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 $\mathbf{FF_{16}} = \mathbf{255}$, 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 )
```

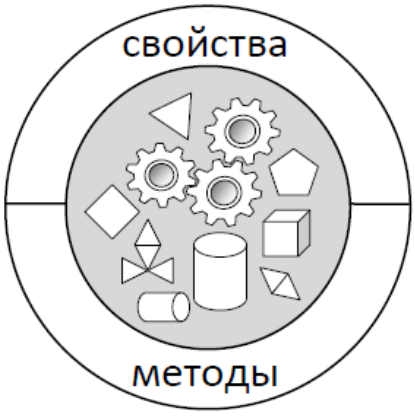
Для перевода числового кода в символьную запись используется функция **format**. Формат «**06x**» означает «вывести значение в шестнадцатеричной системе (**x**) в 6 позициях (**6**), свободные позиции слева заполнить нулями (**0**)». Для обратного преобразования применяем функцию **int**, которой передаётся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