К. Ю. Поляков

ПРОГРАММИРОВАНИЕ

ПРОФИЛЬНАЯ ШКОЛА

Python

C++

Часть 4

ИЗДАТЕЛЬСТВО БИНОМ

К. Ю. Поляков

ПРОГРАММИРОВАНИЕ
Python
C++

Часть 4

Учебное пособие
для общеобразовательных
организаций

Москва
БИНОМ. Лаборатория знаний
2019

УДК 004.9
ББК 32.97
П54

Поляков К. Ю.

П54 Программирование. Python. C++. Часть 4 : учебное пособие / К. Ю. Поляков. — М. : БИНОМ. Лаборатория знаний, 2019. — 192 с. : ил.

ISBN 978-5-9963-4137-5

Книга представляет собой завершающую, четвёртую часть серии учебных пособий по программированию. В отличие от большинства аналогичных изданий, в ней представлены два языка программирования высокого уровня — Python и C++.

Главные темы пособия — объектно-ориентированное программирование и создание программ с графическим интерфейсом. Изучаются основные принципы объектного подхода к созданию программ: абстракция, инкапсуляция, наследование, полиморфизм. Изложение ведётся на примерах программирования игр, в которых моделируются системы взаимодействующих объектов.

Для демонстрации возможностей сред быстрой разработки программ в последней части пособия рассмотрены примеры приложений на языке C#.

После каждого параграфа приводится большое число заданий для самостоятельного выполнения разной сложности и вариантов проектных работ.

Пособие предназначено для учащихся средних школ.

УДК 004.9
ББК 32.97

Учебное издание

Поляков Константин Юрьевич

ПРОГРАММИРОВАНИЕ
Python. C++
Часть 4
Учебное пособие

Ведущий редактор *О. А. Полежаева*
Концепция внешнего оформления *В. А. Андрианов*
Художественный редактор *Н. А. Новак*
Технический редактор *Е. В. Денюкова*
Корректор *Е. Н. Клитина*
Компьютерная верстка: *В. А. Носенко*

(12+)

Подписано в печать 06.09.2018. Формат 84x108/16. Усл. печ. л. 20,16
Тираж 3 000 экз. Заказ № м7023.

ООО «БИНОМ. Лаборатория знаний»
127473, Москва, ул. Краснопролетарская, д. 16, стр. 3,
тел. (495)181-53-44, e-mail: binom@Lbz.ru
http://Lbz.ru, http://metodist.Lbz.ru

Отпечатано в филиале «Смоленский полиграфический комбинат»
ОАО «Издательство «Высшая школа». 214020, Смоленск, ул. Смольянинова, 1
Тел.: +7 (4812) 31-11-96. Факс: +7 (4812) 31-31-70
E-mail: spk@smolpk.ru http://www.smolpk.ru

ISBN 978-5-9963-4137-5

ПРЕДИСЛОВИЕ

Перед вами — последняя, четвёртая часть учебного пособия по программированию на языках Python и C++. Она посвящена очень важному подходу к разработке программ — объектно-ориентированному программированию (ООП). Именно переход к ООП в конце XX века позволил повысить скорость разработки и надёжность больших и сложных программ, код которых состоит из сотен тысяч и даже миллионов строк. Большинство современных профессиональных программ построены на принципах ООП.

С точки зрения ООП, вся программа — это множество объектов, взаимодействующих между собой. Каждый объект относится к некоторому классу — группе объектов с общими свойствами. Класс объединяет вместе данные и методы их обработки. Более того, объекты, как правило, «не разрешают» другим объектам работать со своими данными, и это уменьшает вероятность ошибок.

Создание объектно-ориентированной программы начинается с анализа: нужно выделить в задаче объекты, описать их данные (свойства) и методы управления этими данными, т. е. те команды, на которые будет «откликаться» объект. И только после этого приступают к кодированию алгоритмов на каком-либо объектно-ориентированном языке.

Мы будем рассматривать все идеи ООП на практических примерах, в том числе применим эти идеи для программирования игр. С помощью пособия вы сможете построить «каркас» игры, который потом несложно будет доработать так, как вам захочется.

Сначала, как и в предыдущих частях пособия, мы рассмотрим язык Python. Он использует «неклассическую» модель работы с объектами, которая позволяет полностью управлять каждым объектом из любой точки программы. В то же время злоупотребление этой свободой может привести к неожиданным и трудно обнаруживаемым ошибкам.

Во второй главе мы продолжим изучать язык C++. Объектно-ориентированные возможности C++ — это классика программирования и обязательный материал, которым должен владеть любой профессиональный разработчик. После Python вам может показаться, что в C++ слишком много ограничений. Но на самом деле разработчики языка сделали всё возможное, чтобы у вас было как можно меньше шансов сделать ошибку, которая может привести к серьёзному сбою в работе программы. Все возможности языка C++, описанные в пособии, соответствуют стандарту C++11.

В наше время самый ценный ресурс — это время работы программиста. Для того чтобы сократить сроки разработки сложных программ с графическим интерфейсом, применяют среды быстрой разработки (англ. *RAD*: *Rapid Application Development*).

Одна из популярных профессиональных RAD-сред, которые можно использовать бесплатно, — это *Visual Studio Community*. В ней можно программировать на разных языках; в конце пособия мы создадим несколько оконных приложений на языке C#, который очень похож на C++.

Язык C# — это современный объектно-ориентированный язык программирования общего назначения. На нём можно разрабатывать различные программы для операционной системы Windows, веб-приложения, программы для мобильных устройств и игровых консолей.

Вы увидите, насколько удобно строить весь интерфейс программы на C# с помощью мыши, расставляя в окне программы элементы управления: кнопки, поля ввода, текстовые метки, списки и т. п. Программисту остаётся только определить, как программа будет реагировать на действия пользователя, например на щелчок мышью по кнопке.

Дополнительные материалы к пособию, в том числе файлы с программами, можно загрузить с сайта автора:

http://kpolyakov.spb.ru/school/pycpp.htm.

Автор хочет поблагодарить своих коллег за внимательное чтение рукописи этого пособия и полезные критические замечания, которые позволили устранить неточности и существенно улучшить содержание:

- *Д. В. Богданова*, старшего преподавателя кафедры бизнес-информатики МЭИ;
- *М. Д. Полежаеву*, разработчика веб-сайтов.

Глава 1
ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ PYTHON

§ 1
Что такое ООП?

Ключевые слова:

- декомпозиция
- объект
- абстракция
- процедурное программирование
- объектно-ориентированное программирование

Проблема сложности программ

Во второй половине XX века компьютеры стали применять для моделирования сложных систем, как технических (системы связи), так и информационных (склады, банки, системы бронирования билетов и др.). Новые задачи требовали более сложных алгоритмов, размеры программ выросли до десятков и сотен тысяч строк кода, число переменных и массивов измерялось в тысячах.

Программисты столкнулись с проблемой сложности, которая превысила возможности человеческого разума. Один человек уже неспособен написать надёжно работающую серьёзную программу, так как не может «охватить взглядом» все её детали. Поэтому в разработке современных программ, как правило, 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 ещё 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.

Процедурное программирование

Сначала возникла идея разделить программу на небольшие самостоятельные части (*подпрограммы, процедуры*), каждая из которых решает отдельную часть общей задачи. Такой подход получил название *процедурного программирования*.

Поскольку каждая подпрограмма (процедура, функция) решает самостоятельную задачу, можно поручить разработку и отладку подпрограмм разным программистам и таким образом разделить работу меж-

ду ними. Программисту, который отлаживает какую-то подпрограмму, совершенно не обязательно знать все детали работы остальных подпрограмм. Ему нужно только обеспечить согласованный со всеми *интерфейс* — способ связи «его» подпрограммы с другими. Интерфейс подпрограммы определяет:

- какие данные принимает подпрограмма от других подпрограмм;
- какие данные передаёт подпрограмма другим подпрограммам.

В остальном для программиста «чужие» подпрограммы — это «чёрные ящики», которые решают свои задачи.

Процедурное программирование во многом позволило справиться с проблемой сложности программ. Но оно не решило ещё одну проблему: данные и методы их обработки остались разделены. В большинстве программ использовались глобальные переменные и массивы, которые описывали состояние программы. Любая процедура и функция могли изменить глобальные данные. Это приводило к трудноуловимым ошибкам и снижало надёжность программ. Проблема становилась всё более острой с увеличением объёма программ, которые нередко состояли из сотен тысяч строк кода.

Объектный подход

Следующая замечательная идея появилась в конце 60-х годов XX века — применить в разработке программ тот подход, который использует человек в повседневной жизни.

Как отмечал ещё в XVII веке французский математик и философ Рене Декарт, люди воспринимают мир как множество объектов: предметов, животных, людей. Все объекты имеют внутреннее устройство и состояние, свойства (внешние характеристики) и поведение. Чтобы справиться со сложностью окружающего мира, люди часто игнорируют внутреннее строение объектов, обращая внимание только на те свойства, которые необходимы для решения их практических задач. Такой приём называется *абстракцией*.

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

Для разных задач существенные свойства могут быть совершенно разными. Например, услышав слово «кошка», многие подумают о пушистом усатом животном, которое мурлычет, когда его гладят. В то же время ветеринарный врач представляет скелет, ткани и внутренние органы кошки, которую ему нужно лечить. Для изготовителей кошачьего корма кошка — это объект, который потребляет корм и, таким образом, приносит им доход. В каждом из этих случаев применение абстракции даёт разные модели одного и того же объекта, отражающие его различные свойства и цели моделирования.

Как применить принцип абстракции в программировании? Формулировки задач, решаемых на компьютерах, всё более приближаются к формулировкам реальных жизненных задач. Поэтому возникла такая идея: представить программу в виде множества объектов, каждый из которых обладает своими свойствами и поведением.

Построенная таким образом модель задачи называется *объектной*. Решение задачи сводится к моделированию взаимодействия объектов. Здесь тоже применяется принцип «разделяй и властвуй». Проектирование идёт «сверху вниз», только не по алгоритмам (процедурам и функциям), как в процедурном программировании, а по объектам.

Взаимодействие объектов

Каждый объект «закрыт» в том смысле, что его внутреннее устройство другим объектам неизвестно. Но объекты могут обмениваться данными с другими объектами, используя заранее согласованные каналы связи (рис. 1.1, *а*).

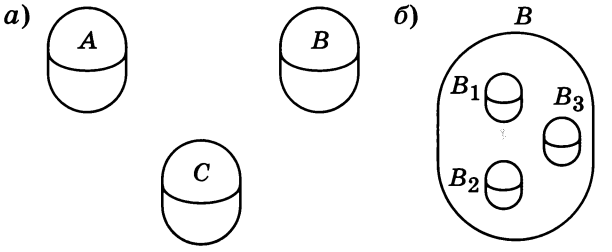


Рис. 1.1

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, т. е. сознательно отбрасываем второстепенные детали.

Если всё-таки «заглянуть внутрь» такого объекта, часто можно обнаружить, что он, в свою очередь, тоже состоит из объектов (рис. 1.1, *б*). Но важно, что объект сам работает со своими внутренними данными, не допуская к ним другие объекты. Таким образом, данные и методы их обработки объединены — вместе они составляют объект.

Объект — это данные плюс методы их обработки.

Итак, сначала мы должны построить объектную модель задачи — выделить объекты и определить правила обмена данными между ними. После этого можно поручить разработку модели каждого объекта отдельному программисту, который должен решить (и запрограммировать), как именно объект выполняет свои функции. Программисту не обязательно держать в голове полную информацию обо всех остальных 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, называют объектно-ориентированным программированием (ООП). Более строгое определение мы дадим немного позже.

Выводы

- Программисты столкнулись с проблемой сложности программ, которая превысила возможности человеческого разума.
- Процедурное программирование (декомпозиция по алгоритмам) — это пример использования принципа «разделяй и властвуй» для борьбы со сложностью программ.
- Абстракция — это выделение характеристик объекта, существенных в данной задаче и отличающих этот объект от других объектов.
- Объект — это данные плюс методы их обработки.
- Объектный подход — это декомпозиция задачи по объектам.
- Программирование, основанное на моделировании задачи реального мира как множества взаимодействующих объектов, называют объектно-ориентированным программированием (ООП).

Вопросы и задания

1. Почему со временем неизбежно изменяются методы программирования?
2. Что такое декомпозиция, зачем она применяется?
3. Какие виды декомпозиции в программировании вы знаете?
4. Что такое процедурное программирование? Какой вид декомпозиции в нём используется?
5. Какие проблемы в программировании привели к появлению ООП?
6. Что такое абстракция? Зачем она используется в обычной жизни?
7. Объясните, почему один и тот же объект может иметь много разных моделей. Какую роль здесь играет абстракция?
8. Какой вид декомпозиции используется в ООП?
9. Какие преимущества даёт объектный подход в программировании?
10. Что такое интерфейс? Приведите примеры объектов, у которых одинаковый интерфейс и разное устройство.

