# Глава 7. Объектно-ориентированное программирование Язык С++ (C#)

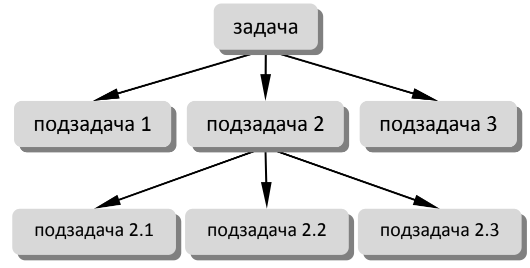
## § 46. Что такое ООП?

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.



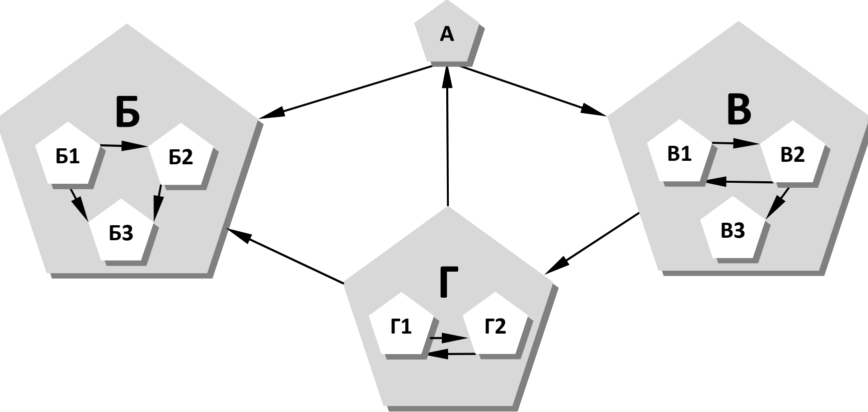

Для этого в классическом (процедурном) программировании используют метод проектирования «сверху вниз»: сложная задача разбивается на части (подзадачи и соответствующие им *алгоритмы*), которые затем снова разбиваются на более мелкие подзадачи и т.д. Однако при этом задачу «реального мира» приходится 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*.

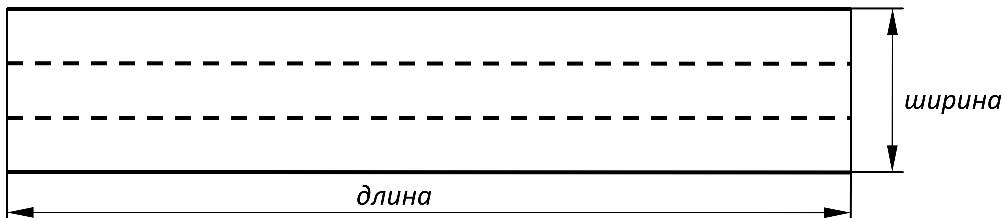
Состояние объекта определяет его возможное поведение. Например, лежачий человек не может прыгнуть, а незаряженное ружье не выстрелит.

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.

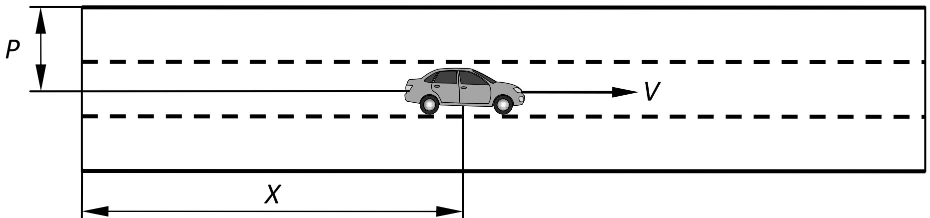
| ***Дорога*** |
| --- |
| длина<br>ширина |
| |

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.



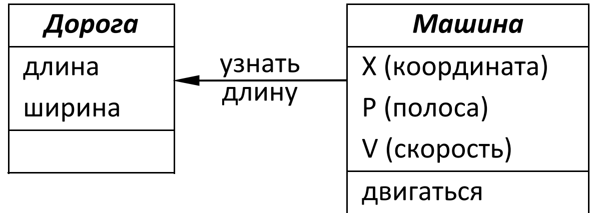

| *Машина* |
| --- |
| X (координата)<br>P(полоса)<br>V (скорость) |
| двигаться |

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».

**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 сложный тип данных, который может объединять переменные различного типа в единый блок. Однако, в отличие от структур в языке С, класс 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: 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*).

Описание класса *Дорога* в программе на C++ выглядит так:

```
class TRoad
{
  float Length;
  int Width;
};
```

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.

Если мы хотим работать с объектом класса **TRoad**, в программе нужно объявить соответствующую переменную:

```
TRoad road;
```

Свойства дороги можно изменить с помощью точечной нотации, с которой вы познакомились, работая со структурами, например, так:

```
road.Length = 60;
road.Width = 3;
```

Однако, если ввести такую команду, мы получим сообщение об ошибке. Дело в том, что по умолчанию все поля класса – закрытые (англ. **private** – частный), то есть, они недоступны другим объектам, а также функциям и процедурам, не принадлежащим классу (о них мы будем говорить дальше). Для того, чтобы открыть доступ к полям, нужно добавить описатель **public** (англ. *общедоступный*)[2]:

```
class TRoad
{
  public:
    float Length;
    int Width;
};
```

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

[2] В языке С вместо слова **class** можно использовать **struct**. Различие между ними только в том, что по умолчанию все поля структур общедоступны, то есть относятся к секции **public**.

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

```
class TRoad
{
  public:
   float Length;
   int Width;
   };
main()
{
  TRoad road;
  road.Length = 60;
  road.Width = 3;
}
```

Теперь разберёмся, что происходит при выполнении строки программы

```
TRoad road;
```

Здесь для создания объекта в памяти вызывается специальный метод класса, который называется *конструктором.*

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

Обратите внимание, что мы никак не описывали этот метод при объявлении класса. В этом случае работает *конструктор по умолчанию,* который просто создает объект в памяти. Все поля будут содержать «мусор» – некоторые значения, которые невозможно предсказать заранее.

Если мы хотим вручную задать начальные значения для полей, необходимо объявить свой конструктор, например так:

```
class TRoad
{
  public:
    float Length;
    int Width;
    TRoad();        // объявление конструктора
};
```

Название конструктора всегда совпадает с названием класса. Код конструктора нужно расположить после такого объявления:

```
TRoad::TRoad()
{
  Length = 0;
  Width = 0;
}
```

Запись **TRoad::abc()** говорит о том, что метод **abc** принадлежит к классу **TRoad**. В данном случае – это конструктор (его имя совпадает с именем класса!); в нём все поля обнуляются.

При вызове конструктора можно задавать начальные значения полей. Для этого нужно создать *конструктор с параметрами*:

```
class TRoad
{
  public:
    ...
      // объявление конструктора с параметрами
    TRoad ( float length0, int width0 );
};
```

Код конструктора может выглядеть так:

```
TRoad::TRoad ( float length0, int width0 )
{
  if ( length0 > 0 )
       Length = length0;
  else Length = 1;
```

```
    if ( width0 > 0 )
         Width = width0;
    else Width = 1;
  }
```

Здесь проверяется правильность переданных параметров, чтобы по ошибке длина и ширина дороги не оказались нулевыми или отрицательными[3]. Чтобы вызвать такой конструктор, при создании объекта нужно указать аргументы (значения параметров):

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

В поле **Length** записывается значение 60, а в поле **Width** – 3 (длина этой дороги – 60 единиц, она содержит 3 полосы).

Параметрам можно задавать значения по умолчанию. Например, конструктор, объявленный как

```
   TRoad ( float length0, int width0 = 3 );
```

можно вызывать с одним параметром (начальным значением поля **Length**), тогда ширина дороги (количество полос) по умолчанию будет равна 3:

```
  TRoad road ( 60 ); // Width = 3 по умолчанию
```

Все параметры, для которых заданы значения по умолчанию, должны быть последними в списке параметром.

У класса может быть несколько конструкторов, но они должны отличаться параметрами (количеством или типами).

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 и один метод – процедуру **move**. Координата **X** и скорость **V** – это вещественные значения, а номер полосы **P** – целое.

```
  class TCar
  {
   public:
    float X, V;
    int P;
    TRoad *Road;
    void move();
    TCar ();      // конструктор без параметров
    TCar ( TRoad *road0, int p0, float v0 );
  };
```

Так как объекты-машины должны обращаться к объекту «дорога», в область данных включено дополнительное поле **Road** – указатель на дорогу. Конечно, это не значит, что в состав машины входит дорога. Напомним, что это только ссылка, и сразу после создания объекта-машины нужно записать в неё адрес заранее созданного объекта класса *Дорога*.

Мы объявили два конструктора: один без параметров, в котором всем полям присваиваются нулевые значения[4]:

```
  TCar::TCar ()
  {
    Road = NULL; P = 0; V = 0; X = 0;
  }
```

а второй – с тремя параметрами, которые позволяют сразу связать машину с дорогой и определить полосу движения и скорость (начальная координата **X** устанавливается в нуль):

[3] Конечно, в реальной программе при передаче неправильных данных нужно выдавать сообщение об ошибке.

[4] В языке C++ значение NULL совпадает с 0, поэтому можно написать **Road = 0**. Однако в этом случае с первого взгляда неясно, что **Road** – это указатель.

```
TCar::TCar ( TRoad *road0, int p0, float v0 )
{
  Road = road0; P = p0; V = v0; X = 0;
}
```

Теперь займемся *реализацией* (программированием) метода **move** («двигаться»). В этом методе нужно вычислить новую координату **X** машины и, если она находится за пределами дороги, установить её 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 можно измерять время в этих интервалах, а за скорость $V$ принять расстояние, проходимое машиной за один интервал. Тогда метод **move**, описывающий изменение положения машины за один интервал ($\Delta t = 1$), может выглядеть так:

```
void TCar::move()
{
  X = X + V;
  if ( X > Road->Length ) X = 0;
}
```

Обратите внимание, что поле **Road** – это указатель, поэтому обращение к полю соответствующего ему объекта-дороги идет через оператор **->**.

