машины можно принять расстояние **X** от левого края рассматриваемого участка шоссе и номер полосы **P** (натуральное число). Скорость автомобиля **V** в нашей модели – неотрицательная величина.



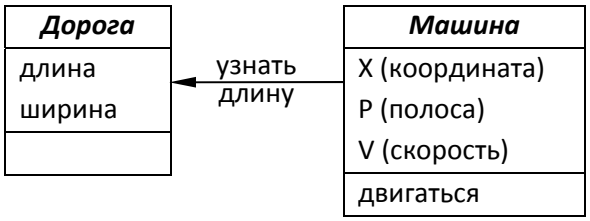

Теперь рассмотрим *поведение* машины. В данной модели она может выполнять всего одну команду – ехать в заданном направлении (назовём её «двигаться»). Говорят, что объекты класса *Машина* имеют метод «двигаться».

| ***Машина*** |
|---|
| X (координата)<br>P(полоса)<br>V (скорость) |
| двигаться |

**Метод** – это процедура или функция, принадлежащая классу объектов.

Другими словами, метод – это некоторое действие, которое могут выполнять все объекты класса.

Пока мы построили только модели отдельных объектов (точнее, классов). Чтобы моделировать всю систему, нужно разобраться, как эти объекты взаимодействуют. Объект-машина должен уметь «определить», что закончился рассматриваемый участок дороги. Для этого машина должна обращаться к объекту «дорога», запрашивая длину дороги (см. стрелку на схеме).




Такая схема определяет

- свойства объектов;
- операции, которые они могут выполнять;
- связи (обмен данными) между объектами.

В то же время мы пока ничего не говорили о том, *как* устроены объекты и *как* именно они будут выполнять эти операции. Согласно принципам ООП, ни один объект не должен зависеть от внутреннего устройства и алгоритмов работы других объектов. Поэтому, построив такую схему, можно поручить разработку двух классов объектов двум программистам, каждый из которых может решать свою задачу независимо от других. Важно только, чтобы все они четко соблюдали *интерфейс* – правила, описывающие взаимодействие «своих» объектов с остальными.

## ? *Контрольные вопросы*

1. Какие этапы входят в объектно-ориентированный анализ?
2. Что такое объект?
3. Что такое класс? Чем отличаются понятия «класс» и «объект»?
4. Что такое метод?
5. Как изображаются классы на диаграмме?
6. Почему при объектно-ориентированном анализе не уточняют, как именно объекты будут устроены и как они будут решать свои задачи?

## *Задачи*

1. Подумайте, какими свойствами и методами могли бы обладать следующие объекты: Ученик, Учитель, Школа, Экзамен, Турнир, Урок, Страна, Браузер. Придумайте свои классы объектов и выполните их анализ.

2. Добавьте в рассмотренную модель светофоры (на дороге их может быть много). Подумайте, какие свойства и методы должны быть у объектов класса *Светофор*. Как могут быть связаны классы *Дорога*, *Светофор* и *Машина* (сравните разные варианты)?
3. Придумайте свою задачу и выполните её объектно-ориентированный анализ. Примеры: моделирование работы магазина, банка, библиотеки и т.п.

# § 48. Создание объектов в программе

## Класс Дорога

Объектно-ориентированная программа начинается с описания *классов* объектов. Класс в программе – это новый тип данных. Как и структура (см. главу 6), класс – это составной тип данных, который может объединять переменные различного типа в единый блок. Кроме того, класс обычно содержит не только данные, но и методы работы с ними (процедуры и функции).

В нашей программе самый простой класс – это *Дорога* (англ. *road*). Объекты этого класса имеют два свойства (в Python они называются *атрибутами*): длину (англ. *length*), которая может быть вещественным числом, и ширину (англ. *width*) – количество полос, целое число. Для хранения значений свойств используются переменные, принадлежащие объекту, которые называются *полями*.

**Поле** – это переменная, принадлежащая объекту.

Значения полей описывают *состояние* объекта (а методы – его *поведение*).

Простейшее описание класса *Дорога* в программе на Python выглядит так:

```
class TRoad:
  pass
```

Эти строчки вводят новый тип данных – класс **TRoad**[1], то есть сообщают компилятору, что в программе, возможно, будут использоваться объекты этого типа. При этом в памяти не создается ни одного объекта. Это описание – как чертёж, по которому в нужный момент можно построить сколько угодно таких объектов.

Чтобы создать объект класса **TRoad**, нужно вызвать специальную функцию без параметров, имя которой совпадает с именем класса:

```
road = TRoad()
```

Созданный объект относится к классу **TRoad**, поэтому его называют *экземпляром* класса **TRoad**.

При описании класса мы ничего не говорили о функции, которую только что вызвали. Она добавляется транслятором ко всем классам автоматически и называется *конструктором*.

**Конструктор** – это метод класса, который вызывается для создания объекта этого класса.

У каждого объекта класс **TRoad** должно быть два поля (длина и ширина), которые мы сразу заполним нулями. Для этого нужно определить свой конструктор – метод класса с именем **__init__** (от англ. *initialization* – начальные установки).

```
class TRoad:
  def __init__( self ):      # конструктор
    self.length = 0
    self.width = 0
```

Первый параметр конструктора (как и любого метода класса) в Python – это ссылка на сам объект, который создаётся. По традиции он всегда называется **self** (от англ. *self* – «сам»). С помощью этой ссылки мы обращаемся к полям объекта, например, **self.length** означает «поле **length** текущего объекта». Если вместо этого написать просто **length**, транслятор будет считать, что речь идет о локальной или глобальной переменной, а не о поле объекта. В этом конструкторе создаются и заполняются нулями два поля (*атрибута*) объекта – длина **length** и ширина **width**.

Свойства дороги можно изменить с помощью точечной записи, с которой вы познакомились, работая со структурами:

[1] Буква T в начале названия класса – это сокращение от слова *type*.

```
road.length = 60
road.width = 3
```

Полная программа, которая создает объект «дорога» (и больше ничего не делает) выглядит так:

```
class TRoad:
  def __init__( self ):       # конструктор
    self.length = 0
    self.width = 0
road = TRoad()
road.length = 60
road.width = 3
```

Начальные значения полей можно задавать прямо при создании объекта. Для этого нужно изменить конструктор, добавив два параметра – начальные значения длины и ширины дороги:

```
class TRoad:
  def __init__( self, length0, width0 ):    # конструктор
    if length0 > 0:
      self.length = length0
    else:
      self.length = 0
    if width0 > 0:
      self.width = width0
    else:
      self.width = 0
```

В этом конструкторе проверяется правильность переданных параметров, чтобы по ошибке длина и ширина дороги не оказались отрицательными. Теперь создавать объект будет проще:

```
road = TRoad ( 60, 3 )
```

Длина этой дороги – 60 единиц, она содержит 3 полосы. Обратите внимание, что мы передали только два параметра, а не три, как указано в заголовке метода **__init__.** Параметр **self**, который указан самым первым, транслятор подставляет автоматически.

Таким образом, класс выполняет роль «фабрики», которая при вызове конструктора «выпускает» (создает) объекты «по чертежу» (описанию класса).

## Класс Машина

Теперь можно описать класс *Машина* (в программе назовём его **TCar**). Объекты класса **TCar** имеют три свойства: координата **X**, скорость **V** и номер полосы **P**:

```
class TCar:
  def __init__( self, road0, p0, v0 ): # конструктор
    self.road = road0
    self.P = p0
    self.V = v0
    self.X = 0
```

Так как объекты-машины должны обращаться к объекту «дорога», в область данных включено дополнительное поле **road**. Конечно, это не значит, что в состав машины входит дорога. Напомним, что это только ссылка, в которую сразу после создания объекта-машины нужно записать адрес заранее созданного объекта «дорога».

Для того, чтобы машина могла двигаться, нужно добавить к классу один метод – процедуру **move**:

```
class TCar:
  def __init__( self, road0, p0, v0 ): # конструктор
    self.road = road0
    self.P = p0
    self.V = v0
    self.X = 0
  def move( self ):           # движение машины
    self.X += self.V
    if self.X > self.road.length:
```

```
        self.X = 0
```

Первый параметр метода **move**, как и у любого метода класса в Python, называется **self** (ссылка на сам объект), других параметров нет, поэтому при вызове этого метода никаких данных ему передавать не нужно.

В методе **move** вычисляется новая координата **X** машины и, если она находится за пределами дороги, эта координата устанавливается в ноль (машина появляется слева на той же полосе). Изменение координаты при равномерном движении описывается формулой

$$X = X_0 + V \cdot \Delta t,$$

где $X_0$ и $X$ – начальная и конечная координаты, $V$ – скорость, а $\Delta t$ – время движения. Вспомним, что любое моделирование физических процессов на компьютере происходит в дискретном времени, с некоторым интервалом дискретизации. Для простоты мы измеряем время в этих интервалах, а за скорость $V$ принимаем расстояние, проходимое машиной за один интервал. Тогда метод **move** описывает изменение положения машины за один интервал ($\Delta t = 1$).

## Основная программа

После определения классов в основной программе создаем массив (список) объектов-машин, каждую из них «связываем» с ранее созданным объектом «*Дорога*»:

```
N = 3
cars = []
for i in range(N):
  cars.append ( TCar(road, i+1, 2*(i+1)) )
```

При вызове конструктора класса **TCar** задаются три параметра: адрес объекта «дорога» (его нужно создать до выполнения этого цикла), номер полосы и скорость. В приведенном варианте машина с номером **i** идет по полосе с номером **i+1** (считаем, что полосы нумеруются с единицы) со скоростью **2·(i+1)** единиц за один интервал моделирования.

Сам цикл моделирования получается очень простой: на каждом шаге вызывается метод **move** для каждой машины:

```
for k in range(100):    # 100 шагов
  for i in range(N):    # для каждой машины
    cars[i].move()
```

В данном случае выполнено 100 шагов моделирования. Теперь можно вывести на экран координаты всех машин:

```
print ( "После 100 шагов:" )
for i in range(N):
  print ( cars[i].X )
```

Можно ли было написать такую же программу, не использую объекты? Конечно, да. И она получилась бы короче, чем наш объектный вариант (с учетом описания классов). В чем же преимущества ООП? Мы уже отмечали, что ООП – это средство разработки больших программ, моделирующих работу сложных систем. В этом случае очень важно, что при использовании объектного подхода

- объекты реального мира проще всего описываются именно с помощью понятий «объект», «свойства», «действия» (методы);
- основная программа, описывающая решение задачи в целом, получается простой и понятной; все команды напоминают действия в реальном мире («машина № 2, вперед!»);
- разработку отдельных классов объектов можно поручить разным программистам, при этом каждый может работать независимо от других;
- если объекты *Дорога* и *Машина* понадобятся в других разработках, можно будет легко использовать уже готовые классы.

## ? *Контрольные вопросы*

1. Что такое поле?
2. Как объявляется класс объектов в программе?

3. Как в памяти создается экземпляр класса (объект)?
4. Что такое конструктор?
5. Что такое точечная запись? Как она используется при работе с объектами?
6. Какими способами можно задать начальные значения для полей объекта?
7. Сравните преимущества и недостатки решения рассмотренной задачи «классическим» способом и с помощью ООП. Сделайте выводы.

### *Задачи*

1. Добавьте в программу операторы, позволяющие изобразить на экране перемещение машин (в текстовом или графическом режиме). Подумайте, какие методы можно добавить для этого в класс **TCar**.
2. *Добавьте в модель светофор, который переключается автоматически по программе (например, 5 с горит красный свет, затем 1 с – жёлтый, потом 5 с – зеленый и т.д.). Измените классы так, чтобы машина запрашивала у объекта *Дорога* местоположение ближайшего светофора, а затем обращалась к светофору для того, чтобы узнать, какой сигнал горит. Машины должны останавливаться у светофора с запрещающим сигналом.