§ 2
Модель задачи: классы и объекты

Ключевые слова:

- объектно-ориентированный анализ
- объект
- класс
- свойство
- метод

Объектно-ориентированный анализ

Для того чтобы построить объектно-ориентированную модель, нужно:

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

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

«Торпедная атака»

Выполним объектно-ориентированный анализ простой игры «Торпедная атака». На горизонте проходят вражеские корабли, которые нужно атаковать с помощью торпед (и потопить) — рис. 1.2.

Рис. 1.2

Игрок, управляющий торпедным аппаратом, может:

- поворачивать торпедный аппарат влево и вправо, изменяя точку прицеливания;
- давать команду на пуск торпеды (нажимая клавишу «пробел»).

Объекты и классы

Как построить объектную модель этой игры? Прежде всего, нужно разобраться, что такое объект.

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

Состояние объекта — это набор его свойств и текущие значения этих свойств. Например, автомобиль имеет свойства «масса», «скорость», «цвет»; их значения в какой-то момент могут быть равны 1510 кг, 60 км/ч, «синий».

Поведение определяется тем, как объект может воздействовать на другие объекты и как он реагирует на сигналы от других объектов. Автомобиль движется, если водитель нажимает на педаль газа; при столкновении с другими объектами он может повредить их или разрушиться сам.

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

В нашей задаче объекты — это корабли, торпеды и торпедный аппарат.

Кораблей может быть много, причем все они, с точки зрения нашей задачи, имеют общие свойства. Поэтому нет смысла описывать отдельно каждый корабль: достаточно один раз определить общие черты кораблей, а потом просто сказать, что все корабли ими обладают. Для этого в объектно-ориентированном программировании используют специальный термин — класс.

Класс — это описание множества объектов, имеющих общий набор свойств и общее поведение.

Класс выполняет роль чертежа или шаблона — по нему можно построить сколько угодно одинаковых объектов.

Классы в программе могут моделировать всё, что угодно: предметы, устройства, людей, организации, события, места на карте, помещения, математические функции, документы и т. д.

Классы объектов в игре

В нашей игре есть три типа объектов, т. е. три класса: корабли, торпеды и торпедный аппарат. Определим их свойства и возможные действия (поведение).

Свойства корабля, которые важны в этой игре, — это его координаты, скорость и его изображение. Корабль движется, т. е. движение — это один из вариантов поведения. Когда в корабль попадает торпеда, он взрывается и тонет — это тоже поведение.

Класс *Корабль* можно изобразить в виде схемы-прямоугольника, в верхней части которого записываются свойства, а в нижней — возможные действия (рис. 1.3, *а*).

Поведение объектов класса — это команды, которые они могут выполнять. При описании класса каждая такая команда оформляется в виде процедуры или функции и называется методом класса.

Метод класса — это процедура или функция, принадлежащая классу объектов.

Другими словами, метод класса определяет некоторое действие, которое могут выполнять все объекты этого класса.

а)

Корабль
координаты скорость изображение
двигаться взорваться

б)

Торпеда
координаты скорость курс изображение
двигаться

в)

Торпедный аппарат
координаты угол поворота
повернуться выстрелить

Рис. 1.3

Класс *Торпеда* — это описание общих свойств и методов торпед (рис. 1.3, *б*). Основные свойства торпеды — это координаты, скорость, курс и изображение, а метод один — двигаться в заданном направлении.

Класс *Торпедный аппарат* (рис. 1.3, *в*) содержит два свойства — координаты и угол поворота, и два метода — *повернуться* и *выстрелить*.

Взаимодействие объектов

Пока мы только описали модели отдельных объектов (точнее, классов). Теперь нужно разобраться, как эти объекты взаимодействуют между собой и с игроком.

Игрок может выполнять два действия:

- менять угол поворота торпедного аппарата (точку прицеливания);
- выполнять пуск торпеды (применять метод *выстрелить* торпедного аппарата).

По команде *выстрелить* из торпедного аппарата выходит торпеда, причём направление движения торпеды совпадает с текущим направлением торпедного аппарата.

Когда торпеда достигнет линии движения кораблей, нужно будет проверить, попала ли она в зону изображения какого-нибудь корабля. Если попала, подбитый корабль взрывается и тонет, если не попала, она просто исчезает с экрана.

Взаимодействие игрока с объектами в игре и объектов между собой показано на рис. 1.4.

Конечно, здесь показан простейший вариант связей. Если совершенствовать программу, нужно учесть, например, что корабль с помощью своих акустических средств способен заранее обнаружить торпеду и применить манёвр по уходу от неё. Вместе с тем, торпеда может иметь блок самонаведения и, обнаружив корабль, преследовать его.

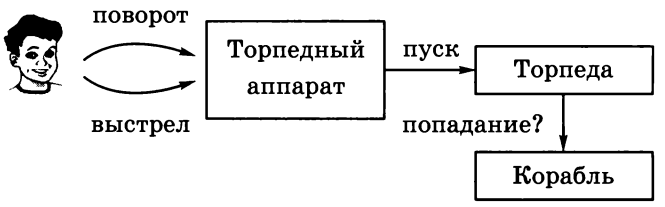


Рис. 1.4

Таким образом, мы построили объектную модель игры: выделили классы объектов, определили их свойства и методы. Обратите внимание, что мы пока ничего не говорили о том, *как* устроены объекты и *как* именно они будут действовать.

Теперь можно поручить разработку этих классов разным программистам. Согласно принципам ООП, ни один объект не должен зависеть от внутреннего устройства и алгоритмов работы других объектов. Поэтому каждый из программистов может решать свою задачу независимо от других. Важно только, чтобы все они точно соблюдали *интерфейсы* — правила, описывающие взаимодействие «своих» объектов с остальными.

Выводы

- Объектно-ориентированный анализ состоит в том, чтобы выделить взаимодействующие объекты и описать их свойства и поведение.
- Объектом можно назвать то, что имеет чёткие границы и обладает состоянием и поведением. Объект объединяет данные и методы их обработки.
- Любой объект относится к какому-то классу. Класс — это описание множества объектов, имеющих общий набор свойств и общее поведение.
- Метод класса — это процедура или функция, принадлежащая классу объектов.
- Ни один объект не должен зависеть от внутреннего устройства и алгоритмов работы других объектов.
- Интерфейсы — это правила, описывающие взаимодействие объектов в программе.

Вопросы и задания

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

6. Подумайте, какими свойствами и методами могли бы обладать объекты следующих классов: *Ученик, Учитель, Школа, Экзамен, Турнир, Урок, Страна, Президент, Браузер, Редактор, Яхта, Птица*. Придумайте свои классы объектов и выполните их анализ.
7. Постройте объектно-ориентированную модель для задачи моделирования дорожного движения. Рассмотрите классы объектов *Дорога, Светофор, Машина* и их взаимосвязь.
8. Придумайте свою задачу и выполните её объектно-ориентированный анализ. Примеры: моделирование работы магазина, банка, библиотеки, склада; гонки автомобилей или яхт, и т. п.

§3 Классы и объекты в программе

Ключевые слова:

- класс
- объект
- свойство
- метод
- атрибут класса
- экземпляр
- конструктор
- параметры
- переменные класса
- анимация

Объявление класса

Программа, использующая объектно-ориентированный подход, начинается с описания классов объектов.

Класс — это новый тип данных, который вводит программист. Как и структура (вспомните материал третьей части пособия), класс может объединять данные различных типов в единый блок. Кроме того, класс содержит ещё и описание *методов* работы со своими данными. Свойства описывают *состояние* объектов данного класса, а методы — их *поведение*. В программе, о которой мы говорили в предыдущем параграфе, есть класс *Корабль* (англ. *ship*). Его можно объявить так:

```
class TShip:
  pass
```

Эти строки вводят новый тип данных (класс) **TShip**[1)], т. е. сообщают компилятору, что в программе, возможно, будут использоваться объекты этого класса. При этом в памяти пока не создаётся ни одного объекта класса **TShip**. Это описание — как чертёж, по которому в нужный момент можно построить сколько угодно таких объектов.

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

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

```
ship = TShip()
```

Созданный объект относится к классу **TShip**, поэтому его называют *экземпляром* класса **TShip** или объектом класса **TShip**.

Поля класса

Класс **TShip**, который мы только что объявили, пока не содержит ни свойств, ни методов.

Свойства корабля — это его координаты, скорость и изображение. Так как корабль движется горизонтально, его *y*-координата не изменяется. Поэтому мы будем работать только с *x*-координатой.

Свойства хранятся в переменных, принадлежащих объекту. Такие переменные называются *полями*.

Поле — это переменная, принадлежащая объекту.

Поля и методы класса в языке Python называются *атрибутами* класса.

В языке Python в любой точке программы вы можете добавить поля объектам с помощью точечной записи, например:

```
ship = TShip()
ship.x = 30          # x-координата
ship.v = 3           # скорость
ship.image = image( ship.x, 150, "ship.gif" )
```

В последней строке в поле `image` записывается ссылка на объект-изображение, которое загружается из файла. Функция `image` объявлена в модуле `graph`, поэтому в самом верху программы нужно импортировать все функции из этого модуля[1]:

```
from graph import *
```

При «ручном» заполнении полей (так, как показано выше, отдельно для каждого объекта) у всех объектов класса **TShip** может оказаться разный набор полей. Как ни странно, транслятор Python «ничего не имеет против» этого и успешно выполнит такую программу. Но при этом во время выполнения программы могут произойти серьёзные ошибки. Чтобы этого не случилось, нужно соблюдать один из главных принципов объектно-ориентированного программирования: объект работает со своими данными только сам. Другие объекты (в том числе и другие процедуры и функции) могут обращаться к данным объекта только через его методы.

[1] Модуль `graph` может быть загружен с сайта автора: **http://kpolyakov.spb.ru/school/probook/python.htm**.

Конструктор

Вернёмся к записи

```
ship = TShip()
```

Внешне это выглядит как вызов функции без параметров — после имени **TShip** стоят пустые скобки. И действительно, здесь вызывается функция, которая создаёт объект. Она называется конструктором.

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

Мы не определили конструктор в описании класса, поэтому транслятор добавил его автоматически. Такой конструктор называется *конструктором по умолчанию*.

Если мы хотим, чтобы у всех объектов класса **TShip** был одинаковый набор полей, нам нужно заменить конструктор по умолчанию на свой конструктор.

Конструктор класса всегда имеет специальное имя `__init__` (от англ. *initialization — инициализация*, начальные установки). По два символа подчёркивания в начале и конце имени говорят о том, что конструктор — это функция, имеющая специальное значение в языке Python.

Добавим конструктор в класс **TShip**:

```
class TShip:
  def __init__ ( self ):      # конструктор
    self.x = 0
    self.v = 0
        self.image = None
```

Обратите внимание, что функция `__init__` записана со смещением вправо. Такой сдвиг делает её частью описания класса **TShip**. Получается, что это не просто функция, а *метод класса* **TShip**, внутри которого она расположена.

Первый (и здесь — единственный) параметр конструктора, как и любого метода класса, — это ссылка на сам объект, который создаётся. По традиции он всегда называется `self` (от англ. *self* — «сам»). С помощью этой ссылки мы обращаемся к полям объекта, например `self.x` означает «поле `x` текущего объекта». Если вместо этого написать просто `x`, транслятор будет считать, что речь идёт о локальной или глобальной переменной, а не о поле объекта.

В нашем конструкторе создаются три поля (*атрибута*): координата `x`, скорость `v` и изображение `image`. Первые два заполняются нулями, а в поле `image` записывается специальное значение `None` («ничего»); которое говорит о том, что изображения пока нет.

Свойствам корабля по-прежнему можно присваивать новые значения с помощью точечной записи, как мы делали это раньше. Что же изменилось? Теперь любой объект класса **TShip** обязательно имеет поля `x`, `v` и `image`, потому что они создаются в конструкторе, при создании объекта.

Полная программа, которая строит объект класса *Корабль* (и пока больше ничего не делает), выглядит так:

```
class TShip:
  def __init__ ( self ):      # конструктор
    self.x = 0
    self.v = 0
    self.image = None

ship = TShip()     # (1)
ship.x = 30
ship.v = 3
ship.image = image( ship.x, 150, "ship.gif" )
```

Не удивляйтесь, что в заголовке конструктора есть один параметр (`self`), а при создании объекта (в строке 1) мы вызвали конструктор без аргументов. В языке Python при вызове метода класса первый аргумент — ссылку на текущий экземпляр объекта — компилятор добавляет автоматически.