## Основная программа

В основной программе объявим массив объектов-машин:

```
const int N = 3;
TCar cars[N];
```

При выполнении этих строк с помощью конструктора без параметров создается массив из трёх объектов-машин. Теперь можно задать для них начальные значения полей:

```
int i;
for ( i = 0; i < N; i++ )
  {
  cars[i].Road = &road;
  cars[i].P = i + 1;
  cars[i].V = 2.0*(i + 1);
  }
```

При вызове конструктора задаются три параметра: адрес объекта «дорога» (его нужно создать до выполнения этого цикла), номер полосы и скорость. В приведенном варианте машина на полосе с порядковым номером **i** идет со скоростью **2i** единиц за один интервал моделирования.

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

```
do {
  for ( i = 0; i < N; i++ )
    cars[i].move();
  }
while ( !kbhit() );
```

Этот цикл закончится тогда, когда пользователь нажмёт на любую клавишу и функция **kbhit** вернет значение **true**. Для использования функции **kbhit** нужно подключить заголовочный файл **conio.h**.

Полностью основная программа выглядит так:

```
#include <conio.h>
  // описание классов TRoad и TCar
main()
```

```
{
  TRoad road ( 60 );
  const int N = 3;
  TCar cars[N];
  int i;
  for ( i = 0; i < N; i ++ )
    {
    cars[i].Road = &road; // записать адрес дороги
    cars[i].P = i + 1;
    cars[i].V = 2.0*(i + 1);
    }
  do
    {
    for ( i = 0; i < N; i ++ )
      cars[i].move();
    }
  while ( !kbhit() );
}
```

Второй вариант создания массива машин основан на использовании указателей. Объявим массив указателей на объекты класса **TCar**:

```
const int N = 3;
TCar *cars[N];
```

Как вы знаете, пока это просто «висящие» ссылки, которые не указывают ни на какие объекты, и использовать их нельзя. Для создания объектов будем в цикле вызывать оператор **new**:

```
for ( i = 0; i < N; i++ )
  cars[i] = new TCar ( &road, i+1, 2.0*(i+1) );
```

Здесь используется объявленный ранее конструктор с параметрами, ему передаётся адрес объекта-дороги, номер полосы и скорость машины. Поскольку в этом варианте **car[i]** – это указатель на машину с номером **i**, для обращения к полям и методам объекта используем оператор **->**:

```
for (i=0; i<N; i++)
  cars[i]->move();
```

Можно ли было написать такую же программу, не используя 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

- 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 объявляется переменная для работы с объектом некоторого класса? Что в ней хранится?
4. Как в памяти создается экземпляр класса (объект)?
5. Что такое конструктор? Может ли быть несколько конструкторов у одного класса?
6. Что такое точечная нотация? Как она используется при работе с объектами?
7. Как можно задать начальные значения для полей объекта?
8. Почему в методе **TCar.move** не объявлены переменные **X** и **V**?
9. 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»).

Заметим, что в объектно-ориентированном программировании 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 **FColor**, 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
{
  private:
    string FColor;
};
```

Те элементы (поля и методы), которые нужно скрыть, в описании класса[5] помещают в «частный» раздел (англ. *private*). Таким образом, поле **FColor** закрытое. Имена всех закрытых полей далее будем начинать с буквы **F** (от англ. *field*, поле). 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 **FColor**, а второй – присваивать полю новое значение. Эти методы доступа назовем **getColor** (англ. *получить Color*) и **setColor** (англ. *установить Color*):

```
class TPen {
  private:
    string FColor;
  public:
```

[5] Мы уже сталкивались с тем, что по умолчанию все поля класса скрытые, так что в данном случае описатель **private** можно было не писать.

```
    string getColor ();
    void setColor ( string newColor );
  };
```

Обратите внимание, что оба метода находятся в секции **public** (общедоступные).

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** можно написать так:

```
string TPen::getColor () { return FColor; }
```

В методе **setColor** мы можем обрабатывать ошибки, не разрешая присваивать полю недопустимые значения. Например, установим, что символьная строка с кодом цвета, передаваемая нашему объекту, должна состоять из шести символов. Если эти условия не выполняются, будем записывать в поле **FColor** код чёрного цвета **'000000'**:

```
void TPen::setColor ( string newColor )
{
  if ( newColor.length()!= 6 )
       FColor = "000000";  // если ошибка, то чёрный цвет
  else FColor = newColor;
}
```

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

```
pen.setColor ( "FFFF00" );                // изменение цвета
cout << "цвет пера: " << pen.getColor();  // получение цвета
```

Итак, мы скрыли внутренние данные, но одновременно обращение к свойствам стало выглядеть довольно неуклюже: вместо **pen.color="FFFF00"** теперь нужно писать **pen.setColor("FFFF00")**.

Фактически мы определили свойство «цвет» для объектов класса **TPen**, другие объекты могут изменять и читать это значение, причём для обмена данными с «внешним миром» важно лишь то, что свойство «цвет» – 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, так как большинство стандартных функций используют числовые коды цвета. Поэтому лучше хранить код цвета как целое число, и поле **FColor** сделать целого типа:

```
class TPen
{
  private:
    int FColor;
  public:
    string getColor();
    void setColor( string newColor );
};
```

При этом необходимо поменять методы **getColor** и **setColor**, которые непосредственно работают с этим полем. Обратите внимание, что заголовки методов (то есть *интерфейс*, способ общений объекта с «внешним миром») остались прежними (!), и другие объекты «не заметят», что во внутреннее устройство объектов класса **TPen** внесены какие-то изменения.

Итак, метод **getColor** должен вернуть символьную строку, в которой записан шестнадцатеричный код цвета, хранящегося в поле **FColor** как целое число. Для того чтобы преобразовать число в символьную строку, используем *строковый поток* (**stringstream**), в который можно записывать данные, как в обычный поток вывода, и из которого потом эти данные можно прочитать, как из входного потока. Трюк состоит в том, что мы будем записывать целое число, а читать строку. Запись в поток числа в шестнадцатеричной системе счисления может выглядеть так:

```
#include <sstream>
...
```

```
stringstream s;
s << hex << FColor;
```

Для работы со строковыми потоками необходимо подключить заголовочный файл **stream**. Формат **hex** обозначает «вывести число в шестнадцатеричной системе счисления». Однако это немного не то, что нужно. Например, цвет $FF_{16}$ будет выведен как «FF», а нам нужно – «0000FF». То есть, необходимо как-то указать, что требуется вывести число в 6 позициях и все пустые позиции слева заполнить нулями. Это делается с помощью манипуляторов вывода:

```
s << setfill('0') << setw(6) << hex << FColor;
```

Здесь используется два манипулятора ввода-вывода: **setw(6)** устанавливает ширину поля 6 символов, а **setfill('0')** задает заполнение пустых позиций цифрой 0. Напомним, что для использования этих манипуляторов необходимо подключить заголовочный файл **iomanip**:

```
#include <iomanip>
```

В итоге метод **getColor** приобретает такой вид:

```
string TPen::getColor()
{
  stringstream s;
  s << setfill('0') << setw(6) << hex << FColor;
  return s.str();
}
```

Обратный переход – от символьной строки с шестнадцатеричным кодом цвета к числовому коду – может быть выполнен в обратном порядке (записываем строку, читаем число):

```
void TPen::setColor ( string newColor )
{
  stringstream s;
  if ( newColor.length()!= 6 )
    FColor = 0;  // если ошибка, то чёрный цвет
  else {
    s << newColor;
    s >> hex >> FColor;
    }
}
```

Если длина входной строки равна 6 символам, мы записываем её в строковый поток, а затем читаем из этого потока число, записанное в шестнадцатеричной системе 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 «цвет» 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.

Мы уже несколько раз говорили о том, что фактически объект **TPen** имеет свойство «цвет», для работы с которым используются два метода: метод чтения **getColor** и метод записи **setColor** (на программистском жаргоне они называются «геттер» и «сеттер»). Однако понятия «свойство» нет в языке C++, и эти два метода формально никак не связаны! Этот недостаток устранён во многих современных языках программирования, где понятие «свойство» используется. Один из таких – язык C#, основанный на идеях C и C++ (мы немного познакомимся с ним в конце этой главы).

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

Обращение к свойству внешне выглядит как работа с переменной объекта, но на самом деле при записи и чтении свойства вызываются методы объекта.

Например, в C# можно ввести свойство **color** так (сначала рассмотрим простейший случай):

```
class TPen
{
  private string FColor;     // закрытое поле
```

```
  public string color         // открытое свойство
  {
    get { return FColor; }    // метод чтения
    set { FColor = value; }   // метод записи
  }
}
```

У объектов такого класса есть закрытое (**private**) поле **FColor** и открытое (**public**) свойство **color**. Для работы с этим свойством используются два метода: метод, объявленный в фигурных скобках после слова **get**, возвращает текущее значение свойства, а метод, объявленный после слова **set** – устанавливает его новое значение. Параметр, который передаётся процедуре-«сеттеру», всегда называется **value** (это ключевое слов языка C#).

Обращение к свойству выполняется так же, как к обычному открытому полю объекта:

```
string s = pen.color;
pen.color = "FFFFFF";
```

Несложно обеспечить и защиту от неправильного значения входе (так же, как мы делали это на языке C++):

```
public string color
{
  get { return FColor; }
  set
    {
    if ( value.Length != 6 )
         FColor = "000000";  // если ошибка, то чёрный цвет
    else FColor = value;
    }
}
```

А вот так мы можем изменить внутреннее устройство объекта, сохранив интерфейс: свойство **color** имеет строковый тип, а данные хранятся как целое число:

```
class TPen
{
  private int FColor;      // закрытое поле
  public string color      // открытое свойство
  {
    get { return FColor.ToString ( "X6" ); }
    set { FColor = Convert.ToInt32 ( value, 16 ); }
  }
}
```

Здесь для преобразования целого числа в символьную строку используется метод **ToString** из библиотеки C#, значение аргумента «**X6**» обозначает, что нужно записать число в шестнадцатеричной системе в 6 позициях. Обратный перевод выполняет метод **ToInt32** класса **Convert**, второй аргумент 16 – это система счисления, в которой записано число. Обработка ошибок здесь достаточно непростая, и мы пока не будем её рассматривать.

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 класса на C++ мы просто не будем создавать метод для записи этого свойства[6]:

```
class TCar {
  private:
    double Fv;
  public:
    double getV() { return Fv; }
  };
```

[6] В языке C# в таких случаях не пишут метод-«сеттер».

Обратите внимание, что короткий код метода **`getV`** записан в объявлении класса. Такие методы называются *встроенными* (**`inline`**), это значит, что компилятор подставляет в точку вызова все тело метода, а не оформляет отдельную подпрограмму в машинном коде.

Таким образом, доступ к внутренним данным объекта возможен, как правило, только с помощью методов. При использовании скрытия данных (*инкапсуляции*) длина программы чаще всего увеличивается, однако мы получаем и важные преимущества. Код, связанный с объектом, разделен на две части: общедоступную часть (секция **`public`**) и закрытую (**`private`**). Их можно сравнить с надводной и подводной частью айсберга. Объект взаимодействует с другими объектами только с помощью своих общедоступных свойств и методов (*интерфейса*). 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 секции **`public`** и **`private`** в описании классов? Как определить, в какую из них поместить свойство или метод?
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. Можно ли с помощью свойства обращаться напрямую к полю объекта, не используя метод?
7. Зачем нужны свойства «только для чтения»? Приведите примеры.
8. Подумайте, в каких ситуациях может быть нужно свойство «только для записи» (которое нельзя прочитать)? Как ввести такое свойство в описание класса?

## *Задачи*

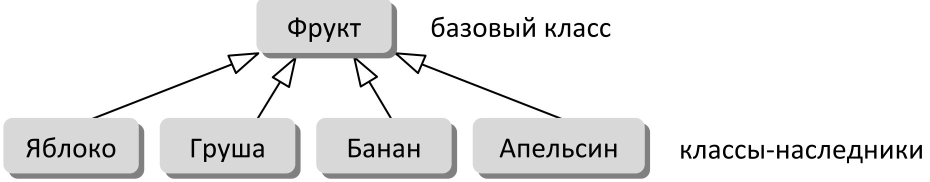
1. Измените построенную ранее программу моделирования движения так, чтобы все поля у объектов были закрытыми. Используйте свойства для доступа к данным.

# § 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[7] (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).




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

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

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*.

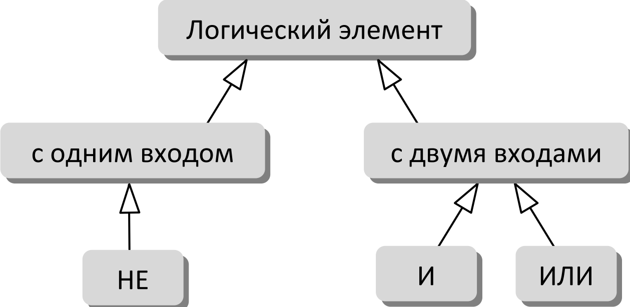
В то же время мы не можем сказать, что «машина – это разновидность двигателя», поэтому класс *Машина* не является наследником класса *Двигатель.* Двигатель – это составная часть машины, поэтому объект класса *Машина* содержит в себе объект класса *Двигатель*. Отношения между двигателем и машиной – это 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)<br>In2 (вход 2)<br>Res (результат) |
| calc |

Любой логический элемент должен уметь вычислять значение выхо-

да по известным входам, для этого введем в класс метод **calc**:

```
class TLogElement
{
  public:
    bool In1, In2, Res;
    void calc();
};
```

В таком варианте все данные открытые (общедоступные). Чтобы защитить внутреннее устройство объекта, скроем внутренние поля (добавив в их название первую букву F) и введём методы для обращения к этим полям:

```
class TLogElement
{
  private:
    bool FIn1, Fin2, FRes;
    void calc();
  public:
    bool getIn1() { return FIn1; }
    void setIn1 ( bool newIn1 );
    bool getIn2() { return Fin2; }
    void setIn2 ( bool newIn2 );
    bool getRes() { return FRes; }
};
```

Обратите внимание, что свойство **Res** – это свойство только для чтения, и другие объекты не могут его менять. Кроме того, мы поместили процедуру **calc** в скрытый раздел (**private**), потому что пересчет результата должен выполняться автоматически при изменении любого входного сигнала (другие объекты не должны об этом беспокоиться).

Несложно написать процедуру **setIn1** (и аналогичную ей процедуру **setIn2**), в ней новое входное значение присваивается полю и сразу пересчитывается результат:

```
void TLogElement::setIn1(bool newIn1)
{
  FIn1 = newIn1;
  calc();
}
```

Если внимательно проанализировать построенное описание класса, можно выявить несколько проблем. Во-первых, элемент «НЕ» имеет только один вход, поэтому не хотелось бы для него открывать доступ к свойству **In2** (это не нужно и может привести к ошибкам).

Во-вторых, процедуру **calc** *невозможно написать*, пока мы не знаем, какой именно логический элемент моделируется. С другой стороны, мы знаем, что такую процедуру имеет любой логический элемент, то есть она должна принадлежать именно классу **TLogElement**. Здесь можно написать процедуру-«заглушку» (которая ничего не делает):

```
void TLogElement::calc(){}
```

Но нужно как-то дать возможность классам-наследникам изменить тот метод так, чтобы он выполнял требуемую операцию. Такой метод называется *виртуальным*. Более точное определение этого понятия мы дадим несколько позже.

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.

Мы уже говорили о том, что метод **calc** не нужно делать общедоступным (**public**). В то же время его нельзя делать закрытым (**private**), потому что в этом случае он не будет доступен классам-наследникам. В таких случаях в описании класса используется третий блок (кроме **private** и **public**), который называется **protected** (защищенный). Данные и методы в этом блоке доступны для классов-наследников, но недоступны для других классов. В этот же блок **protected** мы переместим объявление поля **FRes** (его будут менять наследники в процедуре

**calc**) и методы доступа к полю **FIn2** – они будут скрыты при использовании элемента «НЕ», а элементы с двумя входами их «откроют» (чуть позже).

```
class TLogElement
{
  private:
    bool FIn1, Fin2;
  protected:
    bool FRes;
    virtual void calc()=0;
    bool getIn2() { return Fin2; }
    void setIn2 ( bool newIn2 );
  public:
    bool getIn1() { return FIn1; }
    void setIn1 ( bool newIn1 );
    bool getRes() { return FRes; }
};
```

Обратите внимание на объявление метода **calc**: в самом начале стоит слово **virtual** (виртуальный), а в конце описания – «**=0**». Описатель **virtual** говорит о том, что метод **calc** – виртуальный, и классы-наследники могут его переопределять. Как уже отмечалось, мы должны объявить этот метод (ввести его в описание класса), поскольку он должен быть у любого логического элемента. С другой стороны, *невозможно* написать процедуру **calc**, пока неизвестен тип логического элемента. Такой метод называется *абстрактным* и обозначается символами «**=0**». Для абстрактного метода не нужно ставить «заглушку».

**Абстрактный метод** – это метод класса, который объявляется, 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.

Итак, полученный класс **TLogElement** – это абстрактный класс (компилятор определит это автоматически). 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.

## 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: public TLogElement
{
  protected:
    void calc();
};
```

После имени нового класса **TNot** через двоеточие указано название базового класса и перед ним слово **public**. Все объекты класса **TNot** обладают всеми свойствами и методами класса **TLogElement**. Описатель **public** говорит о том, что видимость методов базового класса не изменяется: все защищённые (**protected**) методы класса **TLogElement** остаются защищёнными и в новом класса, а все общедоступные методы (**public**) также останутся общедоступными.

Новый класс *переопределяет* виртуальный метод **calc** (слово **virtual** уже не нужно, так как мы указали его в базовом классе). Заметим, что у базового класса **TLogElement** этот метод не реализован – он абстрактный, поэтому в данном случае мы фактически программируем метод, объявленный в базовом классе. Для элемента «НЕ» он выглядит очень просто:

```
void TNot::calc()
```

```
{
  FRes =!getIn1();
}
```

Обратите внимание, что этот метод не может обратиться к закрытому полю **FIn1** базового класса, и вместо этого использует метод **getIn1**.

Класс **TNot** уже не абстрактный, потому абстрактный метод предка переопределен и теперь известно, что делать при вызове метода **calc**. Поэтому можно создавать объект этого класса и использовать его:

```
TNot n;
n.setIn1 ( false );
cout << n.getRes();
```

Остальные элементы имеют два входа и будут наследниками класса

```
class TLog2In: public TLogElement
{
  public:
    TLogElement::setIn2;
    TLogElement::getIn2;
};
```

Единственное, что делает этот класс – открывает методы доступа к полю **FIn2**, переводит эти два метода базового класса в раздел **public**.

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 определяют конкретные логические элементы:

```
class TAnd: public TLog2In
{
  protected:
    void calc();
};
class TOr: public TLog2In
{
  protected:
    void calc();
};
```

Реализация переопределенного метода **calc** для элемента «И» выглядит так:

```
void TAnd::calc()
{
  FRes = getIn1() && getIn2();
}
```

Для элемента «ИЛИ» этот метод определяется аналогично.

Обратим внимание на метод **setIn1**, введенный в базовом классе:

```
void TLogElement::setIn1( bool newIn1 )
{
  FIn1 = newIn1;
  calc();
}
```

В нем вызывается метод **calc**, который пересчитывает значение на выходе логического элемента при изменении входа. Какой же метод будет вызван, если в базовом классе **TLogElement** он только объявлен, но не реализован?

Проблема в том, что для вызова любой процедуры нужно знать её адрес в памяти. Для обычных методов транслятор сразу записывает в машинный код нужный адрес, потому что он заранее известен. Это так называемое *статическое* связывание (связывание на этапе трансляции), при выполнении программы этот адрес не меняется.

В нашем случае адрес метода неизвестен: в классе **TLogElement** его нет вообще, а у каждого класса-наследника адрес метода **calc** – свой собственный. Чтобы выйти из положения, используется *динамическое связывание*, то есть адрес вызываемой процедуры определяется при

*выполнении* программы, когда уже определен тип объекта, с которым мы работаем. Такой метод нужно объявлять *виртуальным*, что мы и сделали ранее. Это означает не только то, что его могут переопределять наследники, но и то, что будет использоваться динамическое связывание. Теперь можно дать полное определение виртуального метода.

**Виртуальный метод** – это метод базового класса, который могут переопределить классы-наследники так, что конкретный адрес вызываемого метода определяется только при выполнении программы.

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:

```
main()
{
  TNot elNot;
  TAnd elAnd;
  int A, B;
  cout << " A  B  !(A&B)" << endl;
  cout << "-------------" << endl;
  for ( A = 0; A <= 1; A++ ) {
    elAnd.setIn1 ( A );
    for ( B = 0; B <= 1; B++ ) {
      elAnd.setIn2 ( B );
      elNot.setIn1 ( elAnd.getRes() );
      cout << " " << A << "  " << B
           << "    " << elNot.getRes() << endl;
      }
    }
}
```

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**, они подаются на входы элемента «И», а его выход – на вход элемента «НЕ».