## § 49. Скрытие внутреннего устройства

Во время построения объектной модели задачи мы выделили отдельные объекты, которые для обмена данными друг с другом используют *интерфейс* – внешние свойства и методы. При этом все внутренние данные и детали внутреннего устройства объекта должны быть скрыты от «внешнего мира». Такой подход позволяет

- обезопасить внутренние данные (поля) объекта от изменений (возможно, разрушительных) со стороны других объектов;
- проверять данные, поступающие от других объектов, на корректность, тем самым повышая надежность программы;
- переделывать внутреннюю структуру и код объекта любым способом, не меняя его внешние характеристики (интерфейс); при этом никакой переделки других объектов не требуется.

Скрытие внутреннего устройства объектов называют **инкапсуляцией** («помещение в капсулу»). Инкапсуляцией также называют объединение данных и методов работы с ними в одном объекте.

Разберем простой пример. Во многих системах программирования есть класс, описывающий свойства «пера», которое используется при рисовании линий в графическом режиме. Назовем этот класс **TPen**, в простейшем варианте он будет содержать только одно поле **color**, которое определяет цвет. Будем хранить код цвета в виде символьной строки, в которой записан шестнадцатеричный код составляющих модели RGB. Например, **"FF00FF"** – это фиолетовый цвет, потому что красная (R) и синяя (B) составляющие равны $FF_{16}$ = 255, а зелёной составляющей нет вообще. Класс можно объявить так:

```
class TPen:
  def __init__( self ):
    self.color = "000000"
```

По умолчанию в Python все члены класса (поля и методы) открытые, общедоступные (англ. *public*). Имена тех элементов, которые нужно скрыть, должны начинаться с двух знаков подчёркивания, например, так:

```
class TPen:
  def __init__( self ):
    self.__color = "000000"
```

В этом примере поле **__color** закрытое (англ. *private* – частный). К закрытым полям нельзя обратиться извне (это могут делать только методы самого объекта), поэтому теперь невозможно не только изменить внутренние данные объекта, но и просто узнать их значения. Чтобы решить эту проблему, нужно добавить к классу еще два метода: один из них будет возвращать текущее значение поля **__color**, а второй – присваивать полю новое значение. Эти методы доступа назовем **getColor** (англ. *получить Color*) и **setColor** (англ. *установить Color*):

```
class TPen:
  def __init__ ( self ):
    self.__color = "000000"
  def getColor ( self ):
    return self.__color
  def setColor ( self, newColor ):
    if len(newColor) != 6:
      self.__color = "000000"
    else:
      self.__color = newColor
```

Что же улучшилось в сравнении с первым вариантом (когда поле было открытым)? Согласно принципам ООП, внутренние поля объекта должны быть доступны *только* с помощью методов. В этом случае внутреннее представление данных может как угодно отличаться от того, как другие объекты «видят» эти данные.

В простейшем случае метод **getColor** просто возвращает значение поля (см. текст программы выше). В методе **setColor** мы можем обрабатывать ошибки, не разрешая присваивать полю недопустимые значения. Например, в нашем методе требуется, чтобы символьная строка с кодом цвета состояла из шести символов. Если это не так, в поле **__color** записывается код чёрного цвета **"000000"**.

Теперь, если **pen** – это объект класса **TPen**, то для установки и чтения его цвета нужно использовать показанные выше методы:

```
pen = TPen()
pen.setColor ( "FFFF00" )  # изменение цвета
print ( "цвет пера:", pen.getColor() ) # получение цвета
```

Итак, мы скрыли (защитили) внутренние данные, но одновременно обращение к свойствам стало выглядеть довольно неуклюже: вместо **pen.color="FFFF00"** теперь нужно писать **pen.setColor("FFFF00")**. Чтобы упростить запись, во многие объектно-ориентированные языки программирования ввели понятие *свойства* (англ. *property*), которое внешне выглядит как переменная объекта, но на самом деле при записи и чтении свойства вызываются методы объекта.

**Свойство** – это способ доступа к внутреннему состоянию объекта, имитирующий обращение к его внутренней переменной.

Свойство **color** в нашем случае можно определить так:

```
class TPen:
  def __init__ ( self ):
    self.__color = "000000"
  def __getColor ( self ):
    return self.__color
  def __setColor ( self, newColor ):
    if len(newColor) != 6:
      self.__color = "000000"
    else:
      self.__color = newColor
  color = property ( __getColor, __setColor )
```

Здесь добавили два подчеркивания в начало названий методов **getColor** и **setColor**. Поэтому они стали защищёнными (англ. *private* – частный), то есть закрыты от других объектов. Однако есть общедоступное свойство (англ. *property*) с названием **color**:

```
color = property ( __getColor, __setColor )
```

При чтении этого свойства вызывается метод **__getColor**, а при записи нового значения – метод **__setColor**. В программе можно использовать это свойство так:

```
pen.color = "FFFF00"  # изменение цвета
print ( "цвет пера:", pen.color ); # получение цвета
```

Поскольку приведенная выше функция **getColor** просто возвращает значение поля **__color** и не выполняет никаких дополнительных действий, можно было вообще удалить метод **__getColor** и вместо него использовать «лямбда-функцию»:

```
class TPen:
  def __init__ ( self ):
    self.__color = "000000"
  def __setColor ( self, newColor ):
    if len(newColor) != 6:
      self.__color = "000000"
    else:
      self.__color = newColor
  color = property ( lambda x: x.__color , __setColor )
```

Эта «лямбда-функция» принимает единственный параметр-объект, который называется **x**, и возвращает его поле **__color**.

Таким образом, с помощью свойства **color** другие объекты могут изменять и читать цвет объектов класса **TPen**. Для обмена данными с «внешним миром» важно лишь то, что свойство **color** – символьного типа, и оно содержит 6-символьный код цвета. При этом внутреннее устройство объектов **TPen** может быть любым, и его можно менять как угодно. Покажем это на примере.

Хранение цвета в виде символьной строки неэкономно и неудобно, поэтому часто используют числовые коды цвета. Будем хранить код цвета в поле **__color** как целое число:

```
self.__color = 0
```

При этом необходимо поменять методы **__getColor** и **__setColor**, которые непосредственно работают с этим полем:

```
class TPen:
  def __init__ ( self ):
    self.__color = 0
  def __getColor ( self ):
    return "{:06x}".format ( self.__color )
  def __setColor ( self, newColor ):
    if len(newColor) != 6:
      self.__color = 0
    else:
      self.__color = int ( newColor, 16 )
  color = property ( __getColor, __setColor )
```

Для перевода числового кода в символьную запись используется функция **format**. Формат «**06x**» означает «вывести значение в шестнадцатеричной системе (**x**) в 6 позициях (**6**), свободные позиции слева заполнить нулями (**0**)». Для обратного преобразования применяем функцию **int**, которой передаётся второй аргумент – основание системы счисления.

В этом примере мы принципиально изменили внутреннее устройство объекта – заменили строковое поле на целочисленное. Однако другие объекты даже не «догадаются» о такой замене, потому что сохранился *интерфейс* – свойство **color** по-прежнему имеет строковый тип. Таким образом, инкапсуляция позволяет как угодно изменять внутреннее устройство объектов, не затрагивая интерфейс. При этом все остальные объекты изменять не требуется.

Иногда не нужно разрешать другим объектам менять свойство, то есть требуется сделать свойство «только для чтения» (англ. *read-only*). Пусть, например, мы строим программную модель автомобиля. Как правило, другие объекты не могут непосредственно менять его скорость, однако могут получить информацию о ней – «прочитать» значение скорости. При описании такого свойства метода записи (второй аргумент в объявлении свойства) не указывают вообще (или указывается пустое значение **None**):

```
class TCar:
  def __init__ ( self ):
    self.__v = 0
  v = property ( lambda x: x.__v )
```

Таким образом, доступ к внутренним данным объекта возможен, как правило, только с помощью методов. Применение свойств (*property*) очень удобно, потому что позволяет использовать ту же форму записи, что и при работе с общедоступной переменной объекта.

При использовании скрытия данных длина программы чаще всего увеличивается, однако мы получаем и важные преимущества. Код, связанный с объектом, разделен на две части: общедоступную часть и закрытую. Их можно сравнить с надводной и подводной частью айсберга. Объект взаимодействует с другими объектами только с помощью своих общедоступных свойств и методов (интерфейс). Поэтому при сохранении интерфейса можно как угодно менять внутреннюю структуру данных и код методов, и это никак не будет влиять на другие объекты. Подчеркнем, что все это становится действительно важно, когда разрабатывается большая программа и необходимо обеспечить ее надёжность.


## ? *Контрольные вопросы*

1. Что такое «интерфейс объекта»?
2. Что такое инкапсуляция? Каковы ее цели?
3. Чем отличаются общедоступные и скрытые данные и методы в описании классов?
4. Почему рекомендуют делать доступ к полям объекта только с помощью методов?
5. Что такое свойство? Зачем во многие языки программирования введено это понятие?
6. Почему методы доступа, которые использует свойство, делают зарытыми?
7. Зачем нужны свойства «только для чтения»? Приведите примеры.
8. Подумайте, в каких ситуациях может быть нужно свойство «только для записи» (которое нельзя прочитать)? Подумайте, как ввести такое свойство в описание класса?

## *Задачи*

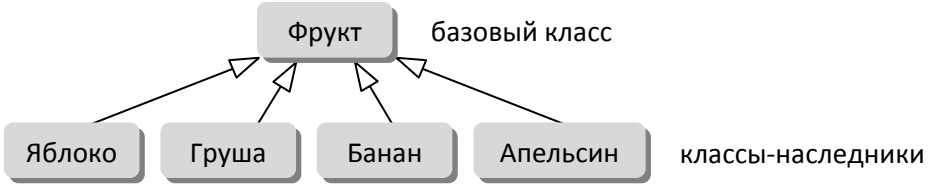
1. Измените построенную ранее программу моделирования движения так, чтобы все поля у объектов были закрытыми. Используйте свойства для доступа к данным.

# § 50. Иерархия классов

## Классификации

Как в науке, так и в быту, важную роль играет *классификация* – разделение изучаемых объектов на группы (классы), объединенные общими признаками. Прежде всего, это нужно для того, чтобы не запутаться в большом количестве данных и не описывать каждый объект заново.

Например, есть много видов фруктов[2] (яблоки, груши, бананы, апельсины и т.д.), но все они обладают некоторыми общими свойствами. Если перевести этот пример на язык ООП, класс *Яблоко* – это подкласс (производный класс, класс-наследник, потомок) класса *Фрукт*, а класс *Фрукт* – это базовый класс (суперкласс, класс-предок) для класса *Яблоко* (а также для классов *Груша*, *Банан, Апельсин* и других).




Стрелка с белым наконечником на схеме обозначает наследование. Например, класс *Яблоко* – это наследник класса *Фрукт*.

[2] Фруктами называют сочные съедобные плоды деревьев и кустарников.

Классический пример научной классификации – классификация животных или растений. Как вы знаете, она представляет собой *иерархию* (многоуровневую структуру). Например, горный клевер относится к роду *Клевер* семейства *Бобовые* класса *Двудольные* и т.д. Говоря на языке ООП, класс *Горный клевер* – это наследник класса *Клевер*, а тот, в свою очередь, наследник класса *Бобовые*, который также является наследником класса *Двудольные* и т.д.

Класс Б является наследником класса А, если можно сказать, что Б – это разновидность А.

Например, можно сказать, что яблоко – это фрукт, а горный клевер – одно из растений семейства *Двудольные*.