Конструктор с параметрами

Начальные значения полей можно задавать прямо при создании объекта. Для этого необходимо изменить конструктор, добавив дополнительные параметры — начальные значения свойств:

```
class TShip:
  def __init__( self, x0, v0, fileName ):
    self.x = x0 if x0 >= 0 else 0
    self.v = v0
    self.image = image( self.x, 150, fileName )
```

В конструкторе можно проверять правильность переданных параметров. Например, здесь гарантируется, что x-координата не окажется отрицательной. Теперь создавать объект будет проще:

```
ship = TShip( 30, 3, "ship.gif" )
```

Координата этого корабля $x = 30$, его скорость равна трём условным единицам, а изображение загружается из файла `ship.gif` в текущем каталоге.

Мы передали конструктору только три аргумента, а не четыре, как указано в заголовке метода `__init__`. Первый аргумент `self` — ссылку на только что созданный объект — компилятор добавит автоматически.

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

Данные класса

В конструкторе класса **TShip** есть одно «магическое» число — 150. Можно выполнить рефакторинг — сделать это число константой.

Будем считать, что все корабли имеют одну и ту же y-координату, поэтому эта константа должна принадлежать не конкретному экземпляру, а всем кораблям. Вместе с тем, делать её глобальной тоже не хочется: любая функция сможет её изменить, и из-за этого могут возникнуть ошибки во время работы программы.

Выход есть — нужно указать, что эта константа (назовём её `SHIP_Y`) принадлежит всему классу, а не отдельному объекту. В языке Python нет констант, и мы вынуждены ввести её как переменную. Обычно переменные, имена которых состоят только из прописных букв, считаются константами — все договариваются, что изменять их не будут.

Переменная, принадлежащая классу, называется *переменной класса*. Её нужно определить внутри описания класса на том же уровне, что и конструктор:

```
class TShip:
  SHIP_Y = 150
  def __init__( self, x, v, fileName ):
    self.x = x if x >= 0 else 0
    self.v = v
    self.image = image( self.x, TShip.SHIP_Y, fileName )
```

При обращении к переменной класса записывают название класса и через точку — имя переменной: `TShip.SHIP_Y`.

Методы

Теперь научимся добавлять в класс методы. Самый простой метод класса *Корабль* — метод *двигаться* (англ. *move*). Изображение корабля нужно просто переместить по оси x вправо на величину, равную скорости:

```
class TShip:
  SHIP_Y = 150
  def __init__( self, x0, v0, fileName ):
    ...
  def move( self ):
    moveObjectBy( self.image, self.v, 0 )
```

Многоточие обозначает тело конструктора, которое мы не стали повторять. Обратим внимание на некоторые особенности:

- метод `move` записан с отступом, который говорит о том, что он входит в класс **TShip**; этот отступ такой же, как и у конструктора;
- у метода `move` есть параметр `self` — это ссылка на текущий объект; для обращения к полям этого объекта нужно использовать точечную запись, например `self.image`.

В нашем методе `move` корабль перемещается по холсту с помощью функции `moveObjectBy` из модуля `graph`. Её первый аргумент — ссылка на перемещаемый объект, второй — смещение по оси x, третий — смещение по оси y (здесь он всегда равен 0).

Анимация

Напомним, как программируется анимация в Python. Программа запускает таймер, который срабатывает с заданным периодом, т. е. через равные интервалы времени. Срабатывание таймера — это событие, на которое программа должна отреагировать, — изменить изображение на экране. Для этого в программе нужно *установить обработчик события*, т. е. определить процедуру, которая будет вызываться при каждом срабатывании таймера.

Сначала выберем нужное количество кадров в секунду и определим соответствующий период обновления картинки:

```
fps = 20
updatePeriod = round( 1000 / fps )
```

Количество кадров в секунду записано в глобальную переменную `fps` (от англ. *frames per second* — кадры в секунду), а период обновления — в переменную `updatePeriod` (от англ. *update* — обновить). Чтобы вычислить период обновления в миллисекундах, мы делим величину интервала 1 с = 1000 мс на количество кадров в секунду и округляем результат с помощью функции `round`.

Будем считать, что изображение корабля хранится в файле `ship.gif`, который сохранён в том же каталоге, что и наша программа. Создадим объект-корабль:

```
ship = TShip( 30, 3, "ship.gif" )
```

и построим новую функцию, которая должна вызываться по таймеру:

```
def update():
  ship.move()
```

Остаётся установить таймер на нужное время, настроить его на вызов нужной функции и запустить анимацию:

```
onTimer( update, updatePeriod )
run()
```

Если вы сделали всё правильно, изображение корабля будет двигаться слева направо.

Строим флотилию

Попробуем создать флотилию кораблей. Сначала в основной программе определяем константу — количество кораблей:

```
NUMBER_OF_SHIPS = 3
```

Затем строим массив кораблей ships, которые начинают движение с разных точек с разными скоростями:

```
ships = []
for i in range(NUMBER_OF_SHIPS):
  ships.append( TShip(100*i, 3+i, "ship.gif") )
```

Остаётся изменить функцию update так, чтобы она вызывала метод move для всех кораблей из массива ships:

```
def update():
  for ship in ships:
    ship.move()
```

Нам не потребовалось работать с массивами, где хранятся отдельные свойства объектов. Каждый объект класса **TShip** при создании (так сказать, «от рождения») «знает», как он устроен и что ему нужно делать по команде move, поэтому никаких сложностей не возникает.

Можно ли было написать такую же программу, не используя объекты? Конечно, да. И она, возможно, получилась бы короче, чем наш объектный вариант (с учётом описания классов). В чём же преимущества ООП? Дело в том, что ООП — это средство разработки больших программ, моделирующих работу сложных систем. В этом случае очень важно, что:

- задачи реального мира проще всего описываются именно с помощью понятий «объект», «свойства», «действия» (методы);
- основная программа, описывающая решение задачи в целом, получается простой и понятной; все команды (вызовы методов) напоминают действия в реальном мире;
- легко разделить работу между программистами в команде: каждый программирует свой класс и может работать независимо от других;
- если объект *Корабль* понадобится в других проектах, можно будет легко использовать уже готовый класс TShip со всеми его свойствами и методами.

Выводы

- В отличие от структуры класс содержит не только данные, но и методы работы с этими данными.
- Поле — это переменная, принадлежащая объекту. Метод — это процедура или функция, принадлежащая объекту.

- Поля и методы класса в языке Python называются атрибутами.
- Конструктор — это метод класса, который вызывается для создания объекта этого класса. Если конструктор не задан при описании класса, транслятор автоматически добавляет конструктор по умолчанию.
- Конструктор в языке Python всегда называется `__init__`.
- Первый параметр любого метода класса, в том числе и конструктора, — это ссылка на текущий объект. По традиции он называется `self`. При вызове метода значение этого параметра не указывается.
- Данные, которые объявлены внутри описания класса, но вне его методов, — это данные класса. При обращении к ним нужно указывать имя класса.

Вопросы и задания

1. Чем различаются понятия «структура» и «класс»?
2. Как можно присвоить полям объекта начальные значения? Сравните разные способы.
3. Почему лучше создавать все поля объекта в конструкторе класса?
4. Может ли быть у одного класса несколько конструкторов? Обоснуйте ответ и проверьте его экспериментально.
5. Проверьте, можно ли обращаться к приведённой в параграфе переменной класса `SHIP_Y` не через класс, а через объект: `ship.SHIP_Y`.
6. *Проект.* Закончите программу, рассмотренную в тексте параграфа. Добавьте в окно программы фоновый рисунок, сделайте так, чтобы корабли двигались справа налево. Что, по-вашему, нужно делать, когда корабль выходит за левый край холста?
7. *Проект.* Добавьте в программу новый класс *Торпедный аппарат*. Изображение прицела должно перемещаться при нажатии на клавиши-стрелки (обработчик события можно установить с помощью функции `onKey` из модуля `graph`).

*8. *Проект.* Добавьте в программу класс *Торпеда*. Торпеда должна выходить из торпедного аппарата при нажатии на пробел. Когда торпеда дойдёт до линии, по которой двигаются корабли, нужно проверить, не попала ли она в какой-нибудь корабль. Если попала, корабль должен взорваться, если не попала, торпеда пропадает с экрана.

   Указания:
   - используйте массивы (списки) для хранения объектов-кораблей и объектов-торпед;
   - в процедуре `update` для каждой пары «корабль — торпеда» проверяйте, не попала ли торпеда в цель;

- если торпеда попала в цель, корабль взрывается (картинка меняется и через некоторое время исчезает), торпеда удаляется с экрана;
- торпеда, прошедшая линию движения кораблей и не попавшая в цель, удаляется с экрана.

Интересный сайт

younglinux.info/oopython.php — Python. Введение в объектно-ориентированное программирование

§4 Скрытие внутреннего устройства

Ключевые слова:

- интерфейс
- инкапсуляция
- свойство

Зачем это делать?

Объекты в программе для обмена данными друг с другом используют *интерфейс* — открытые свойства и методы. При этом все внутренние данные и детали внутреннего устройства объекта должны быть скрыты от «внешнего мира» (рис. 1.5).

Рис. 1.5

Такой подход позволяет:

- обезопасить внутренние данные (поля) объекта от изменений (возможно, разрушительных) со стороны других объектов;
- проверять правильность данных, поступающих от других объектов, тем самым повышая надёжность программы;
- переделывать внутреннюю структуру и код объекта любым способом, не меняя интерфейса; при этом никакой переделки других объектов не потребуется.

Всё это становится действительно важно, когда разрабатывается большая программа и необходимо обеспечить её надёжность.

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

Скрытие полей

Разберём простой пример. Во многих системах программирования есть класс, описывающий свойства «пера», которое используется при рисовании линий в графическом режиме. Назовём этот класс **TPen**, в простейшем варианте он будет содержать только одно поле `color`, которое определяет цвет.

Будем хранить информацию о цвете в виде символьной строки, в которой записан шестнадцатеричный код составляющих цвета в модели RGB. Например, `"00FF7F"` — это светло-зелёный цвет, его красная (R) составляющая равна 0, зелёная (G) — FF_{16} = 255 и синяя (B) — $7F_{16}$ = 127.

Класс *Перо* можно объявить так:

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

По умолчанию в Python все атрибуты класса (поля и методы) открытые, общедоступные (англ. *public*). Это значит, что к полю `color` может обратиться (и изменить его!) любая функция и объект любого класса. Например, некоторая функция может случайно записать в это поле ошибочное значение `"Abra cadabra"`, которое потом приведёт к ошибке в самый неожиданный момент.

Для повышения надёжности программы лучше сделать все поля объектов закрытыми, чтобы их не смогли изменить другие объекты программы. Имена тех атрибутов, которые нужно скрыть, в языке Python должны начинаться с двух знаков подчёркивания, например так:

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

В этом примере поле `__color` — закрытое, или частное (как частная собственность, англ. *private*). Теперь никакой другой объект или подпрограмма не сможет его случайно изменить. К закрытым полям нельзя обратиться извне (это могут делать только методы самого объекта), поэтому теперь (обычными средствами) невозможно не только изменить поле объекта, но и просто узнать его значение.

Итак, все поля и методы, названия которых начинаются с двух знаков подчёркивания, — закрытые. Вы наверняка заметили, что конструктор — метод `__init__` — тоже начинается с двух знаков подчёркивания. Это особый метод: он никогда не вызывается по имени, транслятор выполняет его автоматически при создании каждого нового объекта.

Иногда в программах на языке Python вы можете увидеть поля и методы классов, имена которых начинаются не с двух, а с одного знака подчёркивания. Хотя они доступны другим объектам, знак подчёркивания говорит о том, что, по мнению программиста, обращаться к ним напрямую не следует.

Доступ к полям через методы

Чтобы всё-таки можно было работать со скрытым полем (читать и/или изменять его значение), нужно добавить к классу ещё два метода. Один из них будет возвращать текущее значение закрытого поля `__color`, а второй — присваивать полю новое значение.

Метод для чтения значения поля иногда называют «геттером» (от англ. *get* — получать), а метод записи нового значения — «сеттером» (от англ. *set* — устанавливать). Мы дадим им имена `getColor` и `setColor`:

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

Метод `getColor` — это функция, которая просто возвращает значение внутреннего поля `__color`. А вот в методе `setColor` мы выполняем проверку данных, не разрешая присваивать полю недопустимые значения[1]. Если длина переданной символьной строки `newColor` не равна 6, в поле `__color` записывается значение по умолчанию — код чёрного цвета 000000.