## 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 можно вынести всё, что относится к логическим элементам. В заголовочный файл (назовём его **log_elem.h**) включим объявления всех классов (это *интерфейс* модуля):

```
class TLogElement
{
  private:
    bool FIn1, FIn2;
  protected:
    bool FRes;
    virtual void calc() = 0;
    bool getIn2() { return FIn2; }
    void setIn2 ( bool newIn2 );
  public:
    bool getIn1() { return FIn1; }
    void setIn1 ( bool newIn1 );
    bool getRes() { return FRes; }
};
class TLog2In: public TLogElement
{
  public:
```

```
    TLogElement::setIn2;
    TLogElement::getIn2;
};
class TNot: public TLogElement
{
  protected:
    void calc();
};
class TAnd: public TLog2In
{
  protected:
    void calc();
};
class TOr: public TLog2In
{
  protected:
    void calc();
};
```

Сам модуль (так называемая *реализация*), содержит код методов класса:

```
#include "log_elem.h"
void TLogElement::setIn1 ( bool newIn1 )
{
  FIn1 = newIn1;
  calc();
}
void TLogElement::setIn2 ( bool newIn2 )
{
  FIn2 = newIn2;
  calc();
}
void TNot::calc()
{
  FRes = !getIn1();
}
void TAnd::calc()
{
  FRes = getIn1() &&  getIn2();
}
void TOr::calc()
{
  FRes = getIn1() ||  getIn2();
}
```

Теперь нужно создать проект и включить в него оба файла – основную программу и модуль. Основная программа при этом не изменится, только в самом начале нужно подключить заголовочный файл **log_elem.h** из текущего каталога:

```
#include "log_elem.h"
main()
{
  TNot elNot;
  TAnd elAnd;
  ...
}
```

## 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 поля и один метод:

```
class TLogElement
{
  private:
    TLogElement *FNextEl;
    int FNextIn;
    ...
  public:
    void Link ( TLogElement *nextElement, int nextIn = 1 );
};
```

Поле **FNextEl** хранит ссылку на следующий логический элемент (указатель на **TLogElement**), а поле **FNextIn** – номер входа этого следующего элемента, к которому подключен выход данного элемента. С помощью общедоступного метода **Link** можно связать данный элемент со следующим:

```
void TLogElement::Link( TLogElement *nextElement, int nextIn )
{
  FNextEl = nextElement;
  FNextIn = nextIn;
}
```

Обратите внимание, что при объявлении этого метода в классе **TLogElement** после второго параметра указано «**=1**». Это так называемое *значение по умолчанию*: если второй параметр не указан, считается, что он равен 1. Все параметры с заданными значениями по умолчанию должны быть записаны в конце списка параметров.

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

```
void TLogElement::setIn1 ( bool newIn1 )
{
  FIn1 = newIn1;
  calc();
  if ( FNextEl )
    switch ( FNextIn ) {
      case 1: FNextEl->setIn1 ( getRes() ); break;
      case 2: FNextEl->setIn2 ( getRes() ); break;
      }
}
```

Запись **if(FNextEl)** означает «если следующий элемент задан», то есть если указатель **FNextEl** ненулевой. Если он не был установлен, значение поля **FNextEl** будет равно **NULL** и никаких дополнительных действий не выполняется. Правда, здесь может возникнуть одна проблема: при создании объекта его поля не заполняются нулями, как, например, в языке Паскаль. Поэтому в класс **TLogElement** нужно добавить конструктор по умолчанию, который сделает это явно:

```
TLogElement::TLogElement()
{
  FNextEl = NULL;
}
```

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

```
TNot elNot;
TAnd elAnd;
elAnd.Link ( &elNot );  // или так: elAnd.Link(&elNot,1);
```

```
for ( A = 0; A <= 1; A ++ ) {
  elAnd.setIn1 ( A );
  for ( B = 0; B <= 1; B++ ) {
    elAnd.setIn2 ( 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 виртуальный метод?
9. Что такое полиморфизм?
10. Что такое абстрактный класс? Почему нельзя создать объект этого класса?
11. Как транслятор определяет, что тот или иной класс – абстрактный?
12. Что нужно сделать, чтобы класс-наследник абстрактного класса не был абстрактным?
13. Зачем нужен описатель **protected**? Чем он отличается от **private** и **public**?
14. Какие преимущества даёт применение модулей в программе?
15. Можно ли всё содержимое модуля включить в заголовочный файл? Чем это плохо?
16. Объясните, как объекты могут передавать сообщения друг другу.
17. Что такое значение параметра функции по умолчанию? Когда это может быть полезно?

### *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).

# § 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.

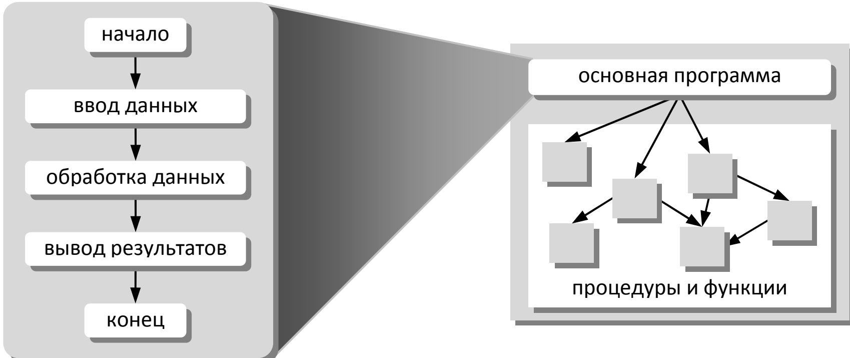
**Сообщение** – это блок данных определённой структуры, который используется для обмена информацией между объектами.

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.

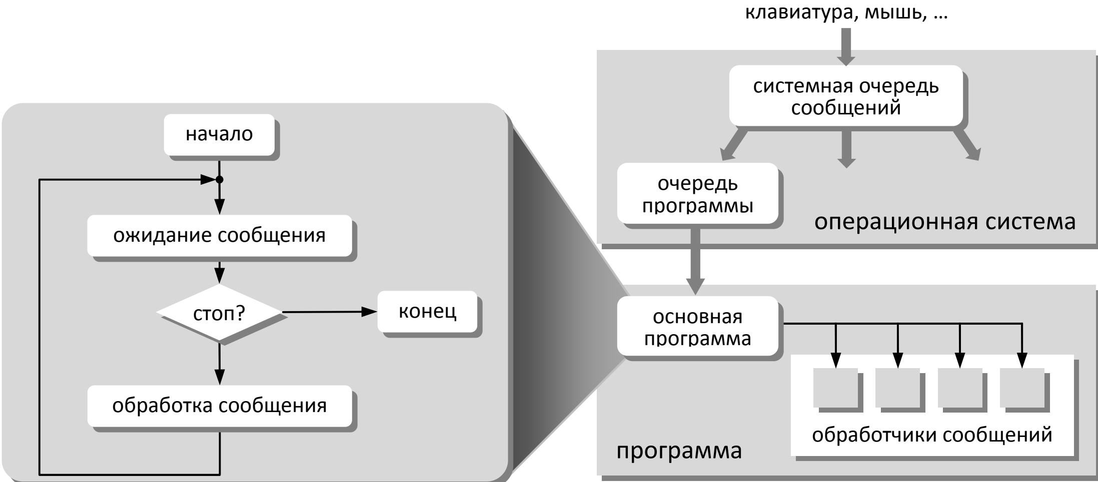
Во всех этих примерах программа должна «включаться в работу» только тогда, когда получит условный сигнал, то есть произойдет некоторое *событие* (изменение состояния).

**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.) сначала 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 некоторые – 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. Свободная *RAD*-среда *Lazarus* (*lazarus.freepascal.org*) 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. Основы программирования в RAD-средах

## Общий подход

В этом разделе мы продемонстрируем основные принципы программирования в RAD-средах на примере бесплатной среды *Microsoft Visual Studio Express* для операционной системы *Windows*. Тем не менее, все изучаемые здесь принципы справедливы также и для других аналогичных программ, например, *Delphi* и *Lazarus.*

Для выполнения примеров мы будем использовать язык C# (читается как «Си шарп»), который можно считать развитием языков C и C++. С помощью языка C# создаются программы для платформы **.NET**, разработанной компанией *Microsoft* для операционной системы *Windows.* Программы на C# транслируются не в команды процессора, а в код на специальном языке CIL (*Common Intermediate Language* – «общий промежуточный язык»). Этот код выполняет виртуальная машина среды **.NET**, которая называется CLR (*Common Language Runtime* – общая среда выполнения для языков). При использовании такого подхода появляется возможность объединять части программ, написанные на разных языках (Visual Basic .NET, Visual C#, F#, PascalABC.NET). В UNIX-подобных операционных системах (*Linux, Mac OS X*) для выполнения таких программ используется среда *Mono.*

Разработка программы начинается с создания *проекта*. Так называется набор файлов, из которых компилятор строит исполняемый файл программы. В состав проекта обычно входят:

- проект (файл с расширением **.csproj**, от *CSharp Project* – проект *C#*), в котором содержится описание проекта в формате XML;
- модули на языке C#, из которых состоит программа (***.cs**);
- файлы ресурсов (***.resx**), содержащие, например, строки для перевода сообщений программы на другие языки.

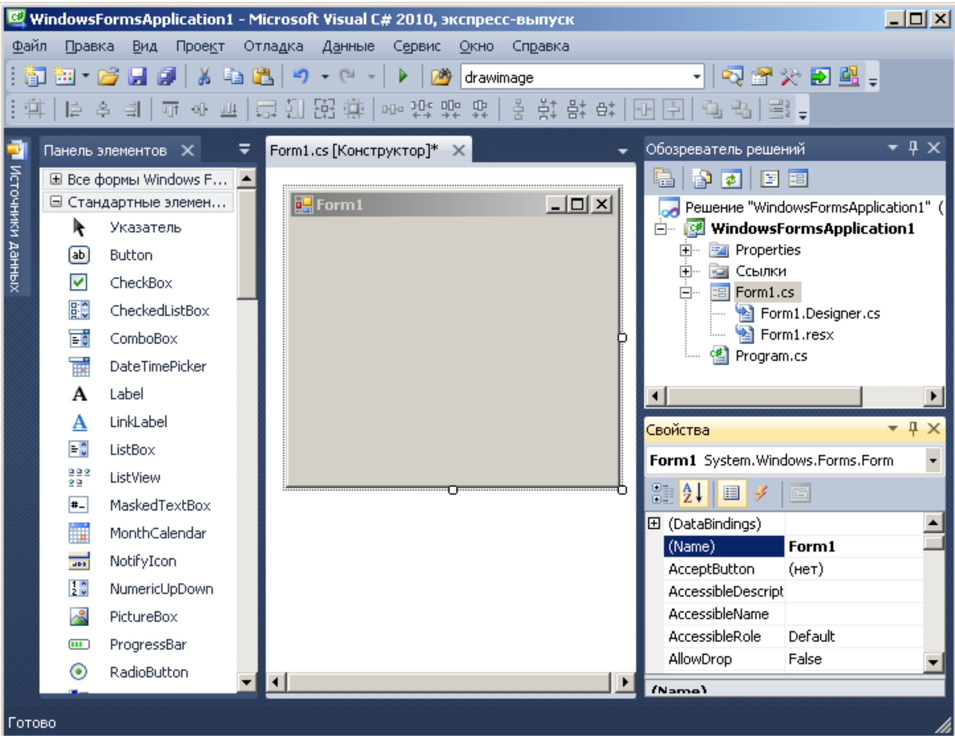
Проекты в современных версиях *Visual Studio* объединяются в *решения*. В состав решения может быть включено несколько проектов, например, программа для пользователя разрабатываемой системы (первый проект), программа для администратора (второй проект), средства разработчика и т.п.

Основная программа проекта находится в файле **Program.cs** (здесь и далее приводятся имена файлов, которые назначаются по умолчанию). В программе с графическим интерфейсом может быть несколько окон, которые называют *формами*. С каждой формой связана пара файлов (с расширением **cs**): в одном хранятся данные о расположении и свойствах элементов интерфейса (в названиях этих файлов есть слово **Designer**, например, **Form1.Designer.cs**), а во втором (**Form1.cs**) – программный код обработчиков сообщений, связанных с этой формой.

Одна форма – главная, она появляется на экране при запуске программы. Когда пользователь закрывает главную форму, работа программы завершается.

## Простейшая программа

Для создания проекта с графическим окном нужно выбрать пункт меню *Файл – Создать проект* и в появившемся окне отметить вариант *Приложение Windows Forms*. При этом создается вполне рабочая программа, которую сразу же можно запустить на выполнение клавишей F5.




При работе в среде *Visual Studio* чаще всего используются следующие окна:

- главное окно;
- *Панель элементов*;
- *Обозреватель решений*;
- окно *Свойства*;
- конструктор формы;
- окно исходного кода.

**Главное окно** среды (расположенное сверху) содержит меню и панель инструментов – кнопки для быстрого вызова команд

На **Панели элементов** размещены готовые объекты (кнопки, поля ввода, списки и т.п.), которые можно использовать в своих программах.

С помощью **Обозревателя решений** можно выбрать для редактирования любой файл проекта (решения).

Вся программа, согласно принципам ООП, состоит из объектов. Для настройки свойств объектов используется окно *Свойства*. В нем две основных вкладки: *Свойства*, где можно изменить общедоступные свойства объекта, и *События*, на которой из списка подходящих подпрограмм выбираются его обработчики событий.

После двойного щелчка мышью по названию файла **Form1.cs** открывается **Конструктор формы**. В этом режиме можно мышью перетаскивать компоненты с *Панели элементов* на форму

и изменять их размеры и расположение. Таким образом, интерфейс программы полностью строится с помощью мыши. При нажатии клавиши **F7** мы увидим в текстовом редакторе содержимое файла **Form1.cs**, где хранятся обработчики событий формы[8]:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace Project1
{
  public partial class Form1: Form  {
    public Form1() { InitializeComponent(); }
    }
}
```