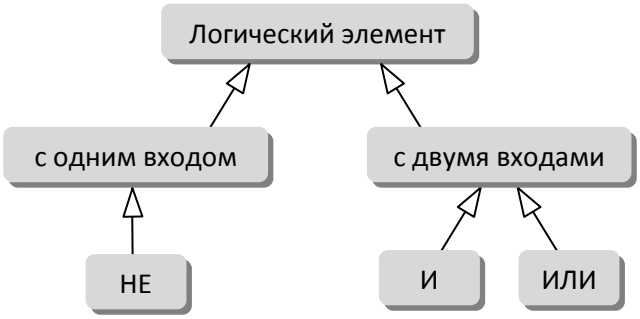
В то же время мы не можем сказать, что «машина – это разновидность двигателя», поэтому класс *Машина* не является наследником класса *Двигатель.* Двигатель – это составная часть машины, поэтому объект класса *Машина* содержит в себе объект класса *Двигатель*. Отношения между двигателем и машиной – это отношение «часть – целое»

## Иерархия логических элементов

Рассмотрим такую задачу: составить программу для моделирования управляющих схем, построенных на логических элементах (см. главу 3 в учебнике 10 класса). Нам нужно «собрать» заданную схему и построить ее таблицу истинности.

Как вы уже знаете, перед тем, как программировать, нужно выполнить объектно-ориентированный анализ. Все объекты, из которых состоит схема – это логические элементы, однако они могут быть разными («НЕ», «И», «ИЛИ» и другие). Попробуем выделить общие свойства и методы всех логических элементов.

Ограничимся только элементами, у которых один или два входа. Тогда иерархия классов может выглядеть так:




Среди всех элементов с двумя входами мы показали только элементы «И» и «ИЛИ», остальные вы можете добавить самостоятельно.

Итак, для того, чтобы не описывать несколько раз одно и то же, классы в программе должны быть построены в виде иерархии. Теперь можно дать классическое определение объектно-ориентированного программирования:

**Объектно-ориентированное программирование** – это такой подход к программированию, при котором программа представляет собой множество взаимодействующих объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

## Базовый класс

Построим первый вариант описания класса *Логический элемент* (**`TLogElement`**). Обозначим его входы как **`In1`** и **`In2`**, а выход назовем **`Res`** (от англ. *result* – результат). Здесь состояние логического элемента определяется тремя величинами (**`In1`**, **`In2`** и **`Res`**). С помощью такого базового класса можно моделировать не только статические элементы (как «НЕ», «И», «ИЛИ» и т.п.), но и элементы с памятью (например, триггеры).

| ***ЛогЭлемент*** |
|---|
| In1 (вход 1) |
| In2 (вход 2) |
| Res (результат) |
| calc |

Для сохранения значений входов и запоминания выхода введём три поля, принимающих логические значения и заполним их в конструкторе:

```
class TLogElement:
  def __init__( self ):
    self.__in1 = False
    self.__in2 = False
    self._res = False
```

Названия полей **__in1** и **__in2** начинаются с двух подчеркиваний, поэтому они будут скрытыми (доступны только внутри методов класса **TLogElement**). Имя поля **_res** начинается с одного подчёркивания, то есть оно *не* скрывается. В то же время одно подчёркивание говорит о его специальной роли, мы вернёмся к этому чуть позже.

Для доступа к полям введём свойства **In1**, **In2** и **Res** (свойство только для чтения):

```
class TLogElement:
  def __init__( self ):
    self.__in1 = False
    self.__in2 = False
    self._res = False
  def __setIn1( self, newIn1 ):
    self.__in1 = newIn1
    self.calc()
  def __setIn2( self, newIn2 ):
    self.__in2 = newIn2
    self.calc()
  In1 = property( lambda x: x.__in1, __setIn1 )
  In2 = property( lambda x: x.__in2, __setIn2 )
  Res = property( lambda x: x._res )
```

Методы чтения для всех свойств записаны как «лямбда-функции», они выполняют прямой доступ к соответствующим полям.

Вы, наверное, заметили, что оба метода записи после присваивания нового значения входу вызывают какой-то метод **calc**, которого нет в описании класса. В то же время видно, что этот метод принадлежит классу, потому что перед его именем добавлена ссылка **self**.

Метод **calc** должен пересчитать значение выхода логического элемента сразу после изменения его входа. Проблема в том, что мы не можем написать метод **calc**, пока неизвестно, какой именно логический элемент моделируется. С другой стороны, мы знаем, что такую процедуру имеет любой логический элемент. В такой ситуации можно написать метод-«заглушку» (который ничего не делает):

```
class TLogElement:
  ...
  def calc( self ):
    pass
```

Но это не совсем правильно, поскольку кто-то может создать такой элемент и пытаться его использовать, а он не работает! Поэтому не будем его определять вообще. Такой метод называется *абстрактным методом.*

**Абстрактный метод** – это метод класса, который используется, но не реализуется в классе.

Более того, *не существует* логического элемента «вообще», как не существует «просто фрукта», не относящегося к какому-то виду. Такой класс в ООП называется абстрактным. Его отличительная черта – хотя бы один абстрактный (нереализованный) метод.

**Абстрактный класс** – это класс, содержащий хотя бы один абстрактный метод.

Для надёжности нужно совсем запретить создание объектов класса **TLogElement**, потому что это бессмысленно. Для этого в конструкторе класса определим, есть ли у объекта метод **calc**, и если нет – будем искусственно воздавать аварийную ситуацию – *исключение*[3]:

```
class TLogElement:
  def __init__( self ):
    self.__in1 = False
```

[3] В Python есть и другие способы объявить класс абстрактным, например, с помощью модуля ABCMeta.

```
    self.__in2 = False
    self._res = False
    if not hasattr( self, "calc" ):
      raise NotImplementedError("Нельзя создать такой объект!")
```

Функция **hasattr** возвращает логическое значение **True**, если у переданного ему объекта (здесь – **self**) есть указанное поле или метод (**calc**). Ключевое слово **raise** означает «создать исключение», тип этого исключения – **NotImplementedError** (ошибка «не реализовано»), в скобках записана символьная строка, которую увидит пользователь в сообщении об ошибке.

Итак, полученный класс **TLogElement** – это абстрактный класс. Его можно использовать только для разработки классов-наследников, создать в программе объект этого класса нельзя. Чтобы класс-наследник не был абстрактным, он должен переопределить все абстрактные методы предка, в данном случае – метод **calc**. Как это сделать, вы увидите в следующем пункте.

Получается, что классы-наследники могут по-разному реализовать один и тот же метод. Такая возможность называется *полиморфизм*.

**Полиморфизм** (от греч. *πολυ* — много, и *μορφη* — форма) – это возможность классов-наследников по-разному реализовать метод, описанный для класса-предка.

Теперь давайте вспомним про поле с именем **_res**, которое хранит значение выхода логического элемента. Очевидно, что оно должно быть доступно классам-наследникам, которые будут изменять его в методе **calc**. Поэтому делать его скрытым нельзя. С другой стороны, часто используют соглашение о том, что одно подчёркивание означает специальное поле, которое не должно изменяться другими объектами и функциями[4].

## Классы-наследники

Теперь займемся классами-наследниками от **TLogElement**. Поскольку у нас будет единственный элемент с одним входом («НЕ»), сделаем его наследником прямо от **TLogElement** (не будем вводить специальный класс «элемент с одним входом»).

```
class TNot ( TLogElement ):
  def __init__ ( self ):
    TLogElement.__init__ ( self )
  def calc ( self ):
    self._res = not self.In1
```

После названия нового класса **TNot** в скобках указано название базового класса. Все объекты класса **TNot** обладают всеми свойствами и методами класса **TLogElement**.

В конструкторе класса-наследника нужно обязательно вручную вызвать конструктор класса-предка. В отличие от других объектно-ориентированных языков, этот вызов автоматически не выполняется.

Новый класс определяет метод **calc**, который записывает в поле **_res** инверсию входного сигнала (результат применения логической операции **not**). Таким образом, класс **TNot** уже не абстрактный, в нём все нужные методы определены. Теперь можно создавать объект этого класса **TNot** и использовать его:

```
n = TNot()
n.In1 = False
print ( n.Res )
```

Все типы логических элементов, которые имеют два входа, будут наследниками класса

```
class TLog2In ( TLogElement ):
  pass
```

[4] В других объектно-ориентированных языках программирования (C++, Паскаль) кроме общедоступных и закрытых элементов класса есть ещё защищённые (англ. *protected*). Доступ к ним имеет только сам класс, в котором они объявлены, и наследники этого класса. К сожалению, в языке Python так сделать невозможно.

В нашем случае этот класс пустой, но если нам понадобится ввести какие-то свойства и методы, характерные для всех элементов с двумя входами, мы сможем это легко сделать в классе **TLog2In**.

Класс **TLog2In** – это тоже абстрактный класс, потому что он не переопределил метод **calc**. Это сделают его наследники **TAnd** (элемент «И») и **TOr** (элемент «ИЛИ»), которые описывают конкретные логические элементы:

```
class TAnd( TLog2In ):
  def __init__( self ):
    TLog2In.__init__( self )
  def calc( self ):
    self._res = self.In1 and self.In2
class TOr( TLog2In ):
  def __init__( self ):
    TLog2In.__init__( self )
  def calc( self ):
    self._res = self.In1 or self.In2
```

Теперь мы готовы к тому, чтобы создавать и использовать построенные логические элементы. Например, таблицу истинности для последовательного соединения элементов «И» и «НЕ» можно построить так:

```
elNot = TNot()
elAnd = TAnd()
print( "  A | B | not(A&B) " );
print( "--------------------" );
for A in range(2):
  elAnd.In1 = bool(A)
  for B in range(2):
    elAnd.In2 = bool(B)
    elNot.In1 = elAnd.Res
    print( " ", A, "|", B, "|", int(elNot.Res) )
```

Сначала создаются два объекта – логические элементы «НЕ» (класс **TNot**) и «И» (класс **TAnd**). Далее в двойном цикле перебираются все возможные комбинации значений переменных **A** и **B** (каждая из них может быть равна 0 или 1). Они подаются на входы элемента «И», а его выход – на вход элемента «НЕ». Чтобы при выводе таблицы истинности вместо **False** и **True** выводились более компактные обозначения 0 и 1, значение выхода преобразуется к целому типу (**int**).

## Модульность

Как вы знаете из главы 6, большие программы обычно разбивают на модули – внутренне связные, но слабо связанные между собой блоки. Такой подход используется как в классическом программировании, так в ООП.

В нашей программе с логическими элементами в отдельный модуль (сохраним его в виде файла **logelement.py**) можно вынести всё, что относится к логическим элементам:

```
class TLogElement:
  def __init__( self ):
    self.__in1 = False
    self.__in2 = False
    self._res = False
    if not hasattr( self, "calc" ):
      raise NotImplementedError("Нельзя создать такой объект!")
  def __setIn1( self, newIn1 ):
    self.__in1 = newIn1
    self.calc()
  def __setIn2( self, newIn2 ):
    self.__in2 = newIn2
    self.calc()
  In1 = property( lambda x: x.__in1, __setIn1 )
```

```
  In2 = property ( lambda x: x.__in2, __setIn2 )
  Res = property ( lambda x: x._res )
class TNot ( TLogElement ):
  def __init__ ( self ):
    TLogElement.__init__ ( self )
  def calc ( self ):
    self._res = not self.In1
class TLog2In ( TLogElement ):
  pass
class TAnd ( TLog2In ):
  def __init__ ( self ):
    TLog2In.__init__ ( self )
  def calc ( self ):
    self._res = self.In1 and self.In2
class TOr ( TLog2In ):
  def __init__ ( self ):
    TLog2In.__init__ ( self )
  def calc ( self ):
    self._res = self.In1 or self.In2
```

Чтобы использовать такой модуль, нужно подключить его в основной программе с помощью ключевого слова **import**:

```
import logelement
elNot = logelement.TNot()
elAnd = logelement.TAnd()
...
```

Обратите внимание, что при создании объектов нужно указывать имя модуля, где определены классы **TNot** и **TAnd**.