Что же улучшилось в сравнении с первым вариантом (когда поле было открытым)? Согласно принципам ООП, обращаться к данным объекта можно *только* с помощью методов. В этом случае внутреннее устройство объекта можно в любой момент изменить, и это не повлияет на другие объекты.

Теперь для записи и чтения цвета любого объекта класса **TPen** необходимо использовать методы:

```
pen = TPen()
pen.setColor( "FFFF00" )
print( "Цвет пера:", pen.getColor() )
```

[1] Конечно, такая простая проверка сделана только для того, чтобы не усложнять пример. В реальной программе нужно ещё проверить, что все 6 символов — шестнадцатеричные цифры.

Таким образом, мы скрыли (защитили) внутренние данные, но одновременно обращение к свойствам стало выглядеть несколько неуклюже: вместо `pen.color="FFFF00"` теперь нужно писать `pen.setColor("FFFF00")`.

Свойства (property)

Чтобы упростить запись при обращении к полям через методы, во многие языки программирования ввели понятие свойства (англ. *property*). Внешне обращение к свойству выглядит так же, как обращение к полю объекта. Но на самом деле при записи и чтении значения свойства вызываются методы объекта, определённые программистом.

Свойство — это способ доступа к внутреннему состоянию объекта, имитирующий обращение к его полю.

Свойство `color` в нашем случае можно определить так:

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

Здесь мы добавили два подчёркивания в начало названий методов `getColor` и `setColor`. Следовательно, они стали частными (англ. *private*), скрытыми от других объектов.

Строка, выделенная фоном, определяет общедоступное свойство с названием `color`. При чтении этого свойства вызывается метод `__getColor`, а при записи нового значения — метод `__setColor`.

В программе можно использовать свойство `color` так, как будто это общедоступное поле:

```
pen.color = "FFFF00"
print( "Цвет пера:", pen.color )
```

Поскольку функция `__getColor` просто возвращает значение поля `__color` и не выполняет никаких дополнительных действий, можно было вообще удалить метод `__getColor` и вместо него использовать лямбда-функцию:

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

Лямбда-функция (она выделена фоном) принимает единственный параметр — объект, который называется x, и возвращает его поле `__color`.

Таким образом, объект взаимодействует с другими объектами только с помощью своих общедоступных свойств и методов (интерфейса). Это не позволяет другим объектам напрямую менять его данные, уменьшая вероятность ошибок. При изменении полей через методы объект может проверять правильность входных данных.

Меняем внутреннее устройство

При сохранении интерфейса можно как угодно менять внутреннюю структуру данных и код методов, и это никак не будет влиять на другие объекты.

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

Представление цвета в виде символьной строки неэкономно и неудобно, поэтому на практике с кодом цвета чаще всего работают как с целым числом. Для хранения этого числа введём скрытое поле `__color`:

```
self.__color = 0
```

При этом мы хотим сохранить интерфейс, т. е. сделать так, чтобы другие объекты даже не заподозрили о внутренних изменениях в классе **TPen**.

Допустим, мы раньше договорились о том, что свойство `color` класса **TPen** — это символьная строка длиной 6 символов. Поскольку теперь цвет хранится как целое число, необходимо поменять методы `__getColor` и `__setColor`, которые работают с этим полем. Метод `__getColor` должен вернуть код цвета, преобразованный в строку из 6 символов, а метод `__setColor` должен преобразовать полученную символьную строку в числовой код и записать этот код в поле `__color`.

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

Для перевода числового кода в символьную запись используется функция `format`. Формат `06X` означает «вывести значение в шестнадцатеричной системе (`X`) в 6 позициях, свободные позиции слева заполнить нулями». Для обратного преобразования применяем функцию `int`, которой передаётся второй аргумент `16` — основание системы счисления.

В этом примере мы принципиально изменили внутреннее устройство объекта — вместо строкового поля для хранения цвета теперь используется целочисленное. Однако другие объекты даже не «догадаются» о такой замене, потому что сохранился интерфейс — свойство `color` по-прежнему имеет строковый тип. Таким образом, инкапсуляция позволяет как угодно изменять внутреннее устройство объектов, не затрагивая интерфейс. При этом все остальные объекты изменять не требуется.

Скрытие данных чаще всего увеличивает длину программы, однако мы получаем и важные преимущества. Код, связанный с объектом, разделён на две части: общедоступную часть (интерфейс) и закрытую. Их можно сравнить с надводной и подводной частями айсберга (рис. 1.6).

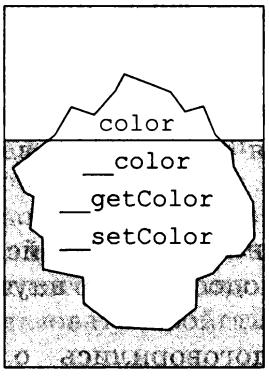


Рис. 1.6

Свойство «только для чтения»

Иногда нужно запретить другим объектам менять какое-то свойство объекта. Другими словами, требуется сделать свойство «только для чтения» (англ. *read-only*).

Пусть, например, мы строим программную модель автомобиля. Как правило, другие объекты не могут непосредственно менять его скорость, однако могут получить информацию о ней — «прочитать» значение скорости.

При описании такого свойства метод записи не указывают вообще (или указывается пустое значение None):

```
class TCar:
  def __init__( self, _v ):
    self.__v = _v
  velocity = property ( lambda x: x.__v )
```

Теперь изменить поле __v может только сам объект.

Возможен другой вариант записи:

```
class TCar:
  def __init__( self, _v ):
    self.__v = _v
  @property
  def velocity(self):
    return self.__v
```

Команда **@property** перед методом velocity говорит о том, что создаётся свойство только для чтения с таким именем, и метод velocity служит для чтения его значения[1].

Свойство не всегда может соответствовать значению какой-то переменной. Например, в класс **TRect**, который описывает прямоугольник (от англ. *rectangle* — прямоугольник), можно добавить свойство area (по-английски — площадь):

```
class TRect():
  def __init__( self, a, b ):
    self.__a = a
    self.__b = b
  area = property( lambda x: x.__a*x.__b )
```

Понятно, что хранить площадь прямоугольника в отдельном поле нет смысла — её всегда можно вычислить, перемножив длины сторон. Кроме того, записывать значение в это поле тоже бессмысленно. Поэтому в строке, выделенной фоном, вводится новое свойство area «только для чтения». При обращении к этому свойству результат вычисляется с помощью лямбда-функции, перемножающей длины сторон — значения двух закрытых полей. При запуске программы

```
rct = TRect( 2, 3 )
print( rct.area )
```

на экран выводится число 6.

[1] Такие инструкции, как **@property**, в языке Python называются *декораторами*. Их обсуждение выходит за рамки пособия.

Выводы

- Обмен данными между объектами выполняется с помощью общедоступных свойств и методов, которые составляют интерфейс класса.
- Скрытие внутреннего устройства объектов называется инкапсуляцией. Инкапсуляцией также называют объединение в одном объекте данных и методов работы с ними.
- Изменение внутреннего устройства объектов (реализации) не влияет на взаимодействие с другими объектами, если не меняется интерфейс.
- Имена закрытых (частных) полей и методов в Python должны начинаться с двух знаков подчёркивания.
- Читать и изменять значения закрытых полей класса можно только с помощью методов этого класса.
- Свойство — это способ доступа к внутреннему состоянию объекта, имитирующий обращение к его полю. При создании свойства нужно указать метод чтения и метод записи.
- Свойство, предназначенное только для чтения, не имеет метода записи. Другие объекты не могут изменить значение этого свойства.

Вопросы и задания

1. Что такое интерфейс объекта?
2. Что такое инкапсуляция? Оцените её достоинства и недостатки.
3. Как различить общедоступные и закрытые данные и методы в описании классов?
4. Почему рекомендуют делать доступ к полям объекта только с помощью методов?
5. При каких ошибочных значениях параметра не сработает наша защита от неправильных данных в методе `setColor` (из параграфа)? Исправьте метод так, чтобы исключить эту ошибку.
6. Что такое свойство? Зачем во многие языки программирования введено это понятие?
7. Почему методы чтения и записи, которые использует свойство, обычно делают закрытыми?
8. Зачем нужны свойства «только для чтения»? Приведите свой пример.
9. Сравните два варианта вычисления площади прямоугольника в классе **`TRect`** (из параграфа): с помощью метода-функции и с помощью свойства. Какой из них вам больше нравится?
10. Подумайте, в каких ситуациях может быть нужно свойство «только для записи» (которое нельзя прочитать). Предположите, как ввести такое свойство в описание класса. Проверьте вашу догадку экспериментально.
11. *Проект*. Измените программу игры «Торпедная атака» (из § 2) так, чтобы все поля у объектов были закрытыми. Используйте свойства для доступа к данным.

§5
Иерархия классов

Ключевые слова:

- классификация
- базовый класс
- производный класс
- класс-наследник
- иерархия
- абстрактный класс
- абстрактный метод
- полиморфизм
- исключение

Наследование

Как в науке, так и в быту важную роль играет *классификация* — разделение объектов на группы (классы), объединённые общими признаками. Значительно проще не полностью описывать каждый объект «с нуля», а сравнивать его с уже известными объектами.

Например, есть много видов фруктов[1] (яблоки, груши, бананы, апельсины и т. д.), но все они обладают некоторыми общими свойствами. Если перевести этот пример на язык ООП, у нас есть *класс Фрукт*. У этого класса есть *производные классы* (*классы-наследники, классы-потомки, подклассы*) — *Яблоко*, *Груша*, *Банан*, *Апельсин* и др. В свою очередь, для этих классов класс *Фрукт* — это *базовый класс, суперкласс, класс-предок* (рис. 1.7).

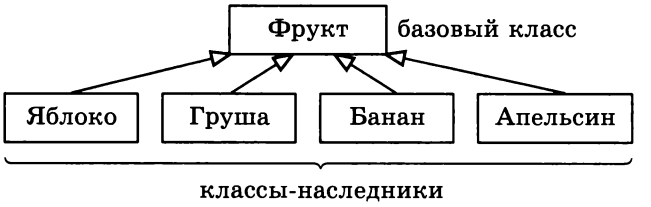


Рис. 1.7

Стрелка с белым наконечником на рис. 1.7 указывает на базовый класс. Например, класс *Яблоко* — это наследник класса *Фрукт*, а класс *Фрукт* — базовый класс для класса *Яблоко*. Это означает, что все объекты класса *Яблоко* обладают всеми свойствами и методами класса *Фрукт*.

Классический пример научной классификации — классификация животных или растений. Она представляет собой *иерархию* — многоуровневую структуру. Например, лев — это вид рода пантер семейства кошачьих отряда хищных и т. д. Говоря на языке ООП, класс *Лев* — это наследник класса *Пантеры*, а тот, в свою очередь, наследник класса *Кошачьи*, который является наследником класса *Хищные* и т. д. Часть этой классификации показана на рис. 1.8.

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

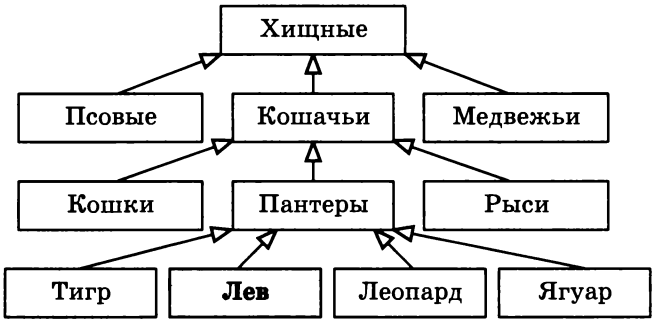


Рис. 1.8

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

Например, можно сказать, что яблоко — это фрукт, а лев — животное класса *Кошачьи*.

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 объектов в игре

Игра, которую мы будем программировать, моделирует поведение и взаимодействие разных объектов в космосе. Перечислим некоторые классы объектов:

- *Корабли* — космические корабли, которые двигаются по прямой линии и «отталкиваются» от стенок игрового поля;
- *Странники* — движутся так же, как и корабли, но в случайные моменты времени меняют направление движения;
- *Чёрные дыры* — неподвижные области, которые поглощают корабли и странников;
- *Пульсары* — то же, что и *Чёрные дыры*, но их размер меняется в случайные моменты времени;

Конечно, вы можете добавить какие-то другие классы в свою игру.

Перед тем как начать программировать, нужно выполнить объектно-ориентированный анализ: выделить классы объектов, определить их свойства и методы.

Каждый объект в нашей игре имеет координаты и размеры на экране (высоту и ширину). Эти общие свойства можно описать в базовом классе *Игровой объект*.