Сначала с помощью команды **using** подключаются пространства имён среды **.NET** (вспомните подключение стандартного пространства имён **std** в C++). Например, если убрать строчку

```
using System.Windows.Forms;
```

то вместо **Form** придётся писать **System.Windows.Forms.Form**, полностью указывая пространство имён, в котором определён класс **Form**.

Затем вводится пространство имён проекта с именем **Project1**:

```
namespace Project1
{
  ...
}
```

В этом пространстве имен определяется новый класс **Form1**, который является наследником базового класса **Form**:

```
public partial class Form1: Form
{
  ...
}
```

Описатель **public** обозначает, что к объектам этого класса могут обращаться и другие классы, а описатель **partial** указывает на то, что класс **Form1** описывается по частям в разных файлах.

Единственный метод, определённый в классе **Form1**, – это конструктор, который сводится к вызову процедуры **InitializeComponent**:

```
public Form1() { InitializeComponent(); }
```

Эта процедура устанавливает свойства формы и расположенных на ней элементов при создании формы. Она находится во втором файле, связанном с формой – **Form1.Designer.cs**. Он строится автоматически и менять его не рекомендуется (но посмотреть полезно).

Основная программа (*точка входа*) расположена в файле **Program.cs**.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;
namespace Project1
{
  static class Program  {
    static void Main() {
      Application.EnableVisualStyles();
      Application.SetCompatibleTextRenderingDefault ( false );
      Application.Run ( new Form1() );
```

[8] С помощью сочетания клавиш Shift+F7 можно перейти обратно в конструктор формы.

```
            }
        }
    }
```

В уже знакомом нам пространстве имен проекта **Project1** создаётся статический (**static**) класс **Program**. Статический класс – это класс особого типа, для которого нельзя создать экземпляр. Он содержит только методы обработки внешних данных, у него нет внутренних переменных (состояния). Все методы статического класса также объявляются статическими. В нашем случае это единственный метод **Main** – точка входа, с которой начинается выполнение основной программы. В этом методе выполняются начальные установки для объекта (точнее, статического класса) **Application** (это приложение, то есть наша программа), а затем вызывается метод **Run** – цикл обработки сообщений. В параметрах метода **Run** указывается форма, которая появляется при запуске программы[9]. Здесь это новая форма класса **Form1**, которая создаётся с помощью оператора **new**.

## Свойства объектов

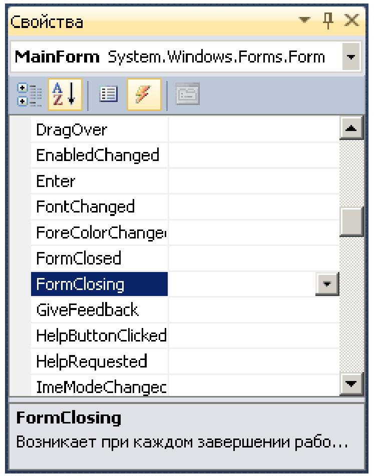
Панель *Свойства* позволяет просматривать и изменять свойства выделенного объекта (например, формы), а также устанавливать обработчики событий для этого объекта с помощью мыши и клавиатуры. Например, можно заметить, что при изменении размера формы изменяются ее свойства **Width** (англ. *ширина*) и **Height** (англ. *высота*), объединённые в группу свойств **Size** (англ. *размер*). В то же время можно вручную изменить эти координаты, вводя новые значения в соответствующие области на панели *Свойства* (при этом размеры формы меняются).

Свойство **Name** (англ. *имя*) – это название объекта-формы в программе. В имени можно использовать только латинские буквы, цифры и знак подчеркивания. Если изменить название формы в окне *Свйоства*, скажем, на **MainForm**, то это название автоматически изменится и в тексте модуля этой формы. Более того, название класса в файле **Form1.Designer.cs** тоже изменится на **MainForm**. Это означает, что многие изменения вносятся в код программы автоматически.

Перечислим еще некоторые важные свойства формы:

- **Text** – текст в заголовке окна;
- **BackColor** – цвет рабочей области;
- **Font** – шрифт надписей.

## Обработчики событий




На вкладке *События* в окне *Свойства* перечислены все события, которые может обрабатывать форма. Чтобы создать обработчик, нужно дважды щелкнуть мышью на поле справа от названия события. При этом открывается окно редактора, и в текст модуля автоматически добавляется пустой обработчик события (шаблон), в который остается только добавить нужные команды.

Рассмотрим простой пример. Вы знаете, что многие программы запрашивают подтверждение, когда пользователь завершает их работу. Для этого можно использовать обработчик события **FormClosing** (англ. *форма закрывается*). Обработчик, созданный в файле **Form1.cs** после двойного щелчка мыши в окне *Свойства*, имеет вид:

```
private void MainForm_FormClosing ( object sender,
                                    FormClosingEventArgs e )
{

}
```

[9] Отметим, что цикл обработки сообщений скрыт внутри метода **Run**, так что здесь тоже использован принцип инкапсуляции (скрытия внутреннего устройства).

Это закрытый метод (**private**), который может вызывать только сама форма.

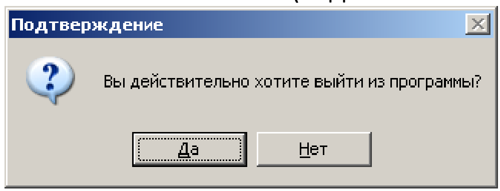
Одновременно этот метод добавляется в файл **Form1.Designer.cs**: в методе формы **InitializeComponent**, который вызывается при создании формы, новый обработчик связывается с событием **FormClosing** объекта с именем **this**, то есть, самой формы:

```
private void InitializeComponent()
{
  ...
  this.FormClosing += new
              System.Windows.Forms.FormClosingEventHandler
              ( this.MainForm_FormClosing );
  ...
 }
```

Как видно из заголовка процедуры, в обработчик передается два параметра:

- **sender** – ссылка на объект, от которого пришло сообщение о событии (в данном случае это будет сама форма);
- **e** – изменяемый объект, с помощью которого можно запретить закрытие формы, установив значение свойства **e.Cancel**, равное **true** (значение **false**, которое записано по умолчанию, разрешает закрытие формы и завершение работы программы).

В библиотеке C# есть класс **MessageBox**, позволяющий выводить на экран запрос с несколькими кнопками и получать ответ пользователя (код нажатой кнопки).




В тело обработчика можно добавить условный оператор, который присвоит свойству **e.Cancel** значение **true**, если пользователь подтвердил выход из программы:

```
private void MainForm_FormClosing ( object sender,
                                    FormClosingEventArgs e)
{
  DialogResult res;
  res = MessageBox.Show (
     "Вы действительно хотите выйти из программы?",
     "Подтверждение",
     MessageBoxButtons.YesNo, MessageBoxIcon.Question );
  if ( res == DialogResult.No )
    e.Cancel = true;
}
```

Здесь вызывается метод **MessageBox.Show**, и его результат записывается в переменную **res** типа **DialogResult**. Если это значение совпадает с константой **DialogResult.No** (то есть пользователь нажал на кнопку «Нет»), в свойство изменяемого параметра **e.Cancel** записывается значение **true**. При получении от обработчика такого значения (запрет закрыть окно), команда отменяется, иначе программа завершается.

Методу **MessageBox.Show** передаются несколько параметров:

- сообщение пользователю;
- заголовок окна;
- набор (множество) кнопок, которые появляются под текстом; в нашем случае это кнопки «Да» и «Нет», обозначенные константой **YesNo** из класса **MessageBoxButtons**;
- значение перечисляемого типа **MessageBoxIcon**, определяющее тип сообщения и рисунок слева от текста:

| | |
|---|---|
| **.Question** | вопрос; |
| **.Warning** | предупреждение; |
| **.Information** | информация; |
| **.Error** | сообщение об ошибке. |

Итак, мы построили простейшую работоспособную программу и познакомились со средой *Microsoft Visual Studio*. В следующем параграфе вы узнаете, как работать с компонентами.

### Контрольные вопросы

1. В каком смысле используется термин «проект» в программировании?
2. Из каких файлов состоит типичный проект в RAD-среде?
3. Что такое форма? Почему для описания формы в *Visual Studio* используются два файла?
4. Какие основные окна используются в среде *Visual Studio*? Зачем они нужны?
5. Покажите, что программа, написанная с помощью *Visual Studio*, состоит из объектов.
6. Где расположена основная программа в проекте *Visual Studio*? Объясните текст основной программы.
7. Почему в основной программе не виден цикл обработки сообщений?
8. Назовите некоторые важнейшие свойства формы. Какими способами можно их изменять?
9. Приведите примеры автоматического построения и изменения кода в RAD-среде.
10. Как создать новый обработчик события? Подумайте, можно ли сделать это вручную.
11. Как передаются параметры сообщения в обработчик?
12. Как можно вывести сообщение об ошибке на экран?

### Задачи

1. Попробуйте изменять какие-нибудь свойства формы, построив обработчик еще одного события (например, **`Shown`** – вывод формы на экран; **`Click`** – щелчок мыши; **`Resize`** – изменение размеров).

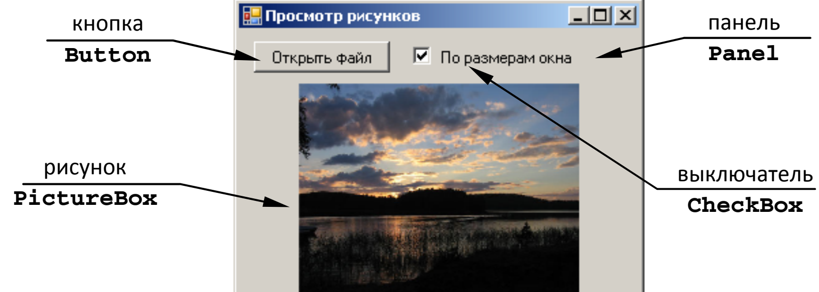
# § 53. Использование компонентов

## Программа с компонентами

В *Visual Studio* существует так называемая *Панель элементов* – библиотека готовых объектов (*компонентов*), которые можно добавить в свою программу, просто перетащив их мышкой на форму.

Компоненты разбиты на группы. Мы будем использовать компоненты из групп *Стандартные элементы*, *Контейнеры* и *Диалоговые окна*. Если задержать мышку над значком компонента, в тексте всплывающей подсказки можно прочитать его краткое описание.

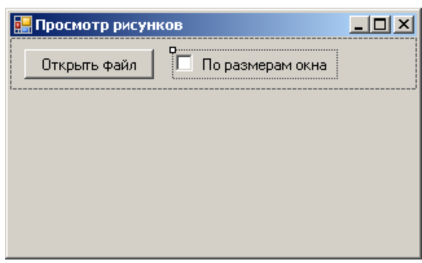
Построим простую программу для просмотра рисунков, используя готовые компоненты. В верхней части на панели разместим кнопку для загрузки файла и флажок-выключатель, который изменяет масштаб рисунка так, чтобы он вписывался в отведенное ему место.




Создадим новый проект (как в предыдущем параграфе), изменим имя формы (свойство **`Name`**) на **`MainForm`**, а её заголовок (свойство **`Text`**) – на *Просмотр рисунков.*

Добавим на форму панель – компонент *Panel* из группы *Контейнеры*. Для этого можно перетащить эту кнопку на форму или щелкнуть по кнопке и нарисовать прямоугольник, ограничивающий панель. Теперь размеры панели можно изменять, перетаскивая маркеры на границах или изменяя значения свойств **`Width`** (ширина) и **`Height`** (высота) на панели *Свойства*. Панель можно перетаскивать мышкой по форме за значок . Хотелось бы, чтобы панель была всё время

прижата к верхней границе окна и её размеры изменялись бы вместе с размерами окна. Для этого нужно установить свойство **Dock** (пристыковать) равным **Top** (англ. *top* – верх).




Теперь панель готова, на ней нужно разместить кнопку (компонент **Button**) и выключатель (компонент **CheckBox**). На кнопке должна быть надпись *Открыть файл* (свойство **Text**), а справа от выключателя – текст *По размерам окна* (тоже свойство **Text**). Размеры и расположение компонентов нужно поменять с помощью мыши. Имена компонентов (свойства **Name**) тоже можно изменить, например, на **OpenBtn** и **SizeCb**.

Если теперь заглянуть внутрь файла **Form1.Designer.cs**, мы увидим, что в состав класса **MainForm** добавлены новые элементы:

```
private System.Windows.Forms.Panel panel1;
private System.Windows.Forms.Button OpenBtn;
private System.Windows.Forms.CheckBox SizeCB;
```

а их свойства устанавливаются в процедуре **InitializeComponent**.

Компоненты на форме связаны отношениями родитель – потомок. Главный объект – это сама форма **MainForm**, она является *родительским объектом* для панели **panel1**. Это означает, что при перемещении формы панель перемещается вместе с ней. Кроме того, форма отвечает за прорисовку всех дочерних элементов на экране. В свою очередь, панель – это родительский объект для кнопки и выключателя.

Теперь добавим на форму специальный объект **PictureBox**, который «умеет» отображать рисунки различных форматов. Для того, чтобы он заполнял все свободное пространство (кроме панели), нужно установить для него выравнивание **Fill** (свойство **Dock**). Изменим название объекта на **Img**.

Остаётся решить два вопроса:

1) как сделать выбор файла и загрузку его в компонент **Img**;
2) как подгонять размер рисунка по размеру формы.

К счастью, для этого достаточно использовать возможности готовых компонентов. В группе *Диалоговые окна* есть готовый компонент для выбора файла на диске, он называется **OpenFileDialog**. У него есть метод **ShowDialog** – функция, которая вызывает стандартный диалог выбора файла и возвращает значение **DialogResult.OK**, если файл успешно выбран. Имя выбранного файла можно получить, прочитав свойство **FileName** этого компонента.

Добавим компонент **OpenFileDialog** на форму (в любое место). Это невизуальный компонент, его не будет видно во время выполнения программы, поэтому он не добавляется на форму, а будет виден в нижней части окна конструктора. Для краткости изменим его название на **OpenDlg** и очистим свойство **FileName** (имя файла по умолчанию). В свойство **Filter** запишем такую строку:

```
"Файлы с рисунками|*.jpg;*.jpeg;*.gif;*.bmp"
```

Это фильтр, определяющий условие отбора файлов (остальные файлы будут скрыты). В данном случае нас интересуют только файлы с указанными четырьмя расширениями, слева от символа «|» записано название этого фильтра.

Теперь в случае щелчка по кнопке нужно вызвать метод **OpenDlg.ShowDialog** и, если он вернет значение **DialogResult.OK**, загрузить выбранный файл, имя которого получается как значение свойства **OpenDlg.FileName**. Щелчок по кнопке – это событие, обработчик которого называется **Click**. Создадим шаблон этого обработчика двойным щелчком на странице *События* (в окне *Свойства*) и запишем в него команды

```
if ( OpenDlg.ShowDialog() == DialogResult.OK )
  Img.Image = new Bitmap ( OpenDlg.FileName );
```

Поясним загрузку файла. Компонент **Img** имеет свойство **Image**, в котором хранится изображение. Это указатель на объект-рисунок в памяти, который создаётся с помощью оператора **new** из файла, выбранного пользователем. Теперь можно запустить программу и проверить, как она работает.

Обратите внимание, что изображение выводится в масштабе 1:1 независимо от размера окна. Выключатель *По размерам окна* можно включать и выключать, но он никак не влияет на результат. Чтобы исправить ситуацию, будем использовать событие изменения состояния выключателя, его обработчик называется **CheckedChanged** (изменение свойства **Checked**, отмеченный). У объекта **Img** есть свойство **SizeMode** (режим подгонки размера), если ему присвоить значение **Zoom**, компонент **Img** сам выполнит подгонку размеров рисунка под размер свободной области (по умолчанию установлен режим **Normal**). Таким образом, обработчик события **CheckedChanged** компонента **SizeCb** содержит такой оператор