## Сообщения между объектами

Когда логические элементы объединяются в сложную схему, желательно, чтобы передача сигналов между ними при изменении входных данных происходила автоматически. Для этого можно немного расширить базовый класс **TLogElement**, чтобы элементы могли передавать друг другу сообщения об изменении своего выхода.

Для простоты будем считать, что выход любого логического элемента может быть подключен к любому (но только одному!) входу другого логического элемента. Добавим к описанию класса два скрытых поля и один метод:

```
class TLogElement:
  def __init__ ( self ):
    ...
    self.__nextEl = None
    self.__nextIn = 0
  ...
  def link ( self, nextEl, nextIn ):
    self.__nextEl = nextEl
    self.__nextIn = nextIn
```

Поле **__nextEl** хранит ссылку на следующий логический элемент, а поле **__nextIn** – номер входа этого следующего элемента, к которому подключен выход данного элемента. С помощью общедоступного метода **link** можно связать данный элемент со следующим.

Нужно немного изменить методы **setIn1** и **setIn2**: при изменении входа они должны не только пересчитывать выход данного элемента, но и отправлять сигнал на вход следующего

```
class TLogElement:
  ...
  def __setIn1 ( self, newIn1 ):
    self.__in1 = newIn1
    self.calc()
```

```
      if self.__nextEl:
        if self.__nextIn == 1:
          self.__nextEl.In1 = self._res
        elif __nextIn == 2:
          __nextEl.In2 = self._res
```

Запись «**if __nextEl**» означает «если следующий элемент задан». Если он не был установлен, значение поля **__nextEl** будет равно **None**, и никаких дополнительных действий не выполняется.

С учетом этих изменений вывод таблицы истинности функции «И-НЕ» можно записать так (операторы вывода заменены многоточиями):

```
elNot = logelement.TNot()
elAnd = logelement.TAnd()
elAnd.link ( elNot, 1 )
...
for A in range(2):
  elAnd.In1 = bool(A)
  for B in range(2):
    elAnd.In2 = bool(B)
    ...
```

Обратите внимание, что в самом начале мы установили связь элементов «И» и «НЕ» с помощью метода **link** (связали выход элемента «И» с первым входом элемента «НЕ»). Далее в теле цикла обращения к элементу «НЕ» нет, потому что элемент «И» автоматически сообщит ему об изменении своего выхода.

## *Контрольные вопросы*

1. Что такое классификация? Зачем она нужна? Приведите примеры.
2. В каком случае можно сказать, что «класс Б – наследник класса А», а когда «объект класса А содержит объект класса Б»? Приведите примеры.
3. Что такое иерархия классов?
4. Объясните приведенную иерархию логических элементов. Обсудите ее достоинства и недостатки.
5. Дайте полное определение ООП и объясните его.
6. Что такое базовый класс и класс-наследник? Какие синонимы используются для этих терминов?
7. На примере класса **TLogElement** покажите, как выполнена инкапсуляция.
8. Что такое абстрактный класс? Почему нельзя создавать объекты этого класса?
9. Что нужно сделать, чтобы класс-наследник абстрактного класса не был абстрактным?
10. Что такое полиморфизм?
11. Какие преимущества даёт применение модулей в программе?
12. Объясните, как объекты могут передавать сообщения друг другу.
13. Подумайте, как можно организовать передачу сигнала с выхода логического элемента сразу на несколько выходов других элементов.

## *Задачи*

1. Добавьте в иерархию классов элементы «исключающее ИЛИ», «И-НЕ» и «ИЛИ-НЕ».
2. «Соберите» в программе RS-триггер из двух логических элементов «ИЛИ-НЕ», постройте его таблицу истинности (обратите внимание на вариант, когда оба входа нулевые).
3. *Используя материалы Интернета, выясните, как можно создать абстрактный класс с помощью модуля **ABCMeta**. Внесите соответствующие изменения в программу. Какой из подходов вам больше нравится? Почему?

# § 51. Программы с графическим интерфейсом

## Особенности современных прикладных программ

Большинство современных программ, предназначенных для пользователей, управляются с помощью графического интерфейса. Вы, несомненно, знакомы с понятиями «окно программы», «кнопка», «выключатель», «поле ввода», «полоса прокрутки» и т.п. Такие оконные системы чаще всего построены на принципах объектно-ориентированного программирования, то есть все элементы окон – это объекты, которые обмениваются данными, посылая друг другу сообщения.

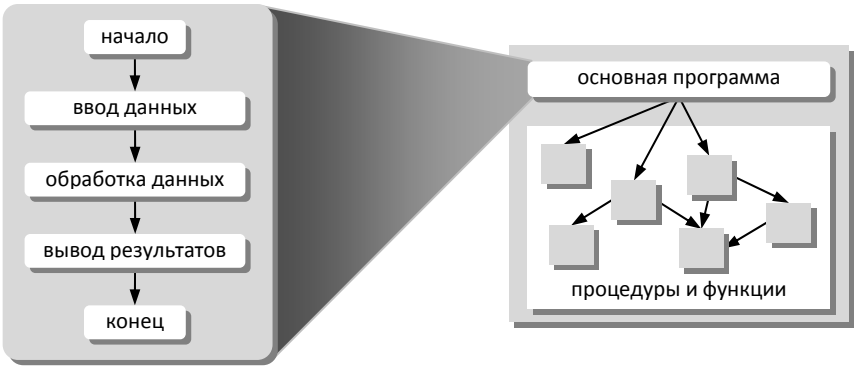
**Сообщение** – это блок данных определённой структуры, который используется для обмена информацией между объектами.

В сообщении указывается

- адресат (объект, которому посылается сообщение);
- числовой код (тип) сообщения;
- параметры (дополнительные данные), например, координаты щелчка мыши или код нажатой клавиши.

Сообщение может быть *широковещательным*, в этом случае вместо адресата указывается особый код и сообщение поступает всем объектам определенного типа (например, всем главным окнам программ).

В программах, которые мы писали раньше (см. рисунок), последовательность действия заранее определена — основная программа выполняется строчка за строчкой, вызывая процедуры и функции, все ветвления выполняются с помощью условных операторов.




В современных программах порядок действий определяется пользователем, другими программами или поступлением новых данных из внешнего источника (например, из сети), поэтому классическая схема не подходит. Пользователь текстового редактора может щелкать по любым кнопкам и выбирать любые пункты меню в произвольном порядке. Программа-сервер, передающая данные с Web-сайта на компьютер пользователя, начинает действовать только при поступлении очередного запроса. При программировании сетевых игр нужно учитывать взаимодействие многих объектов, информация о которых передается по сети в случайные моменты времени.

Во всех этих примерах программа должна «включаться в работу» только тогда, когда получит условный сигнал, то есть произойдет некоторое *событие* (изменение состояния).

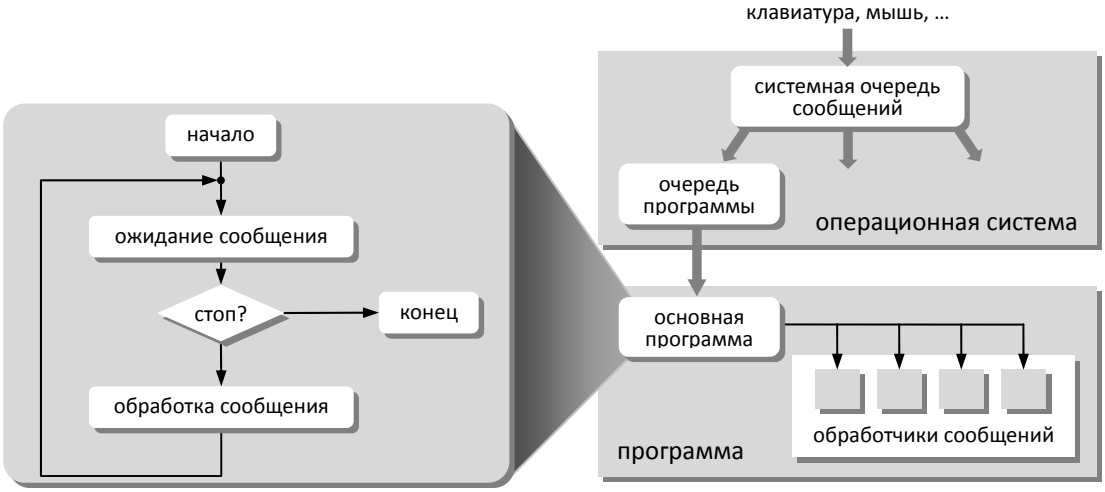
**Событие** – это переход какого-либо объекта из одного состояния в другое.

События могут быть вызваны действиями пользователя (управление клавиатурой и мышью), сигналами от внешних устройств (переход принтера в состояние готовности, получение данных из сети), получением сообщения от другой программы. При наступлении событий объекты посылают друг другу сообщения.

Таким образом, весь ход выполнения современной программы определяется происходящими событиями, а не жестко заданным алгоритмом. Поэтому говорят, что программы управляются событиями, а соответствующий стиль программирования называют *событийно-ориентированным*, то есть основанным на обработке событий.

Нормальный режим работы событийно-ориентированной программы – цикл обработки сообщений. Все сообщения (от мыши, клавиатуры и драйверов устройств ввода и вывода и т.п.) сна-

чала поступают в единую очередь сообщений операционной системы. Кроме того, для каждой программы операционная система создает отдельную очередь сообщений, и помещает в нее все сообщения, предназначенные именно этой программе.




Программа выбирает очередное сообщение из очереди и вызывает специальную процедуру – обработчик этого сообщения (если он есть). Когда пользователь закрывает окно программы, ей посылается специальное сообщение, при получении которого цикл (и работа всей программы) завершается.

Таким образом, главная задача программиста – написать содержание обработчиков всех нужных сообщений. Еще раз подчеркнем, что последовательность их вызовов точно не определена, она может быть любой в зависимости от действий пользователя и сигналов, поступающих с внешних устройств.

## RAD-среды для разработки программ

Разработка программ для оконных операционных систем до середины 1990-х годов была довольно сложным и утомительным делом. Очень много усилий уходило на то, чтобы написать команды для создания интерфейса с пользователем: разместить элементы в окне программы, написать и правильно оформить обработчики сообщений. Значительную часть своего времени программист занимался трудоёмкой работой, которая почти никак не связана с решением главной задачи. Поэтому возникла естественная мысль — автоматизировать описание окон и их элементов так, чтобы весь интерфейс программы можно было построить без ручного программирования (чаще всего с помощью мыши), а человек думал бы о сути задачи, то есть об алгоритмах обработки данных.

Такие системы программирования получили название *RAD-сред* (от англ. *Rapid Application Development* — быстрая разработка приложений). Разработка программы в RAD-системе состоит из следующих этапов:

- создание формы (так называют шаблон, по которому строится окно программы или диалога); при этом минимальный код добавляется автоматически и сразу получается работоспособная программа;
- расстановка на форме элементов интерфейса (полей ввода, кнопок, списков) с помощью мыши и настройка их свойств;
- создание обработчиков событий;
- написание алгоритмов обработки данных, которые выполняются при вызове обработчиков событий.

Обратите внимание, что при программировании в *RAD*-средах обычно говорят не об обработчиках сообщений, а об обработчиках событий. Событием в программе может быть не только нажатие клавиши или щелчок мышью, но и перемещение окна, изменение его размеров, начало и окончание выполнения расчетов и т.д. Некоторые сообщения, полученные от операционной системы, библиотека *RAD*-среды «транслирует» (переводит) в соответствующие события, а неко-

торые – нет. Более того, программист может вводить свои события и определять процедуры для их обработки.

Одной из первых сред быстрой разработки стала среда *Delphi*, разработанная фирмой *Borland* в 1994 году. Самая известная современная профессиональная RAD-система — *Microsoft Visual Studio* – поддерживает несколько языков программирования. Существует свободная *RAD*-среду *Lazarus* (*lazarus.freepascal.org*), которая во многом аналогична *Delphi*, но позволяет создавать кроссплатформенные программы (для операционных систем *Windows, Linux, Mac OS X* и др.).