Среди объектов можно выделить неподвижные (*Чёрные дыры* и *Пульсары*) и подвижные (*Корабли* и *Странники*). У подвижных объек-

тов есть дополнительные свойства: направление и скорость движения. Эти свойства добавим в новый класс *Движущийся объект* — наследник класса *Игровой объект*.

Странник — это особый *Корабль*, который меняет направление движения. Поэтому класс *Странник* должен быть наследником класса *Корабль*, а класс *Корабль* — наследником класса *Движущийся объект*.

Объекты классов *Чёрная дыра* и *Пульсар* не двигаются, это наследники класса *Игровой объект*. Кроме того, *Пульсар* — это разновидность *Чёрной дыры* (наследник этого класса). В итоге у нас получается диаграмма классов, показанная на рис. 1.9.

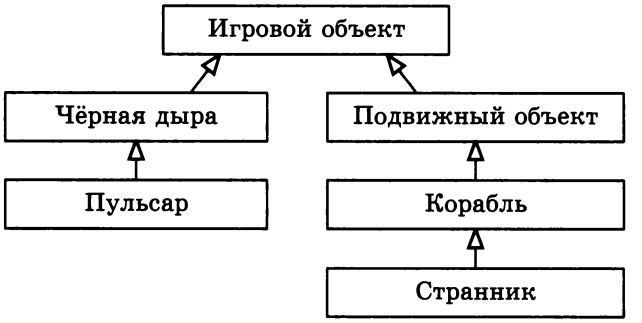


Рис. 1.9

Итак, для того чтобы не описывать несколько раз одинаковые свойства и методы, 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 определённого класса, а классы образуют иерархию наследования.

Базовый класс

Начнём с описания класса *Игровой объект* (**TGameObject**). Он должен объединить общие свойства и методы всех объектов в игре. Свойства — это координаты базовой точки (x и y), ширина (англ. *width*) и высота (англ. *height*). Можно начать с такого варианта:

```
class TGameObject:
  def __init__( self, x, y, width, height ):
    self.__x = x
    self.__y = y
    self.__width = width
    self.__height = height
```

```
@property
def x( self ): return self.__x
@property
def y( self ): return self.__y
@property
def width( self ): return self.__width
@property
def height( self ): return self.__height
```

Все четыре поля (__x, __y, __width и __height) — закрытые, они заполняются при создании объекта. Для чтения их значений мы добавили четыре свойства «только для чтения», а изменять значения полей могут только сами объекты класса **TGameObject**.

Заметим, что *не существует* игрового объекта «вообще», так же как не существует «просто фрукта», не относящегося к какому-то виду. Поэтому понятие «игровой объект» — это просто набор каких-то свойств и методов, т. е. *абстракция*. А класс, описывающий такую абстракцию, называется *абстрактным*. Создавать объекты такого класса бессмысленно.

Абстрактный класс — это класс, который не предназначен для создания объектов (экземпляров). Абстрактный класс служит только для создания классов-наследников.

Классы-наследники могут иметь дополнительные поля и свойства, которых нет у базового класса **TGameObject**, но все поля и свойства базового класса у них обязательно будут.

Теперь перейдём к общим методам. Все объекты участвуют в анимации: меняют своё положение и/или свойства через небольшой интервал времени. Добавим в базовый класс **TGameObject** метод update (в переводе с английского — обновить), который будет вызываться после окончания очередного интервала анимации (рис. 1.10). При вызове этого метода объект должен «прожить» этот интервал, перейти в новое состояние.

Рис. 1.10

Хотя метод update есть у всех объектов, наследники класса **TGameObject** выполняют его по-разному: *Корабли* двигаются, *Пульсары* изменяют свой размер, *Чёрные дыры* никак не меняются... Поэтому

в базовом классе невозможно написать метод update, работающий для всех классов.

Конечно, можно добавить пустой метод-«заглушку»:

```
class TGameObject:
  ...
  def update( self ):
    pass
```

Здесь **pass** — это пустой оператор, команда «ничего не делать». Он обычно говорит о том, что в этом месте программы должны быть какие-то команды, но пока их нет.

Использование «заглушки» — это не совсем правильно, поскольку кто-то может создать объект класса **TGameObject** и пытаться его использовать, а это бессмысленно. Поэтому мы вообще не будем определять метод update. В этом случае метод update называют абстрактным методом.

Абстрактный метод — это метод класса, реализация которого в классе не приводится.

Пока в нашем классе **TGameObject** вообще нет методов (метод update мы убрали). Как же сказать программе, что метод update должен быть, но пока не определён? В языке Python можно запретить создание объектов класса **TGameObject** и его наследников, у которых нет метода update. Этим мы покажем, что класс **TGameObject** — абстрактный[1].

В конструкторе класса проверим, есть ли у объекта метод update, и если такого метода нет, искусственно создадим аварийную ситуацию — *исключение*, — что приведёт к завершению программы:

```
class TGameObject:
  def __init__( self, x, y, width, height ):
    self.__x = x
    self.__y = y
    self.__width = width
    self.__height = height

    if not hasattr( self, "update" ):
      raise NotImplementedError(
              "Нельзя создать такой объект!" )

  ...
```

Функция hasattr возвращает логическое значение True, если у переданного ей объекта (здесь — self) есть указанный атрибут (в нашем случае — поле или метод с именем update).

---

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

Служебное слово `raise` означает «создать исключение», тип этого исключения — `NotImplementedError` (ошибка «не реализовано»), в скобках записана символьная строка, которую увидит пользователь в сообщении об ошибке.

В конструкторе класса **TGameObject** мы требовали, чтобы все создаваемые объекты имели метод `update`. Но такого метода в описании базового класса нет! Поэтому при попытке создать объект класса **TGameObject** с помощью конструктора:

```
obj = TGameObject( 100, 20, 10, 10 ) # ошибка!
```

мы получим сообщение об ошибке. Этот класс — абстрактный.

Откуда же возьмётся метод `update`? Дело в том, что мы построили класс **TGameObject** только для того, чтобы создавать на его основе классы-наследники. В этом классе мы объединили общие свойства всех игровых объектов, чтобы не повторять их описание во всех остальных классах.

Класс-наследник не будет абстрактным, если определит все абстрактные методы базового класса, в нашем случае — метод `update`. Тогда мы сможем создать объект такого класса.

Каждый класс-наследник может реализовать метод `update` по-своему. Такая возможность называется *полиморфизмом*.

Полиморфизм (от греч. πολυ — много, и μορφη — форма) — это возможность классов-наследников по-разному реализовать метод базового класса.

Доступ к полям

В приведённом варианте класса **TGameObject** все поля напрямую недоступны, возможно только чтение их значений через свойства. Однако в дальнейшем такая ситуация будет нам мешать. Например, движущимся объектам нужно изменять свои координаты. Если поля закрыты и нет методов для работы с ними, сделать это невозможно.

Для правильной работы классов-наследников можно обеспечить им доступ к полям базового класса. Но желательно как-то отметить, что никто другой, кроме наследников, не должен изменять значения этих полей. Для этого в начале имени поля ставят *одно* подчёркивание. По соглашению, принятому в сообществе Python-программистов, это говорит о том, что переменную не следует изменять, несмотря на то что она общедоступна. То же самое относится и к методам, названия которых начинаются с одного подчёркивания.

Вот окончательный вариант базового класса:

```
class TGameObject:
  def __init__( self, x, y, width, height ):
    self._x = x
    self._y = y
    self._width = width
    self._height = height
    if not hasattr( self, "update" ):
      raise NotImplementedError(
            "Нельзя создать такой объект!" )
  @property
  def x( self ): return self._x
  @property
  def y( self ): return self._y
  @property
  def width( self ): return self._width
  @property
  def height( self ): return self._height
```

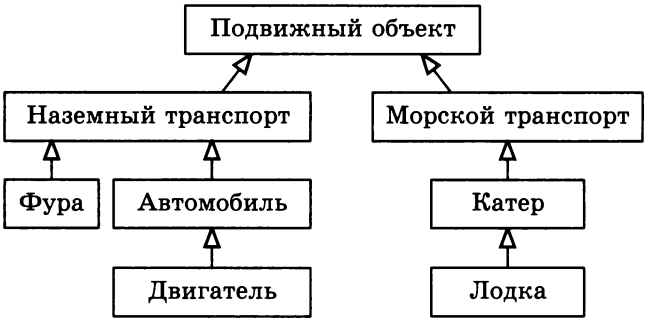
Во многих объектно-ориентированных языках программирования (C++, C#, Java, Delphi) кроме общедоступных (англ. *public*) и закрытых (англ. *private*) элементов класса есть ещё *защищённые* (англ. *protected*). Доступ к ним имеет сам класс, в котором объявлены эти данные и методы, и наследники этого класса. К сожалению, в языке Python защищённые поля и методы сделать невозможно.

Выводы

- Объектно-ориентированное программирование — это такой подход к программированию, при котором программа представляет собой множество взаимодействующих объектов, каждый из которых является экземпляром определённого класса, а классы образуют иерархию наследования.
- Классы-наследники обладают всеми свойствами и методами базового класса. Они могут изменять методы базового класса или добавлять к ним собственные свойства и методы.
- Абстрактный метод — это метод класса, реализация которого в классе не приводится.
- Абстрактный класс — это класс, который не предназначен для создания объектов (экземпляров). Любой класс, в котором есть абстрактный метод, является абстрактным.
- Полиморфизм — это возможность классов-наследников по-разному реализовать метод базового класса.
- Исключение — это аварийная ситуация, которая возникает в случае ошибки при выполнении программы. Программа может сама создать (сгенерировать, «выбросить») исключение.
- Поля и методы, названия которых начинаются с одного знака подчёркивания, предназначены для изменения только в данном классе и его классах-наследниках.

Вопросы и задания

1. Приведите свои примеры отношений «наследование» и «часть — целое».
2. Как вы думаете, стоит ли в рассмотренной в параграфе игре вводить ещё класс *Неподвижный объект* — наследник класса *Игровой объект*?
3. Используя дополнительные источники, составьте свою классификацию каких-то объектов (например, фотокамер, веб-сайтов, печатных изданий и т. п.). Используйте 2–3 уровня наследования.
4. Постройте классификацию, в которую войдут: молоко, масло, говядина, свинина, огурцы, помидоры, овсяная каша, хлеб.
5. Найдите ошибки в проектировании системы классов.



6. В программе нужно моделировать следующие классы: *Корабль, Подводная лодка, Самолёт, Вертолёт, Гидросамолёт, Мотоцикл, Трактор*. Постройте иерархию классов для этой задачи.
7. Что такое абстракция? Приведите примеры.
8. Зачем нужен абстрактный класс, если создать объекты такого класса нельзя?
9. Как построить абстрактный класс в языке Python?
10. В каком случае класс-наследник абстрактного класса сам не будет абстрактным классом?
11. Обсудите достоинства и недостатки методов-«заглушек», состоящих из единственной команды **pass**.
12. Расскажите о разных способах, с помощью которых класс-наследник может обратиться к полям базового класса. Сравните их, оцените достоинства и недостатки каждого способа.
13. Можно ли в функции, не относящейся к классу, изменить значения полей объекта, имена которых начинаются с одного символа подчёркивания?
14. Добавьте в иерархию объектов игры из параграфа (см. рис. 1.9) свои классы объектов.