```
if ( SizeCB.Checked )
      Img.SizeMode = PictureBoxSizeMode.Zoom;
else  Img.SizeMode = PictureBoxSizeMode.Normal;
```

Свойство **Checked** (англ. *отмечен*) – это логическое значение, определяющее состояние выключателя: если он включен, это свойство равно **true**, если выключен, то **false**. Режимы **Zoom** и **Normal** относятся к стандартному перечислимому типу **PictureBoxSizeMode**.

Теперь можно запустить готовую программу и проверить ее работу. Отметим следующие важные особенности:

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

## Ввод и вывод данных

Во многих программах нужно, чтобы пользователь вводил текстовую или числовую информацию. Чаще всего для ввода данных применяют поле ввода – компонент **TextBox**. Для доступа к введённой строке используют его свойство **Text** (англ. *текст*).

Шрифт текста в поле ввода задается сложным свойством **Font** (англ. *шрифт*). Это объект, у которого есть свои свойства, их список можно увидеть, если щелкнуть по значку ⊞ слева от названия свойства. Например, подсвойство **Size** – размер шрифта в пунктах, а логические значения **Bold** (жирный), **Italic** (курсив) и **Underline** (подчеркнутый) определяют стиль шрифта. Если установить шрифт для какого-то объекта, например, для формы, все дочерние компоненты по умолчанию будут иметь такой же шрифт.

Программа, которую мы сейчас построим, будет переводить RGB-составляющие 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 **Panel**), цвет которого изменяется согласно введенным значениям;
- несколько меток (компонентов **Label**).

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

Для того, чтобы пользователь не мог изменять размеры окна программы (в данной задаче это ни к чему!), мы установим свойство формы **FormBorderStyle** (англ. *стиль границы формы*), равное **FixedSingle** (англ. *фиксированная, одиночная*).

Во время работы программы будут использоваться поля ввода **rEdit, gEdit** и **bEdit**, метка **rgbLabel**, с помощью которой будет выводиться результат – код цвета, и панель **rgbPanel**. В качестве начальных значений полей 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 заголовок (свойство **Text**) метки **rgbLabel**, а также изменить цвет заливки для панели **rgbPanel**. Обработчик события, которое происходит при изменении текста в поле ввода, называется **TextChanged**. Так как при изменении любого из трех полей нужно выполнить одинаковые действия, для этих компонентов можно установить один и тот же обработчик. Для этого нужно выделить их, удерживая клавишу *Shift*, и после этого создать новый обработчик двойным щелчком на странице *События* в окне *Свойства.*

Обработчик события **TextChanged** для полей ввода может выглядеть так:

```
private void rEdit_TextChanged ( object sender, EventArgs e )
{
  int r, g, b;
  r = int.Parse ( rEdit.Text );
  g = int.Parse ( gEdit.Text );
  b = int.Parse ( bEdit.Text );
  rgbPanel.BackColor = Color.FromArgb ( r, g, b );
  rgbLabel.Text = "#" + r.ToString ( "X2" )
                  + g.ToString ( "X2" ) + b.ToString ( "X2" );
}
```

Сначала значения составляющих цвета переводятся из символьного вида в числовой и записываются в переменные **r**, **g** и **b**. Для этого используется метод **Parse** класса **int** (целые числа).

Панель имеет свойство **BackColor**, которое определяет заливку внутренней области. Мы строим цвет заливки по значениям составляющих RGB-модели с помощью метода **FromArgb** класса **Color**.

В последней строчке формируется строка, содержащая шестнадцатеричный код цвета. Для перевода значений в шестнадцатеричную систему используется метод **ToString**, второй параметр «X2» указывает на то, что число нужно записать в шестнадцатеричной системе с двумя знаками.

Вы можете заметить, что при запуске программы код цвета и цвет прямоугольника не изменяются, какие бы значения мы ни установили в полях ввода. Чтобы исправить ситуацию, нужно вызвать уже готовый обработчик из обработчика события **Load** формы (он вызывается после создания формы, но до показа её на экране):

```
private void MainForm_Load ( object sender, EventArgs e )
{
  rEdit_TextChanged ( rEdit, e );
}
```

При вызове в скобках указан объект, который посылает сообщение о событии. Здесь в качестве источника указан компонент **rEdit**, но в данном случае можно было использовать любой объект (а также нулевой объект **null**), потому что параметр **sender** в обработчике **TextChanged** не используется. В нашем случае обработчик можно было вызвать и так:

```
rEdit_TextChanged ( null, null );
```

## Обработка ошибок

Если в предыдущей программе пользователь введет не числа, а что-то другое (или пустую строку), программа выдаст сообщение о необработанной ошибке и предложит завершить 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 – catch**:

```
try
{
  // «опасные» команды
}
catch
{
  // обработка ошибки
}
```

Слово **try** по-английски означает «попытаться», **catch** — «поймать». В данном случае мы ловим *исключения* (исключительные или ошибочные, непредвиденные ситуации). Программа попадает в блок **catch** только тогда, когда между **try** и **catch** произошла ошибка.

В нашей программе «опасные» команды – это операторы преобразования данных из текста в числа (вызовы метода **int.Parse**). В случае ошибки мы выведем вместо кода цвета знак вопроса. Улучшенный обработчик с защитой от неправильного ввода принимает вид

```
try
{
  r = int.Parse ( rEdit.Text );
  g = int.Parse ( gEdit.Text );
  b = int.Parse ( bEdit.Text );
  rgbPanel.BackColor = Color.FromArgb ( r, g, b );
  rgbLabel.Text = "#" + r.ToString ( "X2" )
                  + g.ToString ( "X2" ) + b.ToString ( "X2" );
}
catch
{
  rgbLabel.Text = "?";
}
```

Существует и другой способ защиты от неверных входных данных – блокировать при вводе символы, которых быть не должно (буквы, скобки и т.п.). В нашей программе для всех полей ввода можно установить такой обработчик события **KeyPress** (англ. *при нажатии клавиши*):

```
private void rEdit_KeyPress ( object sender, KeyPressEventArgs e )
{
  if ( ! ( Char.IsDigit(e.KeyChar) || e.KeyChar == (char) 8) )
       e.Handled = true;
}
```

Этому обработчику передается блок данных **KeyPressEventArgs** с именем **e**, в составе которого есть изменяемый параметр **KeyChar** – символ, соответствующий нажатой клавише. Если этот символ не входит в допустимый набор (цифры и клавиша *BackSpace*, имеющая код 8), свойство **Handled** (англ. *обработано*) изменяемого параметра **e** устанавливается равным **true**. В этом случае стандартная обработка нажатия на клавишу отключается (ведь мы сказали, что событие уже обработано!) и символ просто игнорируется. Принадлежность символа к цифрам определяется с помощью метода **Char.IsDigit**, который возвращает логическое значение.

## ? *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 **Dock** в размещении элементов на форме.
5. Что такое стандартный диалог? Как его использовать?
6. Назовите основное свойство выключателя. Как его использовать?
7. Расскажите о сложных свойствах на примере свойства **Font**.

8. Какой шрифт устанавливается для компонента по умолчанию?
9. Что такое метка?
10. Зачем используются методы **`int.Parse`** и **`Color.FromArgb`**?
11. Как обрабатываются ошибки в современных программах? В чем, на ваш взгляд, преимущества и недостатки такого подхода?
12. Как при вводе блокировать некорректные символы?

### *Задачи*

1. Добавьте в программу для построения RGB-кода цвета защиту от ввода слишком больших чисел (больших, чем 255).
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. Стандартный компонент **`TextBox`** разрешает вводить любые символы и представляет результат ввода как текстовое свойство **`Text`**. Поэтому для того, чтобы получить нужное нам поведение (ввод целых чисел), мы

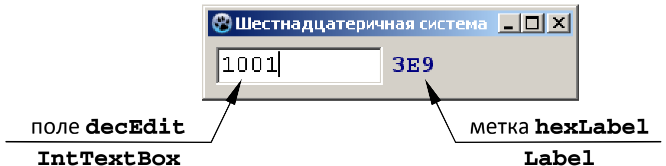
- добавили обработчик **`KeyPress`**, заблокировав ошибочные символы;
- для перевода текстовой строки в число каждый раз использовали метод **`int.Parse`**.

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 **`TextBox`** (поле ввода), поэтому, согласно принципам ООП, наш компонент (назовём его **`IntTextBox`**) будет наследником класса **`TextBox`**, а класс **`TextBox`** будет соответственно базовым классом для нового класса **`IntTextBox`**. Изменения стандартного класса **`TextBox`** сводятся к двум пунктам:

- все некорректные символы (кроме цифр и кода клавиши *BackSpace*) должны блокироваться автоматически, без установки дополнительных обработчиков событий;
- компонент должен уметь сообщать числовое значение; для этого мы добавим к нему свойство **`Value`** (англ. *значение*) целого типа.

Начнём новый проект – программу, которая будет переводить целые числа из десятичной системы в шестнадцатеричную:



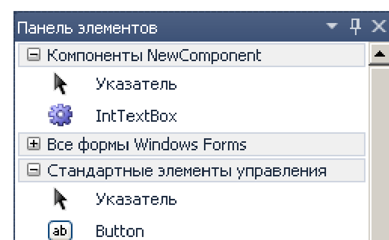
Так как нет никакого смысла изменять размеры этого окна, запретим это: установим свойство формы **FormBorderStyle** равное **FixedSingle**.

Сначала создадим пустой компонент – наследник от **TextBox**. Для этого нужно добавить к проекту новый класс (меню *Проект – Добавить класс*). После этого в окне обозревателе решений появится новый исходный файл – **Class1.cs**. Откройте этот файл и измените название класса, а также укажите, что он является наследником класса **TextBox**:

```
class IntTextBox: TextBox
{
}
```

Для того, чтобы программа прошла трансляцию, в начало модуля нужно добавить еще одну системную библиотеку, содержащую описание базового класса **TextBox**:

```
using System.Windows.Forms;
```




Теперь программу можно запустить. После запуска программы в верхней части окна *Панель элементов* появляется новый компонент **IntTextBox** который уже можно использовать. Правда, пока они ничем не отличается от компонента **TextBox**.

Начнём изменять новый компонент. Сначала добавим в описание класса обработчик события **KeyPress**, который блокирует нежелательные символы:

```
protected override void OnKeyPress ( KeyPressEventArgs e )
{
  if ( !(Char.IsDigit(e.KeyChar) || e.KeyChar == (char)8) )
     e.Handled = true;
  base.OnKeyPress(e);
}
```

Слово **override** указывает на то, что мы перекрываем метод базового класса с таким же именем, а описатель **protected** говорит о том, что этот метод будет доступен всем наследникам класса **IntTextBox**. В последней строчке вызывается соответствующий метод базового класса для того, чтобы не отключить действия, который там выполняются (и о которых мы не знаем!).

Теперь создадим (тоже внутри описания класса) свойство **Value** целого типа:

```
public int Value
{
  set { Text = value.ToString(); }
  get {
    try { return int.Parse ( Text ); }
    catch { return 0; }
    }
}
```

В методе **set** (он используется для записи нового значения свойства) мы просто преобразуем в строку переданное значение (оно всегда называется **value**, с маленькой буквы) и записываем его в свойство **Text** нашего компонента. В методе **get** (чтение значения свойства) пытаемся преобразовать строку в целое число[10] и в случае неудачи возвращаем 0.

Таким образом, описание нового класса выглядит так:

```
class IntTextBox: TextBox
{
  protected override void OnKeyPress(KeyPressEventArgs e)
  {
```

[10] Здесь можно использовать также метод **int.TryParse**, который позволяет обнаружить ошибку при таком преобразовании без использования исключений. Его описание вы можете найти в литературе или в Интернете.