Среды RAD позволили существенно сократить время разработки программ. Однако нужно помнить, что любой инструмент — это только инструмент, который можно использовать грамотно и безграмотно. Использование среды RAD само по себе не гарантирует, что у вас автоматически получится хорошая программа с хорошим пользовательским интерфейсом.

### ? *Контрольные вопросы*

1. Что такое «графический интерфейс»?
2. Как связан графический интерфейс с объектно-ориентированным подходом к программированию?
3. Что такое сообщение? Какие данные в него входят?
4. Что такое широковещательное сообщение?
5. Что такое обработчик сообщения?
6. Чем принципиально отличаются современные программы от классических?
7. Что такое событие? Какое программирование называют событийно-ориентированным?
8. Как работает событийно-ориентированная программа?
9. Какие причины сделали необходимым создание сред быстрой разработки программ? В чем их преимущество?
10. Расскажите про этапы разработки программы в RAD-среде.
11. Объясните разницу между понятиями «событие» и «сообщение».

# § 52. Графический интерфейс: основы

## Общий подход

В этом разделе мы продемонстрируем основные принципы построения программ с графическим интерфейсом на языке Python. К сожалению, для этого языка не создано сколько-нибудь надёжных средств визуальной разработки приложений (RAD).

Мы будем использовать графическую библиотеку **`tkinter`**, которая входит в стандартную библиотеку Python. Существуют также и другие библиотеки для разработки интерфейсов на Python: *wxPython*[5], PyGTK[6], PyQt[7] и некоторые другие. Их нужно устанавливать отдельно.

В программе с графическим интерфейсом может быть несколько окон, которые обычно называют *формами*[8]. На форме размещаются элементы графического интерфейса: поля ввода, флажки, кнопки и др., которые называются *виджетами* (англ. *widget* – украшение, элемент) или (как в большинстве аналогичных сред) *компонентами*. Каждый компонент – это объект определённого класса, у которого есть свойства и методы.

Для событий, которые происходят с компонентом, можно задать функции-обработчики. Они будут выполняться тогда, когда происходит соответствующее событие. Примеры таких событий – это изменение состояния флажка, изменение размеров формы, щелчок мышью на объекте, нажатие клавиши на клавиатуре.

Далее в этой главе мы сосредоточимся на принципах проектирования программ с графическим интерфейсом, а не на особенностях библиотеки **`tkinter`**. Для этого мы будем применять

[5] http://wxpython.org

[6] http://pygtk.org

[7] http://www.riverbankcomputing.com/software/pyqt/intro

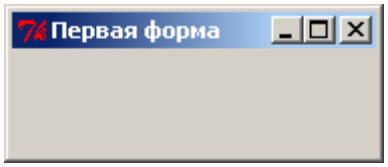
[8] В **`tkinter`** они называются окнами верхнего уровня (**`Toplevel`**).

так называемую «обёртку» **simpletk** – оболочку, которая скрывает сложности использования исходной библиотеки. Она представляет собой небольшой модуль, разработанный авторами. Этот модуль может быть свободно загружен с сайта поддержки учебника[9].

## Простейшая программа

Простейшая программа с графическим интерфейсом состоит из трёх строк:

```
from simpletk import *                    # (1)
app = TApplication("Первая форма")        # (2)
app.Run()                                 # (3)
```




Вероятно, вам понятна первая строка: из модуля **simpletk** импортируются все функции (для того, чтобы не нужно было каждый раз указывать название модуля). В строке 2 создаётся объект класса **TApplication** – это приложение (программа, от англ. *application* – приложение). Конструктору передаётся заголовок главного окна программы.

В последней строке с помощью метода **Run** (от англ. *run* – запуск) запускается цикл обработки сообщений, который скрыт внутри этого метода, так что здесь тоже использован принцип инкапсуляции (скрытия внутреннего устройства). Приведённую выше программу можно запустить, и вы должны увидеть окно, показанное на рисунке.

## Свойства формы

Объект класса **TApplication** (наследник класса **Tk** библиотеки **tkinter**) имеет методы, позволяющие управлять свойствами окна. Например, можно изменить начальное положение окна на экране с помощью свойства **position**:

```
app.position = (100, 300)
```

Позиция окна – это *кортеж*[10] (неизменяемый набор данных), состоящий из x-координаты и y-координаты левого верхнего угла окна.

Аналогично изменяются и размеры окна:

```
app.size = (500, 200)
```

Первый элемент кортежа – ширина, а второй – высота окна.

Свойство **resizable** (от англ. *изменяемый размер*) позволяет запретить изменение размеров окна по одному или обоим направлениям. Например, вызов

```
app.resizable = (True, False)
```

разрешает только изменение ширины, а изменить высоту окна будет невозможно.

С помощью свойств **minsize** и **maxsize** можно установить минимальные и максимальные размеры формы, например:

```
app.minsize = (100, 200)
```

Теперь ширина формы не может быть меньше 100 пикселей, а высота – меньше 200 пикселей.

## Обработчик событий

Рассмотрим простой пример. Вы знаете, что многие программы запрашивают подтверждение, когда пользователь завершает их работу. Для этого можно использовать обработчик события «удаление окна». Он устанавливается так:

```
app.onCloseQuery = AskOnExit
```

Здесь **onCloseQuery** – название обработчика события (от англ. *on close query* – «при запросе на закрытие»). Справа записано название функции **AskOnExit**, которая вызывается при этом собы-

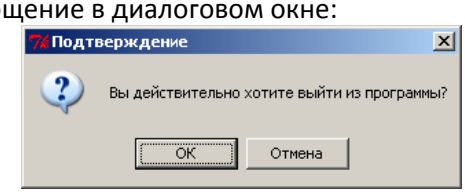
[9] http://kpolyakov.spb.ru/school/probook/python.htm.

[10] Кортежи в Python записывают в круглых скобках. С ними можно делать практически всё то же самое, что и со списками, но нельзя изменять, добавлять и удалять элементы. Однако можно построить новый кортеж, используя срезы.

тии. Эта функция определяет действия в том случае, когда пользователь закрывает окно программы, например, так:

```
def AskOnExit():
  app.destroy()
```

Всё действие этого обработчика сводится к вызову метода **destroy** (англ. разрушить): окно удаляется из памяти и работа программы завершается. Нам нужно спросить пользователя, не ошибся ли он, то есть выдать ему сообщение в диалоговом окне:




и завершить работу программы только в том случае, когда он нажмёт на кнопку «ОК».

Для этого используем готовую функцию **askokcancel** (англ. *ask* – спросить, OK – кнопка «ОК», *cancel* – кнопка «Отмена») из модуля **messagebox** пакета **tkinter**. Импортировать эту функцию можно так:

```
from tkinter.messagebox import askokcancel
```

Теперь функция-обработчик принимает следующий вид:

```
def AskOnExit():
  if askokcancel ( "Подтверждение",
     "Вы действительно хотите выйти из программы?" ):
    app.destroy()
```

Функция **askokcancel** принимает два аргумента: заголовок окна и сообщения для пользователя. Она возвращает ненулевое значение, если пользователь нажат кнопку «ОК». В этом случае срабатывает условный оператор и вызывается метод **destroy**, завершающий работу программы.

### *Контрольные вопросы*

1. Какие вы знаете средства для построения графического интерфейса в программах на Python? Какие из них входят в стандартную библиотеку языка?
2. Что такое форма?
3. В чём проявляется объектно-ориентированный подход к разработке интерфейса?
4. Что такое приложение? К какому классу библиотеки относится объект-приложение?
5. Почему в основной программе не виден цикл обработки сообщений?
6. Назовите некоторые важнейшие свойства формы. Какими способами можно их изменять?
7. Как создать обработчик события «закрытие формы»?
8. Какой модуль библиотеки **tkinter** содержит стандартные диалоги с пользователем?
9. Назовите известные вам методы объекта-приложения.

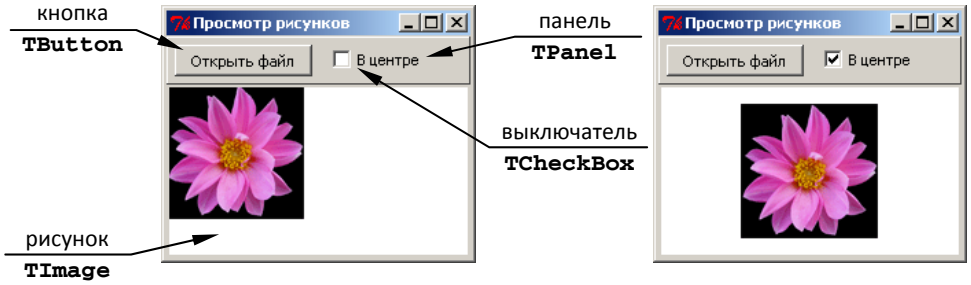
### *Задачи*

1. Постройте программу с запросом разрешения на завершение работы. Попробуйте изменять свойства главного окна.
2. *Найдите в Интернете информацию о других свойствах главного окна **Toplevel** и попробуйте их изменять.
3. *Найдите в Интернете информацию о других стандартных диалогах, входящих в модуль **messagebox**. Попробуйте их использовать.

# § 53. Использование компонентов (виджетов)

## Программа с компонентами

Теперь построим программу для просмотра файлов, которая умеет открывать файлы с диска и показывать их в нижней части окна в левом верхнем углу или по центру свободной области:




К сожалению, наша программа будет работать только с файлами формата GIF, поскольку с остальными распространёнными форматами (JPEG, PNG, BMP) библиотека **tkinter** работать не умеет.

На форме размещены четыре компонента:

- кнопка с надписью «Открыть файл» (компонент **TButton**, наследник класса **Button** библиотеки **tkinter**);
- выключатель с текстом «В центре» (компонент **TCheckBox**, наследник класса **Checkbox** библиотеки **tkinter**);
- панель, на которой лежат кнопка и выключатель (компонент **TPanel**, наследник класса **Frame** библиотеки **tkinter**);
- поле для рисунка, заполняющее все остальное место на форме (компонент **TImage**, наследник класса **Canvas** библиотеки **tkinter**).

Начнем с простой программы, которая устанавливает свойства главного окна:

```
from simpletk import *
app = TApplication ( "Просмотр рисунков" )
app.position = (200, 200)
app.size = (300, 300)
app.Run()
```

Теперь построим верхнюю панель:

```
panel = TPanel ( app, relief = "raised", height = 35, bd = 1 )
```

Для создания объекта-панели вызывается конструктор класса **TPanel**. Ему передаётся несколько параметров:

- **app** – ссылка на родительский объект;
- **relief** – объёмный вид, значение **"raised"** означает «приподнятый»; этот параметр может принимать также значения **"flat"** (плоский, по умолчанию), **"sunken"** («утопленный») и др.;
- **height** – высота в пикселях;
- **bd** (от англ. *border* – граница) – толщина границы в пикселях.

Родительский объект – это объект, отвечающий за перемещение и вывод на экран нашей панели, которая становится «дочерним» объектом для главного окна.

Расположим панель на форме, прижав её к верхней границе с помощью свойства **align**:

```
panel.align = "top"
```

Это свойство задаёт расположение панели внутри родительского окна, **"top"** означает «прижать вверх» (**"bottom"** – вниз, **"left"** – влево, **"right"** – вправо).

Теперь нужно разместить на панели компоненты **TButton** (кнопка) и **TCheckBox** (выключатель). Для них «родителем» уже будет не главное окно, а панель **panel**. Начнём с кнопки:

```
openBtn = TButton ( panel, width = 15, text = "Открыть файл" )
openBtn.position = (5, 5)
```

Для создания кнопки вызван конструктор класса **TButton**. Первый параметр – это родительский объект (панель), параметр **width** задаёт ширину кнопки в символах (не в пикселях!), а параметр **text** – надпись на кнопке. В следующей строке с помощью свойства **position** определяются координаты кнопки на панели.