§6
Классы-наследники (I)

Ключевые слова:

- базовый класс
- наследник
- конструктор
- переопределение метода
- полиморфизм

Классы-наследники

Теперь будем строить классы-наследники. Начнём с простого класса *Чёрная дыра* (**TBlackHole**). Как следует из диаграммы классов на рис. 1.9, класс **TBlackHole** должен быть прямым наследником базового класса **TGameObject**. При объявлении класса-наследника после его имени в скобках записывается название базового класса:

```
class TBlackHole( TGameObject ):
  ...
```

Будем изображать на холсте чёрные дыры чёрными кругами. Чтобы определить такой круг, нужно задать координаты его центра (назовём их xCenter и yCenter) и радиус (radius). Уже можно написать заголовок конструктора:

```
def __init__( self, xCenter, yCenter, radius ):
  ...
```

При создании объекта класса-наследника в первую очередь вызывается конструктор базового класса (**TGameObject**). Ему нужно передать координаты базовой точки и размеры объекта. За базовую точку чёрной дыры удобно принять её центр, а высота и ширина равны диаметру, т. е. двум радиусам:

```
def __init__( self, xCenter, yCenter, radius ):
  TGameObject.__init__( self, xCenter, yCenter,
                        2*radius, 2*radius )
  ...
```

После того как отработает конструктор базового класса, нужно выполнить дополнительные действия: создать объект-круг чёрного цвета, запомнить ссылку на него во внутреннем поле и вывести объект на экран:

```
brushColor( "black" )
self._image = circle( xCenter, yCenter, radius )
```

Приведём описание класса полностью:

```
class TBlackHole( TGameObject ):
  def __init__( self, xCenter, yCenter, radius ):
    TGameObject.__init__( self, xCenter, yCenter,
                          2*radius, 2*radius )
    brushColor("black")
    self._image = circle( xCenter, yCenter,radius )
  def update( self ):
    pass
```

Обратите внимание, что мы добавили в конец описания класса метод update, **который ничего не делает. Действительно, во время анимации чёрная дыра никак не изменяется. Если не добавить этот метод, класс окажется абстрактным, так как в нём не будет реализации обязательного метода. Поэтому создать объекты этого класса будет невозможно.**

Теперь мы готовы написать всю программу:

```
from random import randint
from graph import *

SCREEN_WIDTH = 600    # ширина окна
SCREEN_HEIGHT = 400   # высота окна
fps = 20              # частота кадров
updatePeriod = round(1000 / fps)
# Здесь должны быть описания классов
# TGameObject и TBlackHole

windowSize(SCREEN_WIDTH, SCREEN_HEIGHT)
canvasSize(SCREEN_WIDTH, SCREEN_HEIGHT)

NUMBER_OF_BLACKHOLES = 10
blackHoles = []
for i in range(NUMBER_OF_BLACKHOLES):
  blackHoles.append( TBlackHole(
          randint(0, SCREEN_WIDTH),
          randint(0, SCREEN_HEIGHT),
          randint(10, 20) ) )

def update():
  for bh in blackHoles:
    bh.update();

onTimer( update, updatePeriod )

run()
```

В программе создаётся массив blackHoles **из объектов класса TBlackHole, их количество определяется константой** NUMBER_OF_BLACKHOLES.

Координаты и радиусы этих объектов выбираются случайно с помощью функции randint из модуля random.

Анимация выполняется с помощью процедуры update, которая вызывается по таймеру с периодом updatePeriod. В этой процедуре в цикле перебираются все чёрные дыры из массива blackHoles, для каждой из них вызывается метод update. В данном случае метод update ничего не делает, так как чёрная дыра неподвижна и не меняет свой размер.

Пульсар

Пульсар — это объект, который напоминает чёрную дыру, но его размеры постоянно изменяются. Следовательно, можно сказать, что пульсар — это разновидность объекта *Чёрная дыра*, или, говоря на языке программистов, подкласс (производный класс, класс-наследник) класса **TBlackHole**.

В отличие от чёрных дыр пульсары будем закрашивать коричневым цветом. Получается такой класс:

```
class TPulsar( TBlackHole ):
  def __init__( self, xCenter, yCenter, radius ):
    TBlackHole.__init__( self, xCenter, yCenter, radius )
    changeFillColor( self._image, "brown" )
```

В заголовке указываем, что TPulsar — это наследник класса **TBlackHole**, т. е. он обладает всеми свойствами базового класса **TBlackHole**. Заметим, что, кроме того, он является и наследником класса **TGameObject**.

Конструктор нового класса принимает те же параметры, что и конструктор базового класса — координаты центра и начальный радиус круга. После вызова конструктора базового класса мы меняем цвет заливки на коричневый с помощью функции changeFillColor из модуля graph.

Пока новый класс не отличается от базового ничем, кроме цвета заливки. А должен отличаться тем, что радиус пульсара меняется случайным образом на каждом шаге анимации. Для этого нужно добавить в класс **TPulsar** собственный метод update.

У класса **TBlackHole**, который служит базовым классом для **TPulsar**, тоже есть метод update, поэтому мы выполняем *переопределение* метода (англ. *override*) — пишем новую версию метода с таким же именем для класса-наследника.

Переопределение метода — это создание в классе-наследнике новой версии метода, определённого ранее для базового класса.

Новый метод update будет вызывать скрытый метод __changeRadius, который устанавливает новый радиус:

```
class  TPulsar(  TBlackHole  ):
  ...
  def update( self ):
    self.__changeRadius( randint(5, 20) )
  def __changeRadius( self, newRadius ):
    self._width = 2*newRadius
    self._height = 2*newRadius
    changeCoords( self._image,
      [(self._x-newRadius, self._y-newRadius),
       (self._x+newRadius, self._y+newRadius)] )
```

В начале функции __changeRadius мы изменяем свойства объекта, которые хранятся в переменных базового класса **TGameObject**: ширину и высоту.

Затем функция changeCoords из модуля graph изменяет координаты рисунка на холсте. При обращении к ней первый аргумент — self._image — это ссылка на рисунок, второй — массив из двух кортежей: $[(x_1, y_1), (x_2, y_2)]$. Эти кортежи содержат координаты двух противоположных углов прямоугольника, в который вписан круг: (x_1, y_1) — это координаты левого верхнего угла, а (x_2, y_2) — координаты правого нижнего угла.

Теперь можно создать массив из пульсаров:

```
NUMBER_OF_PULSARS = 15
pulsars = []
for i in range(NUMBER_OF_PULSARS):
  pulsars.append( TPulsar(
          randint(0, SCREEN_WIDTH),
          randint(0, SCREEN_HEIGHT),
          randint(10, 20) ) )
```

Отметим, что изображения пульсаров сразу добавляются на холст. Это происходит в конструкторе класса **TBlackHole** при создании объекта-круга с помощью функции circle из модуля graph.

Чтобы пульсары обновлялись при моделировании, в процедуре update для каждого из этих объектов нужно вызвать метод update:

```
def update():
  for bh in blackHoles:
    bh.update()
  for ps in pulsars:
    ps.update()
```

Полиморфизм

Интересно, что мы можем записывать чёрные дыры и пульсары в один и тот же массив, и всё равно программа будет работать правильно:

```
NUMBER_OF_BLACKHOLES = 10
allObjects = []
for i in range(NUMBER_OF_BLACKHOLES):
  allObjects.append( TBlackHole(
          randint( 0, SCREEN_WIDTH ),
          randint( 0, SCREEN_HEIGHT ),
          randint(10, 20) ) )
NUMBER_OF_PULSARS = 15
for i in range(NUMBER_OF_PULSARS):
  allObjects.append( TPulsar(
          randint( 0, SCREEN_WIDTH ),
          randint( 0, SCREEN_HEIGHT ),
          randint( 10, 20) ) )
```

Теперь массив `allObjects` содержит как чёрные дыры, так и пульсары. Причём каждый элемент массива «знает» свой тип. Это легко проверить с помощью такого цикла:

```
for obj in allObjects:
  print( type(obj) )
```

Для первых 10 элементов мы увидим сообщение

```
<class '__main__.TBlackHole'>
```

а для остальных:

```
<class '__main__.TPulsar'>
```

И поэтому в процедуре `update` пульсары и можно обрабатывать вместе, в одном цикле:

```
def update():
  for obj in allObjects:
    obj.update()
```

Для каждого объекта, попавшего в переменную `obj`, программа во время выполнения определит тип и вызовет метод `update` того класса, к которому этот объект принадлежит. Эта особенность — проявление *полиморфизма*.

Выводы

- При объявлении класса-наследника после имени нового класса в скобках указывают имя базового класса.
- В конструкторе класса-наследника сначала нужно вызвать конструктор базового класса.
- Чтобы класс не был абстрактным, все абстрактные методы базового класса должны быть определены (реализованы).
- Переопределение метода — это создание в классе-наследнике новой версии метода, определённого ранее для базового класса.

- Ссылки на объекты разных классов можно хранить в одном массиве. Если, например, для каждого элемента массива вызвать метод `update`, программа определит тип конкретного объекта и вызовет метод `update` того класса, к которому этот объект принадлежит. Эта особенность — проявление полиморфизма.

Вопросы и задания

1. *Проект.* Постройте полную программу (из параграфа), которая выводит на экран чёрные дыры и пульсары и выполняет анимацию.
2. *Проект.* Напишите вариант программы из предыдущего задания, в котором классы **`TBlackHole`** и **`TPulsar`** не связаны в иерархию, а независимы. Сравните её с вариантом, который рассматривался в параграфе. Какой подход вам больше нравится и почему?
3. *Проект.* Добавьте в программу из задания 1 ещё один класс неподвижных объектов, которые изображаются квадратом или ромбом.
4. *Проект.* Добавьте в программу из задания 1 ещё один класс неподвижных объектов, которые при анимации меняют цвет случайным образом.

§ 7
Классы-наследники (II)

Ключевые слова:

- базовый класс
- класс-наследник
- защищённое поле
- абстрактный класс
- суперкласс
- переменные модуля

Подвижные объекты

Класс *Подвижный объект* из нашей игры отличается от базового класса *Игровой объект* тем, что у него есть два дополнительных свойства — скорость и курс (направление движения). Будем хранить их в защищённых полях `_v` и `_course` (по-английски — курс). Курсом будем называть угол α между вектором скорости объекта и осью *OX*, измеренный в радианах (рис. 1.11).

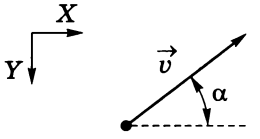


Рис. 1.11

Конструктор нового класса имеет 6 параметров (не считая ссылки `self` на сам объект):

```
class TMovingObject( TGameObject ):
  def __init__( self, x, y, width, height,
                v, course ):
    TGameObject.__init__( self, x, y, width, height )
    self._v = v
    self._course = course
```

В самом начале вызывается конструктор базового класса **TGameObject**. Он создаёт в памяти новый объект и сохраняет его свойства `x`, `y`, `width` и `height`. Затем выполняются дополнительные действия по созданию объекта класса **TMovingObject** — сохранение заданной начальной скорости и курса во внутренних полях.

У класса **TMovingObject** должен быть метод для движения — `move`. При выполнении этого метода нужно вычислить горизонтальное и вертикальное смещения объекта и передвинуть изображение объекта на экране в нужное место.

Пусть наш объект движется со скоростью v курсом α (рис. 1.12).

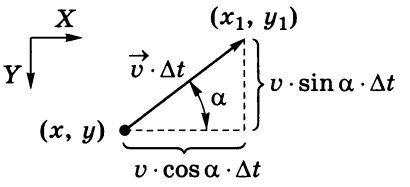


Рис. 1.12

За время Δt объект проходит расстояние $v \cdot \Delta t$, так что из точки (x, y) перемещается в точку (x_1, y_1), где:

$$x_1 = x + v \cdot \cos\alpha \cdot \Delta t,$$
$$y_1 = y - v \cdot \sin\alpha \cdot \Delta t.$$

Из-за того что ось OY на экране направлена вниз, во второй формуле появился знак «минус».

Для использования математических функций в начале программы нужно подключить модуль `math`:

```
import math
```

Удобно принять за единицу времени интервал между двумя последовательными вызовами процедуры `update` (она вызывается по таймеру), тогда $\Delta t = 1$. Будем считать, что в поле `_course` хранится значение курса — угла α — в радианах. Тогда метод `move` можно написать так:

```
class TMovingObject( TGameObject ):
  ...
  def move( self ):
    dx = self._v*math.cos( self._course )
```

```
    dy = -self._v*math.sin( self._course )
    self._x += dx
    self._y += dy
    moveObjectBy( self._image, dx, dy )
```

Сначала вычисляются смещения по осям x и y, они записываются в локальные переменные dx и dy. В следующих строках изменяются поля объекта _x и _y и вызывается функция moveObjectBy из модуля graph, которая перемещает круг на холсте.

Обратите внимание, что создать объект класса **TMovingObject** нельзя — у него, как и у базового класса, нет метода update. Поэтому **TMovingObject** — это тоже *абстрактный* класс, он объединяет все свойства подвижных объектов и служит только для создания классов-наследников.

Космические корабли

Космический корабль — это подвижный объект, т. е. наследник класса **TMovingObject**:

```
class TSpaceship( TMovingObject ):
  ...
```

В конструкторе класса нужно, прежде всего, вызвать конструктор базового класса **TMovingObject**, а затем — создать изображение (круг с синей заливкой) и сохранить ссылку на него в поле _image:

```
class TSpaceship( TMovingObject ):
  def __init__( self, xCenter, yCenter,
                radius, v, course ):
    TMovingObject.__init__( self,
        xCenter, yCenter, 2*radius, 2*radius,
        v, course )
    brushColor( "blue" )
    self._image = circle ( xCenter, yCenter, radius )
```

Этот класс уже не должен быть абстрактным, так как нам нужно создавать объекты класса. Поэтому необходимо определить для него метод update. В этом методе передвинем объект, вызвав метод move, и затем проверим, не столкнулся ли корабль со стенкой. Проверку столкновения будет выполнять метод bounce (по-английски — отскок), который мы скоро напишем.

```
class TSpaceship( TMovingObject ):
  ...
  def update( self ):
    self.move()    # движение
    self.bounce()  # проверка столкновения
```

Обратите внимание, что метода move в классе **TSpaceship** нет. Но он есть у базового класса **TMovingObject**, поэтому при вызове self.move() на самом деле будет выполнен метод базового класса.