```
    if ( !(Char.IsDigit(e.KeyChar) || e.KeyChar == (char)8) )
      e.Handled = true;
    base.OnKeyPress ( e );
  }
  public int Value
  {
    set { Text = value.ToString(); }
    get {
      try { return int.Parse(Text); }
      catch { return 0; }
      }
  }
}
```

Теперь можно использовать готовый компонент. Поместим на форму компонент **decEdit** типа **IntTextBox** и метку **hexLabel** для вывода шестнадцатеричного значения.

Остается определить для нового компонента обработчик события **TextChanged** – при изменении содержимого поля ввода нужно показать соответствующее шестнадцатеричное число с помощью метки **hexLabel**. Добавим к полю ввода такой обработчик события **TextChanged**:

```
private void decEdit_TextChanged ( object sender, EventArgs e )
{
  hexLabel.Text = decEdit.Value.ToString ( "X" );
}
```

Сначала мы запрашиваем числовое (!) значение у поля ввода, используя новое свойство **Value**, а затем переводим его в шестнадцатеричную систему счисления с помощью метода **ToString**. Теперь программа готова и ее можно использовать.

### 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 **IntTextBox** и **TextBox**. Чем они отличаются?
5. Как преобразовать числовое значения в текстовое и обратно?
6. Какой метод применяется для перевода числа в шестнадцатеричную систему счисления?
7. Объясните, как работает свойство **Value** у компонента **IntTextBox**?
8. Почему в приведенном примере для обработки вводимых символов мы не устанавливали свой обработчик события **KeyPress**?
9. Что означают слова **protected** и **override** при описании метода?

### Задачи

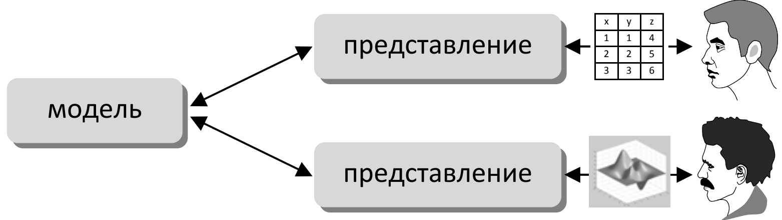
1. Измените программу так, чтобы в самом начале метка показывала шестнадцатеричный код числа, которое записано в поле ввода.
2. Разработайте компонент, который позволяет вводить шестнадцатеричные числа.
3. *Используя дополнительные источники, разберитесь, как установить новый компонент в палитру среды *Lazarus*. Переделайте свою программу так, чтобы компонент добавлялся на форму из палитры.

## § 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 зависимости, 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**.

```
             k
             ↓
  22 + 13    -   3 * 8
    n1            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 )
  результат = перевести всю строку в число
else
  {
  n1 = результат вычисления левой части
  n2 = результат вычисления правой части
```

```
  результат = применить найденную операцию к n1 и n2
  }
```

Для того, чтобы выделить модель в отдельный модуль (исходный файл), создадим новый класс (см. предыдущий параграф) и назовём его **Calculator**:

```
static class Calculator
{
}
```

Слово **static** означает, что это статический класс: он содержит только методы, создать экземпляр такого класса нельзя. Фактически это набор функций и процедур. В этот класс мы поместим все функции, которые используются для вычисления выражения, записанного в символьной строке:

```
static class Calculator
{
  static int Priority ( char op )  {
  ...
  }
  static int LastOp ( string s ) {
  ...
  }
  public static int Calc ( string s ) {
  ...
  }
}
```

Функции **Priority** и **LastOp** мы уже использовали в главе 6 (на языке С++). Первая из них возвращает приоритет арифметической операции, а вторая ищет в строке символ, соответствующий последней выполняемой операции, и возвращает его номер. Перед всеми функциями стоит описатель **static**, поскольку все методы статического класса **Calculator** должны быть статическими. Кроме того, метод **Calc** должен быть доступен извне класса (например, в методах формы), поэтому перед ним стоит также описатель **public**. Остальные два метода – закрытые.

Функцию **LastOp** придётся немного переписать (с языка С++ на C#)

```
static int LastOp(string s)
{
  int i, minPrt, res;
  minPrt = 50;
  res = -1;
  for ( i = 0; i < s.Length; i++ )
    if ( Priority(s[i]) <= minPrt )
      {
      minPrt = Priority(s[i]);
      res = i;
      }
  return res;
}
```

Сделано два изменения: во-первых, параметр функции – строка типа **string**, а не массив символом; во-вторых, длина строки определяется с помощью свойства **Length**.

Итак, 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**:

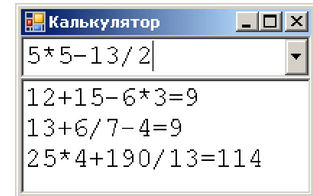
```
public static int Calc(string s)
{
  int k, n1, n2, res = 0;
  k = LastOp ( s );
  if ( k < 0 ) return int.Parse(s);
  n1 = Calc( s.Substring(0, k) ); // левая часть
```

```
   n2 = Calc( s.Substring(k+1) ); // правая часть
   switch ( s[k] ) {
     case '+': res = n1 + n2; break;
     case '-': res = n1 - n2; break;
     case '*': res = n1 * n2; break;
     case '/': res = n1 / n2; break;
     }
   return res;
 }
```

Левая и правая части выражения (слева и справа от знака последней операции) определяются с помощью метода **Substring** класса **String**. В первом случае этот метод вызывается с двумя аргументами: первый обозначает начальный номер символа (0 – с начала строки), а второй – длину подстроки. Во втором случае используется только один аргумент – начальный номер символа, и метод возвращает все символы до конца строки. Обратите внимание, что функция **Calc** – рекурсивная, она дважды вызывает сама себя.

Таким образом, наша модель – это класс **Calculator**, с помощью которого вычисляется арифметическое выражение, записанное в строке.

## Вычисление арифметических выражений: представление




Теперь построим интерфейс программы. В верхней части окна будет размещен выпадающий список (компонент **ComboBox**), в котором пользователь вводит выражение. При нажатии на клавишу *Enter* выражение вычисляется и его результат выводится в последней строке многострочного редактора текста (компонента **TextBox** с установленным свойством **Multiline**). Список полезен для того, чтобы можно было вернуться к уже введенному ранее выражению и исправить его. Для этого каждое новое выражение будем добавлять в выпадающий список.

Итак, на форму нужно добавить компонент **ComboBox**. Чтобы прижать его к верху, установим свойство **Dock**, равное **Top**. Назовем этот компонент **Input** (англ. *ввод*).

Добавляем второй компонент – **TextBox**, устанавливаем для него выравнивание **Fill** (заполнить всю свободную область), имя **Answers** (англ. *ответы*) и свойство **Multiline** равное **True** («да», это многострочный редактор).

Для того, чтобы пользователь не мог менять поле вывода, для компонента **Answers** устанавливаем логическое свойство **ReadOnly** (англ. *только для чтения*), равное **True** (да). При этом фон компонента становится серым, если вам это не нравится, нужно изменить его свойство **BackColor**.

Логика работы программы может быть записана в виде псевдокода:

```
if ( нажата клавиша Enter )
  {
  x = значение выражения
  добавить результат вычислений в конец поля вывода
  if ( выражения нет в списке )
    добавить его в список
  }
```

Для перехвата нажатия клавиши *Enter* будем использовать обработчик **KeyPress** компонента **Input**. Клавиша *Enter* имеет код 13, поэтому условие «если нажата клавиша *Enter*» запишется в виде

```
if ( e.KeyChar == (char)13 )
{
...
}
```

Значение выражения будем вычислять с помощью функции **Calc**:

```
int x = Calculator.Calc(Input.Text);
```

Эта функция принадлежит классу **Calculator**, поэтому перед названием функции записано ещё и имя класса.

Содержимое многострочного компонента **TextBox** хранится как свойство **Text**, для разбивки на строки используется сочетание двух служебных символов: «**\r**» (перевод строки) и «**\n**» (возврат каретки в начало строки). Поэтому дописать введённое выражение и результат его вычисления в конец текста в виде отдельной строки можно так:

```
Answers.Text += Input.Text + "=" + x.ToString() + "\r\n";
```

Обратите внимание, что результат вычислений переведен в символьный вид с помощью метода **ToString**.

Строки, входящие в выпадающий список, доступны как свойство-массив **Items** объекта **Input**. Метод **FindString** служит для поиска строки в списке и возвращает номер найденного элемента (нумерация начинается с нуля) или значение –1, если образец не найден. Поэтому команда добавления в список новой строки выглядит так:

```
int i = Input.FindString(Input.Text);
if ( i < 0 )
  Input.Items.Insert ( 0, Input.Text );
```

Метод **Insert** добавляет строку в указанное место списка. На первом месте записывается позиция, в которую добавляется строка (0 – в начало списка). Приведём полностью обработчик **KeyPress**:

```
private void Input_KeyPress ( object sender, KeyPressEventArgs e )
{
  if ( e.KeyChar == (char)13 )
    {
    int x = Calculator.Calc ( Input.Text );
    Answers.Text += Input.Text + "=" + x.ToString() + "\r\n";
    int i = Input.FindString(Input.Text);
    if ( i < 0 )
      Input.Items.Insert ( 0, Input.Text );
    }
}
```

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.

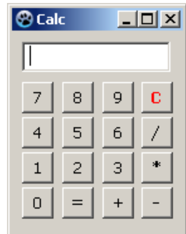
Часто к паре «модель-представление» добавляют еще управляющий блок (контрóллер), который, например, обрабатывает ошибки ввода данных. Но при программировании в RAD-средах контроллер и представление, как правило, объединяются вместе – контроль данных происходит в обработчиках событий.

## ? *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 метод **`double.Parse`**, аналогичный **`int.Parse`**.
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).