Используя тот же подход, создаём и размещаем выключатель:

```
centerCb = TCheckBox ( panel, text = "В центре" )
centerCb.position = (115, 5)
```

Можно запустить программу и убедиться, что все компоненты появляются на форме, но пока не работают (не реагируют на щелчки мыши).

В нижнюю часть формы нужно «упаковать» поле для рисунка – объект класса **TImage** – так, чтобы он занимал все оставшееся место. Создаём такой объект:

```
image = TImage ( app, bg = "white" )
```

Его «родителем» будет главное окно **app**, параметр **bg** (от англ. *background* – фон) задаёт цвет «холста» (англ. *white* – белый). Затем «упаковываем» его в окно так, чтобы он занимал все оставшееся место, то есть заполнял оставшуюся область по вертикали и горизонтали:

```
image.align = "client"
```

Теперь нужно задать обработчики событий. Нас интересуют два события:

- щелчок по кнопке (после него должен появиться диалог выбора файла);
- переключение флажка-выключателя (нужно изменить режим показа рисунка).

Сначала определим, какие действия нужно выполнить в случае щелчка мышью по кнопке. Псевдокод выглядит так:

```
выбрать файл с рисунком
if файл выбран:
  загрузить рисунок в компонент image
```

Выбрать файл можно с помощью стандартного диалога операционной системы, который вызывает функция **askopenfilename** из модуля **filedialog** библиотеки **tkinter**. Сначала этот модуль нужно импортировать:

```
from tkinter import filedialog
```

Теперь вызываем функцию **askopenfilename** и передаём ей расширения, который будет показан в выпадающем списке «Тип файла»:

```
fname = filedialog.askopenfilename (
            filetypes = [ ("Файлы GIF", "*.gif"),
                          ("Все файлы", "*.*")] )
```

Параметр **filetypes** – это список, составленный из двух кортежей. Каждый кортеж содержит два элемента: текстовое объяснение и расширение имени файлов. Результат работы функции – имя выбранного файла или пустая строка, если пользователь отказался от выбора (нажал на кнопку «Отмена»). Если файл всё-таки выбран (строка не пустая), записываем имя файла в свойство **picture** объекта **TImage**:

```
if fname:
  image.picture = fname
```

При изменении этого свойства файл будет автоматически загружен и выведен на экран (это выполняет класс **TImage**). Обратите внимание, что пустая строка в Python воспринимается как ложное значение. Теперь можно составить всю функцию:

```
def selectFile ( sender ):
  fname = filedialog.askopenfilename(
              filetypes = [ ("Файлы GIF", "*.gif"),
                            ("Все файлы", "*.*")] )
  if fname:
    image.picture = fname
```

Функция-обработчик должна принимает единственный параметр **sender** – ссылку на объект, с которым произошло событие. Но в нашем случае эту ссылка не нужна, и мы её использовать не будем.

Теперь нужно установить обработчик события: сделать так, чтобы функция **selectFile** вызывалась в случае щелчка по кнопке. Свойство, определяющее обработчик события «щелчок мышью», называется **onClick** (от англ. *click* – щёлкнуть):

```
openBtn.onClick = selectFile
```

Теперь можно запустить программу и проверить загрузку файлов.

Остаётся добавить еще один обработчик, который при изменении состояния выключателя **centerCb** переключает режим вывода рисунка (в левом углу «холста» или по центру»). Напишем код такого обработчика:

```
def cbChanged ( sender ):
```

```
    image.center = sender.checked
    image.redrawImage()
```

У объекта **TImage** есть логическое свойство **center**, которое должно быть равно **True**, если нужно вывести рисунок по центру и **False**, если рисунок выводится в левом верхнем углу «холста». С другой стороны, у выключателя **TCheckBox** есть логическое свойство **checked**, которое равно **True**, если выключатель включен (флажок отмечен). В первой строке функции **cbChanged** мы установили свойство **center** в соответствии с состоянием выключателя. Затем вызывается метод **redrawImage** (от англ. *redraw image* – перерисовать рисунок), который выводит рисунок в новой позиции.

Этот обработчик нужно подключить к событию «изменение состояния выключателя», записав его адрес в свойство **onChange**:

```
centerCb.onChange = cbChanged
```

Если теперь запустить программу, мы должны увидеть правильную работу всех элементов управления. Более того, при изменении размеров окна перерисовка выполняется автоматически (за это также «отвечает» компонент **TImage**).

Отметим следующие важные особенности:

- программа целиком состоит из объектов и основана на идеях ООП;
- использование готовых компонентов скрывает от нас сложность выполняемых операций, поэтому скорость разработки программ значительно повышается.

## Новый класс: всё в одном

Доведём объектный подход до логического завершения, собрав все элементы интерфейса в новый класс **TImageViewer** – оконное приложение для просмотра рисунков. При этом основная программа будет состоять всего из двух строчек: создание главного окна и запуск программы:

```
class TImageViewer( TApplication ):
  ...
app = TImageViewer()
app.Run()
```

Вместо многоточия нужно добавить описание класса: конструктор, который создает и размещает на форме все компоненты, и обработчики событий.

Начнём с конструктора:

```
class TImageViewer( TApplication ):
  def __init__(self):
    TApplication.__init__( self, "Просмотр рисунков" )
    self.position = (200, 200)
    self.size = (300, 300)
    self.panel = TPanel(self, relief = "raised",
                            height = 35, bd = 1)
    self.panel.align = "top"
    self.image = TImage ( self, bg = "white" )
    self.image.align = "client"
    self.openBtn = TButton ( self.panel,
                          width = 15, text = "Открыть файл" )
    self.openBtn.position = (5, 5)
    self.openBtn.onClick = self.selectFile
    self.centerCb = TCheckBox ( self.panel, text = "В центре" )
    self.centerCb.position = (115, 5)
    self.centerCb.onChange = self.cbChanged
```

Сначала вызываем конструктор базового класса **TApplication** и настраиваем свойства окна. Вместо имени объекта app в предыдущей программе используем **self** – ссылку на текущий объект.

Затем строятся и размещаются компоненты, причём ссылки на них становятся полями объекта (см. «приставку» **self.** перед именами). Иначе мы не сможем обратиться к компонентам в обработчиках событий.

Обработчики событий – функции **selectFile** и **cbChanged** – тоже должны быть членами класса **TImageViewer**, поэтому их первым параметром будет **self**, а вторым – ссылка **sender** на объект, в котором произошло событие:

```
class TImageViewer( TApplication ):
  ... # здесь должен быть конструктор __init__
  def selectFile( self, sender ):
    fname = filedialog.askopenfilename(
       filetypes = [ ("Файлы GIF", "*.gif"),
                     ("Все файлы", "*.*")] )
    if fname:
      self.image.picture = fname
  def cbChanged( self, sender ):
    self.image.center = sender.checked
    self.image.redrawImage()
```

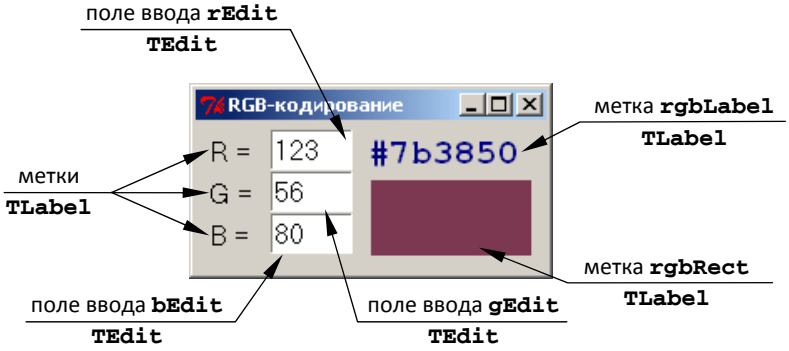
Поскольку при создании объекта в конструкторе мы запомнили адреса всех элементов в полях объекта, теперь можно к ним обращаться для изменения свойств.

В итоге получилась программа, полностью построенная на принципах объектно-ориентированного программирования: все данные и методы работы с ними объединены в классе **TImageViewer**.

## Ввод и вывод данных

Во многих программах нужно, чтобы пользователь вводил текстовую или числовую информацию. Чаще всего для этого применяют поле ввода – компонент **TEdit** (наследник класса **Entry** библиотеки **tkinter**). Для доступа к введённой строке используют его свойство **text** (англ. *текст*).

Построим программу для перевода RGB-составляющих цвета в соответствующий шестнадцатеричный код, который используется для задания цвета в языке HTML (см. главу 4).




На форме расположены

- три поля ввода (в них пользователь может задать значения красной, зелёной и синей составляющих цвета в модели RGB);
- прямоугольник (компонент **TLabel**), цвет которого изменяется согласно введенным значениям;
- несколько меток (компонентов **TLabel**).

Метки – это надписи, которые пользователь не может редактировать, однако их содержание можно изменять из программы через свойство **text**.

Во время работы программы будут использоваться поля ввода **rEdit, gEdit** и **bEdit**, метка **rgbLabel**, с помощью которой будет выводиться результат – код цвета, и метка без текста **rgbRect** – прямоугольник, закрашенный нужным цветом. В качестве начальных значений полей ввода можно ввести любые целые числа от 0 до 255 (свойство **text**).

При изменении содержимого одного из трёх полей ввода нужно обработать введённые данные и вывести результат в свойство **text** метки **rgbLabel**, а также изменить цвет фона для метки **rgbRect**. Обработчик события, которое происходит при изменении текста в поле ввода,

называется **onChange**. Так как при изменении любого из трех полей нужно выполнить одинаковые действия, для этих компонентов можно установить один и тот же обработчик.

Обработчик события **onChange** для поля ввода может выглядеть так:

```
def onChange ( sender ):
  r = int ( rEdit.text )              # (1)
  g = int ( gEdit.text )              # (2)
  b = int ( bEdit.text )              # (3)
  s = "#{:02x}{:02x}{:02x}".format(r, g, b)    # (4)
  rgbRect.background = s              # (5)
  rgbLabel.text = s                   # (6)
```

Содержимое всех полей ввода преобразуется в числа с помощью функции **int** (строки 1-3). Затем в строке 4 строится символьная строка – шестнадцатеричная запись цвета в HTML-формате. Значения красной, зелёной и синей составляющих (переменные **r**, **g** и **b**) выводятся по формату «**02x**», то есть в шестнадцатеричной записи, состоящей из двух цифр с заполнением пустых позиций нулями. В строке 5 изменяется фон метки-прямоугольника **rgbRect**, а в строке 6 код цвета заносится в метку **rgbLabel**.

В основной программе создаём объект-приложение с главным окном:

```
app = TApplication ( "RGB-кодирование" )
app.size = (210, 90)
app.position = (200, 200)
```

и метки слева от полей ввода:

```
f = ( "MS Sans Serif", 12 )
lblR = TLabel ( app, text = "R = ", font = f )
lblR.position = (5, 5)
lblG = TLabel ( app, text = "G = ", font = f )
lblG.position = (5, 30)
lblB = TLabel ( app, text = "B = ", font = f )
lblB.position = (5, 55)
```

При создании меток мы изменили шрифт с помощью параметра **font**. Значение этого параметра – кортеж **f**, который в данном случае состоит из двух элементов: названия шрифта и его размера в пунктах.