Отскок происходит тогда, когда объект касается границ холста. Границы объекта можно определить с помощью функции coords:

```
x1, y1, x2, y2 = coords( self._image )
```

Здесь (x_1, y_1) — координаты левого верхнего угла объекта на холсте и (x_2, y_2) — координаты его правого нижнего угла.

При отталкивании от левой и правой границ холста по законам физики (угол отражения равен углу падения), нужно вычесть текущий курс из развёрнутого угла величиной π радиан:

```
if x1 <= 0  or  x2 >= SCREEN_WIDTH:
   self._course = math.pi - self._course
```

При отталкивании от верхней и нижней границ мы просто изменяем знак курса на противоположный:

```
if y1 <= 0  or  y2 >= SCREEN_HEIGHT:
   self._course = - self._course
```

Теперь можно полностью написать метод bounce:

```
class TSpaceship( TMovingObject ):
  ...
  def bounce( self ):
    x1, y1, x2, y2 = coords( self._image )
    if x1 <= 0  or  x2 >= SCREEN_WIDTH:
       self._course = math.pi - self._course
    if y1 <= 0  or  y2 >= SCREEN_HEIGHT:
       self._course = - self._course
```

Класс **TSpaceship** готов, и теперь можно создавать космические корабли, добавляя их в тот же массив allObjects, где хранятся ссылки на другие объекты:

```
NUMBER_OF_SPACESHIPS = 10
SIZE_OF_SPACESHIP = 5
for i in range(NUMBER_OF_SPACESHIPS):
  allObjects.append( TSpaceship(
     randint( 0, SCREEN_WIDTH ),      # центр
     randint( 0, SCREEN_HEIGHT ),
     SIZE_OF_SPACESHIP,               # размер
     randint( 1, 5 ),                 # скорость
     randint( 0, 359 )*math.pi/180 ) # направление
     )
```

Константы NUMBER_OF_SPACESHIPS и SIZE_OF_SPACESHIP определяют количество и размер космических кораблей. Скорость корабля —

случайное целое число на отрезке [1; 5]. Курс выбирается случайным образом в диапазоне от 0° до 359° и сразу переводится в радианы:

```
randint( 0, 359 )*math.pi/180
```

Вспомните, что в процедуре update для каждого элемента массива allObjects вызывается метод update. Благодаря полиморфизму, для космических кораблей сработает метод update из класса **TSpaceship**.

Осталось написать класс *Странник* (**TRanger**) — это специальный космический корабль, который меняет направление движения. Приведём сразу код класса **TRanger**, который является наследником класса **TSpaceship**:

```
class TRanger( TSpaceship ):
  def __init__( self, xCenter, yCenter,
                radius, v, course ):
    TSpaceship.__init__( self, xCenter, yCenter,
                         radius, v, course )
    changeFillColor( self._image, "yellow" )
  def update( self ):
    if randint(1, 20) == 1:
       self._course = randint(0,359)*math.pi/180
    super().update()
```

В конструкторе класса **TRanger** сначала вызывается конструктор базового класса **TSpaceship**, а затем цвет заливки круга изменяется на жёлтый.

В начале метода update мы изменяем направление движения — значение поля _course. Причем делаем это только тогда, когда вызов функции randint(1,20) вернёт значение, равное 1, т. е. в среднем 1 раз за 20 периодов обновления картинки.

После изменения направления нужно передвинуть объект и проверить, не столкнулся ли он со стенками. Но всё это уже есть в методе update базового класса **TSpaceship**! Поэтому нам нужно просто вызвать этот метод базового класса — в языке Python он ещё называется *суперклассом*:

```
super.update()
```

Закончить программу вы можете самостоятельно.

Модуль с классами

Большие программы обычно разбивают на модули, каждый модуль хранится как отдельный файл. В модуль объединяются функции, связанные между собой. Такой подход используется как в классическом программировании, так в ООП.

В нашей программе в отдельный модуль gameobjects (в файл gameobjects.py) можно вынести описания всех классов:

```
class TGameObject:
  ...
class TBlackHole( TGameObject ):
  ...
class TPulsar( TBlackHole ):
  ...
class TMovingObject( TGameObject ):
  ...
class TSpaceship( TMovingObject ):
  ...
class TRanger( TSpaceship ):
  ...
```

Поскольку в этих классах используются функции из других модулей (graph, math и random), в начале модуля нужно поместить команды импорта:

```
from graph import *
from random import randint
import math
```

В основной программе импортируем модуль gameobjects:

```
from gameobjects import *
```

Вроде бы всё сделано правильно, но при запуске программы вы получите сообщение об ошибке: переменная SCREEN_WIDTH не найдена. Дело в том, что глобальные переменные в Python — это переменные *модуля*, а не всей программы. То есть модуль gameobjects ничего «не знает» о переменных, объявленных в файле с основной программой. Поэтому размеры игрового поля, которые нужны для выполнения отскока в методе bounce, в модуле gameobjects неизвестны.

Самый простой способ решения этой проблемы — добавить в файл gameobjects.py новые переменные и процедуру для их изменения:

```
SCREEN_WIDTH = 800
SCREEN_HEIGHT = 600
def gameScreenSize( width, height ):
  global SCREEN_WIDTH, SCREEN_HEIGHT
  SCREEN_WIDTH = width
  SCREEN_HEIGHT = height
```

По умолчанию в этом модуле считается, что поле имеет размер 800 × 600 пикселей. Но эти значения всегда можно поменять, вызвав процедуру gameScreenSize:

```
gameScreenSize( SCREEN_WIDTH, SCREEN_HEIGHT )
```

Таким образом, мы можем из основной программы изменить значения глобальных переменных модуля.

Выводы

- В методах класса-наследника можно вызывать открытые методы базового класса.
- Чтобы в методе класса-наследника обратиться к объекту базового класса, используют метод `super`.
- В больших программах описания классов выделяют в модули, которые хранятся как отдельные файлы.
- Глобальные переменные в языке Python — это переменные модуля, а не всей программы.

Вопросы и задания

1. *Проект*. Соберите в одном файле программу из параграфа, которая выводит на экран объекты четырёх типов и выполняет их анимацию.
2. *Проект*. Измените программу из параграфа так, чтобы космический корабль, который попал в чёрную дыру или в пульсар, уничтожался. Вместо него должен появиться новый корабль в случайном месте.
3. Переделайте программу из параграфа так, чтобы описание всех классов находилось в отдельном модуле.

*4. *Проект*. Измените программу из параграфа так, чтобы чёрная дыра постепенно уменьшалась «без пищи», т. е. тогда, когда в неё долгое время не попадают корабли.

*5. *Проект*. Добавьте в игру из параграфа объекты класса **`TDestroyer`** — специальные боевые космические корабли, которые охотятся за кораблями и странниками и уничтожают их при встрече.

6. *Проект*. Измените поведение объектов класса **`TDestroyer` из предыдущего задания так, чтобы они двигались к ближайшему объекту типа **`TSpaceship`** или **`TRanger`** и уничтожали его при встрече.

**7. *Проект*. Придумайте и запрограммируйте собственную игру, где будут неподвижные и движущиеся объекты. Один объект должен управляться игроком с клавиатуры.

§ 8
Событийно-ориентированное программирование

Ключевые слова:

- сообщение
- событие
- очередь сообщений
- цикл обработки сообщений
- обработчик события
- форма
- виджет
- приложение

Особенности современных прикладных программ

Большинство современных прикладных программ управляются с помощью графического интерфейса. Вы, несомненно, знакомы с понятиями «окно программы», «кнопка», «флажок», «переключатель», «поле ввода», «полоса прокрутки» и т. п. Такие оконные системы чаще всего построены на основе объектно-ориентированного подхода. И сами окна программ, и все их элементы — это объекты, которые обмениваются данными, посылая друг другу сообщения.

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

Сообщение содержит:

- адрес объекта, которому посылается сообщение;
- числовой код сообщения;
- параметры (дополнительные данные), например координаты щелчка мышью или код нажатой клавиши.

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

В классических программах (рис. 1.13), последовательность действий заранее определена — основная программа выполняется строка за строкой, вызывая процедуры и функции; все ветвления выполняются с помощью условных операторов.

Рис. 1.13

В современных программах порядок действий определяется пользователем, другими программами или поступлением новых данных из внешнего источника (например, из сети). Пользователь текстового редактора может щёлкать по любым кнопкам и выбирать любые пункты меню в произвольном порядке. Программа-сервер, передающая данные с веб-сайта на компьютер пользователя, начинает действовать только при поступлении очередного запроса. При программировании сетевых

игр нужно учитывать взаимодействие многих объектов, информация о которых передаётся по сети в случайные моменты времени.

Во всех этих примерах программа должна «включаться в работу» только тогда, когда получит условный сигнал, т. е. когда произойдёт 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 жёстко заданным алгоритмом. Поэтому говорят, что программы управляются событиями, а соответствующий стиль программирования называют *событийно-ориентированным*, т. е. основанным на обработке событий.

Нормальный режим работы событийно-ориентированной программы — это *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 создаёт отдельную очередь сообщений и помещает в неё все сообщения, предназначенные этой программе (рис. 1.14).

Клавиатура, мышь, ...

Рис. 1.14

Основная программа выбирает очередное сообщение из очереди и вызывает специальную процедуру — обработчик этого сообщения (если он есть). Когда пользователь закрывает окно программы, ей посылает-

ся особое сообщение, при получении которого цикл обработки сообщений (и работа всей программы) завершается (рис. 1.15).

Таким образом, главная задача программиста — написать содержание обработчиков всех сообщений, на которые должна реагировать программа. Если для полученного сообщения нет обработчика, вызывается обработчик по умолчанию. Чаще всего при этом сообщение игнорируется.

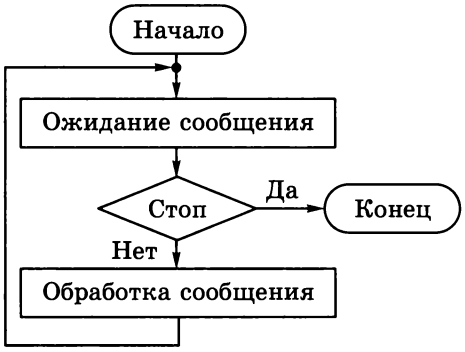


Рис. 1.15

Ещё раз подчеркнём, что последовательность вызовов обработчиков сообщений может быть любой, она зависит от действий пользователя и сигналов, поступающих с внешних устройств.

Возможно, вы вспомнили, что мы уже применяли такой подход раньше, когда определяли обработчики событий «срабатывание таймера» и «нажатие на клавишу».

Программы с графическим интерфейсом

В этом разделе рассматриваются основные принципы построения программ с графическим интерфейсом на языке Python. Для этого мы будем использовать графическую библиотеку `tkinter`, которая входит в стандартную библиотеку Python. Существуют и другие библиотеки для разработки интерфейсов на Python: `wxPython`, `PyGTK`, `PyQt`. Их нужно устанавливать отдельно.

В программе с графическим интерфейсом может быть несколько окон, которые обычно называют *формами*[1]. На форме размещаются элементы графического интерфейса: поля ввода, кнопки, списки и др. Они называются *виджетами* (англ. *widget* — украшение, элемент) или *компонентами*. Каждый компонент — это объект определённого класса, у которого есть свойства и методы.

Для каждого события, которое происходит с компонентом, можно задать процедуру-обработчик. Она будет автоматически вызвана, когда произойдёт соответствующее событие. Примеры таких событий — это щелчок мышью на кнопке (или другом объекте), выбор элемента списка, изменение размеров формы, нажатие клавиши на клавиатуре и т. д.

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

Далее в этой главе основное внимание будет уделяться принципам проектирования программ с графическим интерфейсом, а не особенностям библиотеки tkinter. Мы будем применять так называемую «обёртку» simpletk — специальный модуль, который упрощает использование сложной библиотеки[1].

Простейшая программа

Простейшая программа с графическим интерфейсом состоит из трёх строк:

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

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

В последней строке с помощью метода run (по-английски — запуск) запускается цикл обработки сообщений, скрытый внутри этого метода. Таким образом, здесь тоже использован принцип инкапсуляции (скрытия внутреннего устройства).