Создаём еще две метки для вывода результата:

```
fc = ( "Courier New", 16, "bold" )
rgbLabel = TLabel ( app, text = "#000000", font = fc, fg = "navy" )
rgbLabel.position = (100, 5)
rgbRect = TLabel ( app, text = "", width = 15, height = 3 )
rgbRect.position = (105, 35)
```

Размеры меток – ширина (англ. *width*) и высота (англ. *height*) указываются не в пикселях, а в текстовых единицах, то есть в размерах символа. Шрифт для метки **rgbLabel** задаётся в виде кортежа из трёх элементов: название шрифта, размер и свойство «жирный» (англ. *bold*). Параметр **fg** (от англ. *foreground* – цвет переднего плана) при вызове конструктора метки определяет цвет символов («navy» – тёмно-синий, от англ. *navy* – военно-морской).

Затем добавляем на форму три поля ввода:

```
rEdit = TEdit ( app, font = f, width = 5 )
rEdit.position = (45, 5)
rEdit.text = "123"
gEdit = TEdit ( app, font = f, width = 5 )
gEdit.position = (45, 30)
gEdit.text = "56"
bEdit = TEdit ( app, font = f, width = 5 )
bEdit.text = "80"
bEdit.position = (45, 55)
```

для каждого из них устанавливаем обработчик **onChange**, который вызывается при любом изменении текста в поле ввода:

```
rEdit.onChange = onChange
```

```
gEdit.onChange = onChange
bEdit.onChange = onChange
```

и запускаем программу:

```
app.Run()
```

Обратите внимание, что назначение обработчика событий нужно делать тогда, когда все объекты, используемые в обработчике **onChange** (поля ввода **rEdit**, **gEdit**, **bEdit** и метки **rgbLabel** и **rgbRect**) уже созданы.

При желании можно переписать программу в стиле ООП, как это мы сделали в конце предыдущего примера. Это вы уже можете сделать самостоятельно, используя образец.

## Обработка ошибок

Если в предыдущей программе пользователь введет не числа, а что-то другое (или пустую строку), программа выдаст сообщение о необработанной ошибке на английском языке и программа завершит работу. Хорошая программа никогда не должна завершаться аварийно, для этого все ошибки, которые можно предусмотреть, надо обрабатывать.

В современных языках программирования есть так называемый *механизм исключений*, который позволяет обрабатывать практически все возможные ошибки. Для этого все «опасные» участки кода (на которых может возникнуть ошибка) нужно поместить в блок **try – except**:

```
try:
  # «опасные» команды
except:
  # обработка ошибки
```

Слово **try** по-английски означает «попытаться», **except** — «исключение» (исключительная или ошибочная, непредвиденная ситуация). Программа попадает в блок **except** только тогда, когда между **try** и **except** произошла ошибка.

В нашей программе «опасные» команды – это операторы преобразования данных из текста в числа (вызовы функции **int**). В случае ошибки мы выведем вместо кода цвета знак вопроса, а цвет фона метки-прямоугольника **rgbRect** изменим на стандартное значение **"SystemButtonFace"**, которое означает «системный цвет кнопки». При этом прямоугольник становится невидимым, потому что сливается с фоновым цветом окна.

Улучшенный обработчик с защитой от неправильного ввода принимает вид:

```
def onChange( sender ):
  s = "?"
  bkColor = "SystemButtonFace"
  try:
    # получить новый цвет из полей ввода
  except:
    pass
  rgbLabel.text = s
  rgbRect.background = bkColor
```

Здесь переменная **s** обозначает шестнадцатеричный код цвета (в виде символьной строки), а переменная **bkColor** – цвет прямоугольника. Если при попытке получить новый код цвета происходит ошибка, то в этих переменных остаются значения, установленные в первых строках обработчика.

Если значения полей ввода удалось преобразовать в числа, нужно убедиться, что все составляющие цвета не меньше 0 и не больше 255, добавив проверку диапазона **range(256)** для всех значений:

```
def onChange( sender ):
  s = "?"
  bkColor = "SystemButtonFace"
  try:
    r = int( rEdit.text )
    g = int( gEdit.text )
    b = int( bEdit.text )
```

```
    if r in range(256) and \
       g in range(256) and b in range(256):
      s = "#{:02x}{:02x}{:02x}".format(r, g, b)
      bkColor = s
  except:
    pass
  rgbLabel.text = s
  rgbRect.background = bkColor
```

### *Контрольные вопросы*

1. Что такое компоненты? Зачем они нужны?
2. Объясните, как связаны компоненты и идея инкапсуляции.
3. Что такое родительский объект? Что это значит?
4. Объясните роль свойства **align** в размещении элементов на форме.
5. Что такое стандартный диалог? Как его использовать?
6. Назовите основное свойство выключателя. Как его использовать?
7. Что такое метка?
8. Как обрабатываются ошибки в современных программах? В чем, на ваш взгляд, преимущества и недостатки такого подхода?

### *Задачи*

1. Напишите программу для построения RGB-кода цвета в стиле ООП, «убрав» все действия по созданию формы в конструктор класса. Обработчики событий также должны быть членами класса.
2. Разработайте программу для перевода морских милей в километры (1 миля = 1852 м).
3. Разработайте программу для решения системы двух линейных уравнений. Обратите внимание на обработку ошибок при вычислениях.
4. Разработайте программу для перевода суммы в рублях в другие валюты.
5. Разработайте программу для перевода чисел и десятичной системы в двоичную, восьмеричную и шестнадцатеричную.
6. Разработайте программу для вычисления информационного объема рисунка по его размерам и количеству цветов в палитре.
7. Разработайте программу для вычисления информационного объема звукового файла при известных длительности звука, частоте дискретизации и глубине кодирования (числу бит на отсчёт).

## § 54. Совершенствование компонентов

Как вы видели в предыдущем параграфе, на практике нередко нужны поля ввода особого типа, с помощью которых можно вводить целые числа. Компонент **TEdit** разрешает вводить любые символы и представляет результат ввода как текстовое свойство **text**. Поэтому для того, чтобы получить нужное нам поведение (ввод целых чисел), мы

- добавили обработку ошибок в процедуру **onChange**;
- для перевода текстовой строки в число каждый раз использовали функцию **int**.

Если такие поля ввода нужны часто и в разных программах, можно избавиться от этих рутинных операций. Для этого создается новый компонент, который обладает всеми необходимыми свойствами.

Конечно, можно создавать компонент «с нуля», но так почти никто не делает. Обычно задача сводится к тому, чтобы как-то улучшить существующий стандартный компонент, который уже есть в библиотеке.

Мы будем совершенствовать компонент **TEdit** (поле ввода), поэтому, согласно принципам ООП, наш компонент (назовём его **TIntEdit**) будет наследником класса **TEdit**, а класс **TEdit** будет соответственно базовым классом для нового класса **TIntEdit**:

```
class TIntEdit ( TEdit ):
  ...
```

Изменения стандартного класса **TEdit** сводятся к двум пунктам:

- все некорректные символы, которые приводят к тому, что текст нельзя преобразовать в целое число, должны блокироваться автоматически, без установки дополнительных обработчиков событий;
- компонент должен уметь сообщать числовое значение целого типа; для этого мы добавим к нему свойство **value** (англ. *значение*).

Сначала добавим в новый класс конструктор:

```
class TIntEdit ( TEdit ):
  def __init__ ( self, parent, **kw ):
    TEdit.__init__ ( self, parent, **kw )
```

При вызове этого конструктора нужно указать, по крайней мере, один параметр – ссылку на «родительский» объект **parent**. Затем могут следовать именованные параметры (подробности можно найти в Интернете в описании компонента **Entry** библиотеки **tkinter**). Запись с двумя звёздочками ****kw** говорит о том, что они «собираются» в словарь.

В приведённом варианте класс **TIntEdit** ничем не отличается от **TEdit**. Добавим к нему свойство **value**. Для этого введём закрытое поле **__value**, в котором будем хранить последнее правильное числовое значение:

```
class TIntEdit ( TEdit ):
  def __init__ ( self, parent, **kw ):
    TEdit.__init__ ( self, parent, **kw )
    self.__value = 0
  def __setValue ( self, value ):
    self.text = str ( value )
  value = property ( lambda x: x.__value, __setValue )
```

Из предыдущего материала (см. § 49) вам должно быть понятно, что для чтения введенного числового значения используется «лямбда-функция», которая возвращает значение поля **__value**. Метод записи **__setValue** записывает в свойство **text** переданное число в виде символьной строки.

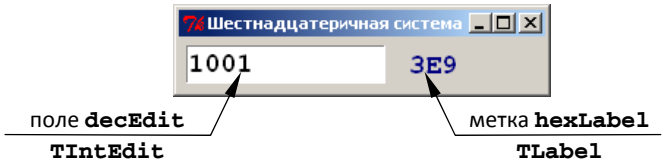
Для того, чтобы заблокировать ввод нецифровых символов, будем использовать обработчик события **onValidate** (от англ. *validate* – проверять на правильность) компонента **TEdit**. Установим этот обработчик на скрытый метод нового класса **TIntEdit**, который назовём **__onValidate**. Обработчик должен возвращать логическое значение: **True**, если символ допустимый и **False**, если нет (тогда ввод этого символа блокируется). В этом коде оставлены только те строки, который относятся к установке этого обработчика:

```
class TIntEdit ( TEdit ):
  def __init__ ( self, parent, **kw ):
    ...
    self.onValidate = self.__validate
  def __validate ( self ):
    try:
      newValue = int ( self.text )
      self.__value = newValue
      return True
    except:
      return False
```

При получении сигнала о событии, обработчик пытается преобразовать новое значение в число. Если это удаётся, полученное число записывается в переменную **__value** и возвращается результат **True**. Если произошла ошибка, функция вернёт **False**.

Готовый компонент лучше всего поместить в отдельный модуль, назовем его **int_edit.py**.

Построим новый проект: программа будет переводить целые числа из десятичной системы в шестнадцатеричную:




Импортируем компонент **TIntEdit** из модуля **int_edit**:

```
from int_edit import TIntEdit
```

Создадим объект-приложение:

```
app = TApplication ( "Шестнадцатеричная система" )
app.size = (250, 36)
app.position = (200, 200)
```

Поместим на форму метку **hexLabel** для вывода шестнадцатеричного значения и компонент класса **TIntEdit** для ввода:

```
f = ( "Courier New", 14, "bold" )
hexLabel = TLabel ( app, text = "?", font = f, fg = "navy" )
hexLabel.position = (155, 5)
decEdit = TIntEdit ( app, width = 12, font = f )
decEdit.position = (5, 5)
decEdit.text = "1001"
```

Добавим обработчик события **onChange** для поля ввода:

```
def onNumChange ( sender ):
  hexLabel.text = "{:X}".format ( sender.value )
decEdit.onChange = onNumChange
```

Этот обработчик читает значение свойства **value** объекта-источника события и выводит его на метку **hexLabel** в шестнадцатеричной системе. Формат «**X**» (а не «**x**») означает, что будут использовать *заглавные* буквы A-F.

Теперь программа готова и можно проверить её работу.

### ? *Контрольные вопросы*

1. В каких случаях имеет смысл разрабатывать свои компоненты?
2. Подумайте, в чем достоинства и недостатки использования своих компонентов?
3. Почему программисты редко создают свои компоненты «с нуля»?
4. Объясните, как связаны классы компонентов **TIntEdit** и **TEdit**. Чем они отличаются?
5. Какие функции используются для преобразования числового значения в текстовое и обратно?
6. Как получить запись числа в шестнадцатеричной системе счисления?
7. Объясните, как работает свойство **value** у компонента **TIntEdit**?
8. Почему мы не обрабатывали возможные ошибки в обработчике **onChange**?
9. Как установить обработчик события во время выполнения программы?
10. Почему можно использовать обработчик события **onChange**, который не был объявлен в классе **TIntEdit**?

### *Задачи*

1. Разработайте компонент, который позволяет вводить шестнадцатеричные числа.

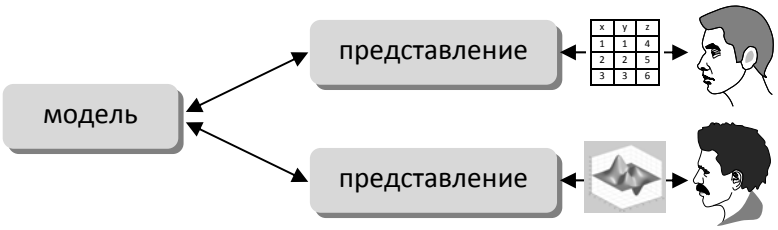
## § 55. Модель и представление

Одна из важнейших идей технологии быстрого проектирования программ (RAD) – повторное использование написанного ранее готового кода. Чтобы облегчить решение этой задачи, было предложено использовать еще одну декомпозицию: разделить *модель*, то есть данные и методы их обработки, и *представление* – способ взаимодействия модели с пользователем (интерфейс).

Пусть, например, данные об изменении курса доллара хранятся в виде массива, в котором требуется искать максимальное и минимальное значения, а также строить приближенные зави-

симости, позволяющие прогнозировать изменение курса в ближайшем будущем. Это описание задачи на *уровне модели*.

Для пользователя эти данные могут быть представлены в различных формах: в виде таблицы, графика, диаграммы и т.п. Полученные зависимости, приближенно описывающие изменение курса, могут быть показаны в виде формулы или в виде кривой. Это *уровень представления* или интерфейса с пользователем.




Чем хорошо такое разделение? Его главное преимущество состоит в том, что модель не зависит от представления, поэтому одну и ту же модель можно использовать без изменений в программах, имеющих совершенно различный интерфейс.

## Вычисление арифметических выражений: модель

Построим программу, которая вычисляет арифметическое выражение, записанное в символьной строке. Для простоты будем считать, что в выражении используются только

- целые числа и
- знаки арифметических действий **+-*/**.

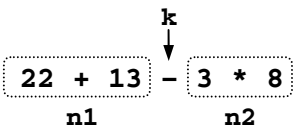
Предположим, что выражение не содержит ошибок и посторонних символов.

Какова *модель* для этой задачи? По условию данные хранятся в виде символьной строки. Обработка данных состоит в том, что нужно вычислить значение записанного в строке выражения.

Вспомните, что аналогичную задачу мы решали в главе 6, где использовалась структура типа «дерево». Теперь мы применим другой способ.

Как вы знаете, при вычислении арифметического выражения последней выполняется крайняя справа операция с наименьшим приоритетом (см. главу 6). Таким образом, можно сформулировать следующий алгоритм вычисления арифметического выражения, записанного в символьной строке **s**:

1. Найти в строке **s** последнюю операцию с наименьшим приоритетом (пусть номер этого символа записан в переменной **k**).
2. Используя дважды этот же алгоритм, вычислить выражения слева и справа от символа с номером **k** и записать результаты вычисления в переменные **n1** и **n2**.




3. Выполнить операцию, символ которой записан в **s[k]**, с переменными **n1** и **n2**.

Обратите внимание, что в п. 2 этого алгоритма нужно решить ту же самую задачу для левой и правой частей исходного выражения. Как вы знаете, такой прием называется *рекурсией*.

Основную функцию назовём **Calc** (от англ. *calculate* – вычислить). Она принимает символьную строку и возвращает целое число – результат вычисления выражения, записанного в этой строке. Алгоритм её работы на псевдокоде:

```
k = номер символа, соответствующего последней операции
if k < 0: # нет знака операции
  перевести всю строку в число
else:
  n1 = результат вычисления левой части
  n2 = результат вычисления правой части
  применить найденную операцию к n1 и n2
```

Для того, чтобы найти последнюю выполняемую операцию, будем использовать функцию **lastOp** из главы 6. Если эта функция вернула «–1», то операция не найдена, то есть вся переданная ей строка – это число (предполагается, что данные корректны).

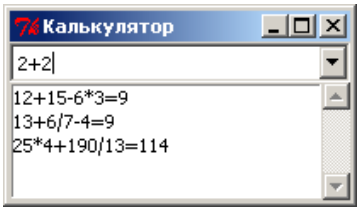
Теперь можно написать функцию **Calc**:

```
def Calc ( s ):
  k = lastOp ( s )
  if k < 0:          # вся строка - число
    return int(s)
  else:
    n1 = Calc ( s[:k] )    # левая часть
    n2 = Calc ( s[k+1:] ) # правая часть
     # выполнить операцию
    if   s[k] == "+": return n1+n2
    elif s[k] == "-": return n1-n2
    elif s[k] == "*": return n1*n2
    else: return n1 // n2
```

Обратите внимание, что функция **Calc** – рекурсивная, она дважды вызывает сама себя.

Функции **Calc** и **lastOp** (а также функцию **priority**, которая вызывается из **lastOp**) удобно объединить в отдельный модуль **model.py** (модуль модели). Таким образом, наша модель – это функции, с помощью которых вычисляется арифметическое выражение, записанное в строке.

## Вычисление арифметических выражений: представление




Теперь построим интерфейс программы. В верхней части окна будет размещен выпадающий список (компонент **TComboBox**), в котором пользователь вводит выражение. При нажатии на клавишу *Enter* выражение вычисляется и его результат выводится в первой строке обычного списка (компонента **TListBox**). Выпадающий список, в котором хранятся все вводимые выражения, полезен для того, чтобы можно было вернуться к уже введённому ранее варианту и исправить его.

Итак, импортируем из модуля **model** функцию **Calc** и создаём приложение:

```
from model import Calc
app = TApplication ( "Калькулятор" )
app.size = (200, 150)
```

На форму нужно добавить компонент **TComboBox**. Чтобы прижать его к верху, установим свойство **align**, равное **"top"**. Назовем этот компонент **Input** (англ. *ввод*).

```
Input = TComboBox ( app, values = [], height = 1 )
Input.align = "top"
Input.text = "2+2"
```

При вызове конструктора, кроме родительского объекта, мы указали два именованных параметра: **values** (англ. «значения») – пустой список (сначала в списке нет ни одного элемента) и **height** – высота компонента (одна строка).

Добавляем второй компонент – **TListBox**, устанавливаем для него выравнивание **"client"** (заполнить всю свободную область) и имя **Answers** (англ. *ответы*).

```
Answers = TListBox ( app )
Answers.align = "client"
```

Логика работы программы может быть записана в виде псевдокода:

```
if нажата клавиша Enter:
  вычислить выражение
  добавить результат вычислений в начало списка
  if выражения нет в выпадающем списке:
    добавить его в выпадающий список
```

Для перехвата нажатия клавиши *Enter* будем использовать обработчик события «**<Key-Return>**» (англ. «клавиша перевод каретки»), который подключается стандартным способом библиотеки **tkinter** – через метод **bind** (англ. «*связать*»):

```
Input.bind ( "<Key-Return>", doCalc )
```

Здесь **doCalc** – это название функции, принимающей один параметр – объект-структуру, которая содержит полную информацию о произошедшем событии:

```
def doCalc ( event ):
  ...
```

Вместо многоточия нужно написать операторы Python, которые нужно выполнить. Во-первых, читаем текст, введённый в выпадающем списке, и вычисляем выражение с помощью функции **Calc**:

```
expr = Input.text
x = Calc ( expr )
```

Затем добавляем в начало списка вычисленное выражение и его результат:

```
Answers.insert ( 0, expr + "=" + str(x) )
```

Здесь используется метод **insert** (англ. «*вставить*»), первый аргумент – это номер вставляемого элемента (0 – в начало списка).

Теперь проверим, есть ли такое выражение в выпадающем списке, и если нет – добавим его:

```
if not Input.findItem(expr):
  Input.addItem ( expr )
```

Метод **findItem** (англ. *find item* – найти элемент) возвращает логическое значение: **True**, если переданная ему строка есть в списке и **False**, если нет. Метод **addItem** (англ. *add item* – добавить элемент) добавляет элемент в конец списка.

Таким образом, полная функция **doCalc** приобретает следующий вид:

```
def doCalc ( event ):
  expr = Input.text
  x = Calc ( expr )
  Answers.insert ( 0, expr + "=" + str(x) )
  if not Input.findItem ( expr ):
    Input.addItem ( expr )
```

Теперь программу можно запускать и испытывать.

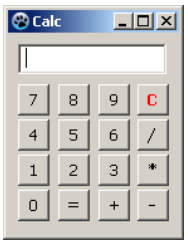
Итак, в этой программе мы разделили модель (данные и средства их обработки) и представление (взаимодействие модели с пользователем), которые разнесены по разным модулям. Это позволяет использовать модуль модели в любых программах, где нужно вычислять арифметические выражения.

Часто к паре «модель-представление» добавляют еще управляющий блок (контро́ллер), который, например, обрабатывает ошибки ввода данных. Но во многих случаях, например, при программировании в RAD-средах, контроллер и представление объединяются вместе – управление данными происходит в обработчиках событий.

## ? *Контрольные вопросы*

1. Чем хорошо разделение программы на модель и интерфейс? Как это связано с особенностями современного программирования?
2. Что обычно относят к модели, а что – к представлению?
3. Что от чего зависит (и не зависит) в паре «модель – представление»?
4. Приведите свои примеры задач, в которых можно выделить модель и представление. Покажите, что для одной модели можно придумать много разных представлений.
5. Объясните алгоритм вычисления арифметического выражения без скобок.
6. Пусть требуется изменить программу так, чтобы она обрабатывала выражения со скобками. Что нужно изменить: модель, интерфейс или и то, и другое?

## *Задачи*




1. Измените программу так, чтобы она вычисляла выражения с вещественными числами (для перевода вещественных чисел из символьного вида в числовой используйте функцию **`float`**).
2. Добавьте в программу обработку ошибок. Подумайте, какие ошибки может сделать пользователь. Какие ошибки могут возникнуть при вычислениях? Как их обработать?
3. *Измените программу так, чтобы она вычисляла выражения со скобками. *Подсказка*: нужно искать последнюю операцию с самым низким приоритетом, стоящую *вне* скобок.
4. Постройте программу «Калькулятор» для выполнения вычислений с целыми числами (см. рисунок).

## Самое важное в главе 7:

- Сложность и размеры современных программ таковы, что в их разработке принимает участие множество программистов. Объектно-ориентированное программирование – это метод, позволяющий разбить задачу на части, каждая из которых в максимальной степени независима от других.
- Программа в ООП – это набор объектов, которые обмениваются сообщениями.
- Перед программированием выполняется объектно-ориентированный анализ задачи. На этом этапе выделяются взаимодействующие объекты, определяются их существенные свойства и поведение.
- Любой объект – экземпляр какого-то класса. Классом называют группу объектов, обладающих общими свойствами.
- Объекты не могут «узнать» устройство других объектов (принцип *инкапсуляции*). При описании класса закрытые поля и методы помещаются в секцию **`private`**, а общедоступные – в секцию **`public`**.
- Обмен данными между объектами выполняется с помощью общедоступных свойств и методов, которые составляют *интерфейс* объектов. Изменение внутреннего устройства объектов (*реализации*) не влияет на взаимодействие с другими объектами, если не меняется интерфейс.
- Как правило, классы образуют иерархию (многоуровневую структуру). Классы-потомки обладают всеми свойствами и методами классов-предков, к которым добавляются их собственные свойства и методы.
- ООП позволяет обеспечивать высокую скорость и надежность разработки больших и сложных программ. В простых задачах применение ООП, как правило, увеличивает длину программы и замедляет её работу.
- Современные программы с графическим интерфейсом основаны на обработке событий, которые вызваны действиями пользователя и поступлением данных из других источников, и могут происходить в любой последовательности.
- Для быстрой разработки программ применяют системы визуального программирования, в которых интерфейс строится без «ручного» написания программного кода. Такие системы, как правило, основаны на ООП.
- В современных программах принято разделять *модель* (данные и алгоритмы их обработки) и *представление* (способ ввода исходных значений и вывода результатов).