Приведённую выше программу можно запустить, и вы увидите окно, показанное на рис. 1.16.

Рис. 1.16

Свойства формы

Класс TApplication — это класс-наследник класса Tk библиотеки tkinter. У него есть методы, позволяющие управлять свойствами окна. Например, можно изменить начальное положение окна на экране с помощью свойства position:

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

Позиция окна — это *кортеж*, состоящий из x-координаты и y-координаты левого верхнего угла окна на экране. Обе координаты задаются в пикселях.

---

[1] Модуль simpletk можно скачать на сайте автора **http://kpolyakov.spb.ru/school/probook/python.htm.**

Аналогично изменяются и размеры окна:

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

Здесь первый элемент кортежа — ширина окна, а второй — высота окна (в пикселях).

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

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

разрешает только изменение ширины (первый аргумент — `True`), а изменить высоту окна будет невозможно (второй аргумент — `False`).

С помощью свойств `minsize` и `maxsize` можно установить минимальные и максимальные размеры формы:

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

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

Обработчик события

Рассмотрим простой пример. Многие программы запрашивают подтверждение, когда пользователь завершает их работу. Это делается для того, чтобы по ошибке не потерять нужные данные.

Когда пользователь закрывает окно, происходит событие «запрос на закрытие окна». При обработке этого события нужно спросить пользователя, не ошибся ли он, т. е. выдать сообщение в диалоговом окне (рис. 1.17) и обработать ответ.

Рис. 1.17

Работа программы завершается только в том случае, если пользователь щёлкнет по кнопке *OK*.

Для вывода диалогового окна можно использовать готовую функцию `askokcancel` из модуля `messagebox` пакета `tkinter`. По-английски *ask* — спросить, *OK* — кнопка ОК, *cancel* обозначает кнопку «Отмена». Импортировать эту функцию (подключить к нашей программе) можно так:

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

Теперь можно написать процедуру-обработчик:

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

Функция `askokcancel` принимает два аргумента: заголовок окна и сообщение для пользователя. Она возвращает ненулевое значение, если пользователь щёлкнул по кнопке *ОК*. В этом случае вызывается метод `destroy` (в переводе с английского — разрушить): окно удаляется из памяти, и работа программы завершается.

Теперь нужно установить обработчик, т. е. связать только что написанную процедуру с нужным событием:

```
app.onCloseQuery = askOnExit
```

Здесь `onCloseQuery` — название обработчика события (от англ. *on close query* — «при запросе на закрытие»). Справа от оператора присваивания записано название нашей процедуры-обработчика.

Выводы

- Обмен данными между современными программами происходит с помощью сообщений.
- Сообщение — это блок данных определённой структуры, который используется для обмена информацией между объектами.
- Событие — это переход какого-либо объекта из одного состояния в другое.
- Современные программы с графическим интерфейсом основаны на обработке событий, вызванных действиями пользователя и поступлением данных из других источников. События могут происходить в любой последовательности.
- С любым событием можно связать процедуру — обработчик события, — которая будет вызываться при наступлении этого события.
- Окна в программе с графическим интерфейсом называют формами или окнами верхнего уровня.
- Элементы графического интерфейса (поля ввода, списки, кнопки и др.) называются виджетами или компонентами.
- Для построения программ с графическим интерфейсом в Python можно использовать библиотеку `tkinter`.

Вопросы и задания

1. Что такое графический интерфейс?
2. Как графический интерфейс связан с объектно-ориентированным подходом к программированию?
3. Что такое сообщение? Какие данные в него входят?
4. Что такое широковещательное сообщение?

5. Чем принципиально отличаются современные программы от классических?
6. Что такое событие? Какое программирование называют событийно-ориентированным?
7. Что такое обработчик события?
8. Объясните разницу между понятиями «событие» и «сообщение».
9. В чём различие между работой событийно-ориентированной и «обычной» программы?
10. Что такое форма?
11. В чём проявляется объектно-ориентированный подход к разработке интерфейса?
12. Что такое объект-приложение?
13. Почему в основной программе, приведённой в параграфе, не виден цикл обработки сообщений?
14. Назовите известные вам свойства формы. Как можно их изменять?
15. *Проект.* Постройте программу, которая запрашивает разрешение на завершение работы. Попробуйте изменять свойства главного окна.

*16. *Проект.* Найдите в дополнительных источниках информацию о других свойствах главного окна `Toplevel` и попробуйте их изменять.

*17. *Проект.* Найдите в дополнительных источниках информацию о дугих стандартных диалогах, входящих в модуль `messagebox`. Попробуйте их использовать.

Интересные сайты

kpolyakov.spb.ru/school/probook/python.htm — модуль `simpletk` для создания приложений с графическим интерфейсом на Python

wxpython.org — библиотека `wxPython`

pygtk.org — библиотека `PyGTK`

riverbankcomputing.com/software/pyqt/intro — библиотека `PyQt`

§9
Использование компонентов (виджетов)

Ключевые слова:

- виджет
- компонент
- родительский объект
- обработчик события
- исключение

Программа с компонентами

В предыдущем параграфе мы научились работать с главным окном программы (окном верхнего уровня). Для того чтобы пользователь мог получать информацию от программы и управлять её работой (вводить

исходные данные), в это окно нужно добавить элементы интерфейса — надписи (метки), кнопки, выпадающие списки, флажки, переключатели и т. п. Каждый такой элемент — это объект, имеющий свой набор свойств и методов. Объекты, размещаемые на форме, называются *виджетами* или *компонентами*.

Мы построим программу для просмотра изображений, которая умеет открывать графические файлы и показывать их в левом верхнем углу или по центру рабочей области (рис. 1.18 и см. цветной рисунок на обороте обложки).

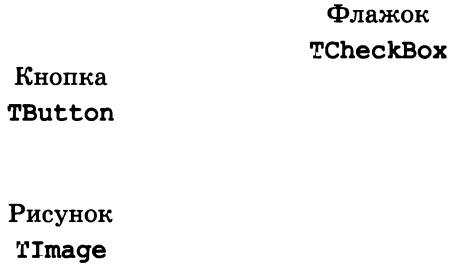


Рис. 1.18

К сожалению, наша программа будет гарантированно работать только с файлами формата GIF. С остальными распространёнными форматами (JPEG, PNG, BMP) библиотека `tkinter` работать не умеет. Для поддержки файлов этих форматов нужно установить бесплатную библиотеку `Pillow` (`PIL`). Модуль `simpletk` при загрузке автоматически подключит её, если она установлена.

На форме (см. рис. 1.18) размещены четыре компонента:

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

Начнем с простой программы, которая устанавливает свойства главного окна:

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

Напомним, что вызов метода `run` должен стоять в самом конце текста программы. Поэтому все команды, рассмотренные далее, нужно добавлять выше него.

Добавим верхнюю панель, на которой будут расположены кнопка и флажок:

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

Конструктору класса **TPanel** передаётся несколько параметров:

app — ссылка на родительский объект — главное окно;

relief — объёмный вид, значение "raised" означает «приподнятый»; этот параметр может принимать также значения "flat" (плоский, по умолчанию), "sunken" («утопленный») и др.;

height — высота в пикселях;

bd (от англ. *border* — граница) — толщина границы в пикселях.

Родительский объект — это объект, отвечающий за перемещение и вывод на экран всех своих дочерних объектов. Здесь панель — это дочерний объект для главного окна.

Разместим панель на форме, прижав её к верхней границе:

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

Свойство align задаёт расположение панели внутри родительского окна: "top" означает прижать вверх, "bottom" — вниз, "left" — влево, "right" — вправо.

Далее нужно добавить на панель компоненты TButton (кнопка) и TCheckBox (флажок). Для них «родителем» уже будет не главное окно, а панель panel. Начнём с кнопки:

```
openBtn = TButton( panel, width = 13,
                   text = "Открыть файл" )
openBtn.position = (5, 5)
```

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

Используя тот же подход, создаём и размещаем флажок:

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

Сейчас уже можно запустить программу и убедиться, что все компоненты появляются на форме, но пока не работают. И это неудивительно, ведь мы не определили, как они должны реагировать на события.

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

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

Его «родителем» будет главное окно app, параметр bg (от англ. *background* — фон) задаёт цвет холста (англ. *white* — белый). Затем «упаковываем» этот объект в окно так, чтобы он заполнял всю оставшуюся область по вертикали и горизонтали:

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

Теперь построим обработчики событий. Нас интересуют два события:

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

Сначала определим, что нужно сделать в случае щелчка мышью по кнопке. Псевдокод выглядит так:

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

Для выбора файла используем стандартный диалог операционной системы. Его вызывает функция askopenfilename из модуля filedialog библиотеки tkinter. Этот модуль нужно импортировать в начале программы:

```
from tkinter import filedialog
```

Теперь вызываем функцию askopenfilename и передаём ей расширения, которые будут показаны в выпадающем списке *Тип файла*:

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

Именованный аргумент filetypes — это список, составленный из кортежей. Каждый кортеж содержит два элемента: текстовое описание и маску для выбора имён файлов.

После вызова этой функции в переменную fname будет записан результат её работы — имя выбранного файла или пустая строка, если пользователь отказался от выбора (нажал на кнопку *Отмена*). Если файл всё-таки выбран (строка fname не пустая), записываем имя файла в свойство picture объекта **TImage**:

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

При изменении этого свойства файл будет автоматически загружен и выведен на экран, эту операцию выполняет класс **TImage**. Обратите внимание, что пустая строка в Python воспринимается как ложное значение. Поэтому если пользователь не выбрал файл, условие в операторе **if** будет ложно и рисунок не изменится.

Теперь можно составить всю процедуру-обработчик:

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

Она должна принимать единственный параметр sender — ссылку на объект, с которым произошло событие. Эта ссылка нужна тогда, когда один обработчик «обслуживает» несколько компонентов. Тогда в обработчике мы сможем определить, какой из них стал источником события. В нашем случае эта ссылка не нужна, и мы её использовать не будем.

Установим обработчик события: сделаем так, чтобы после щелчка по кнопке вызывалась процедура selectFile. Свойство, определяющее обработчик события «щелчок мышью», называется onClick (от англ. *on click* — в ответ на щелчок):

```
openBtn.onClick = selectFile
```

Если сейчас запустить программу, то можно проверить, как загружаются файлы.

Остаётся добавить второй обработчик события, который при изменении состояния флажка centerCb переключает режим вывода рисунка. Напишем код такого обработчика:

```
def cbChanged( sender ):
  image.center = sender.checked
  image.redrawImage()
```

У объекта **TImage** есть логическое свойство center, которое должно быть равно True, если нужно вывести рисунок по центру поля и False, если рисунок выводится в левом верхнем углу холста.

Вместе с тем, у флажка **TCheckBox** есть логическое свойство checked, которое равно True, если флажок включён.

В первой строке процедуры cbChanged мы задали значение свойства center в соответствии с состоянием флажка. Затем вызывается метод redrawImage (от англ. *redraw image* — перерисовать рисунок), который выводит рисунок в новой позиции.

Этот обработчик нужно установить, т. е. подключить к событию «изменение состояния флажка». Для этого необходимо записать ссылку на него (адрес обработчика) в свойство onChange (от англ. *on change* — при изменении):

```
centerCb.onChange = cbChanged
```

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

Отметим следующие важные особенности:

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

Новый класс: всё в одном

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

```
class TImageViewer ( TApplication ):
  ... # здесь будет описание класса TimageViewer

app = TImageViewer()
app.run()
```

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

Начнём с конструктора:

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

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

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

Обработчики событий — функции `selectFile` и `cbChanged` — тоже должны быть членами класса **TImageViewer**, поэтому их первым параметром будет `self`, а вторым — ссылка `sender` на объект — источник события:

```
class TImageViewer ( TApplication ):
  ... # здесь должен быть конструктор __init__

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

  def cbChanged( self, sender ):
    self.image.center = sender.checked
    self.image.redrawImage()
```

Поскольку при создании объекта в конструкторе мы запомнили ссылки на все элементы в полях объекта, теперь можно к ним обращаться через ссылку **self** для изменения свойств.

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

Ввод и вывод данных

Во многих программах нужно, чтобы пользователь вводил текстовую или числовую информацию. Чаще всего для этого применяют поле ввода — компонент **TEdit** (наследник класса **Entry** библиотеки `tkinter`). Для доступа к введённой строке используется его свойство `text` (в переводе с английского — текст).

Построим программу для перевода составляющих цвета модели RGB в соответствующий шестнадцатеричный код, который применяется для задания цвета в языке HTML[1] (рис. 1.19 и см. цветной рисунок на обороте обложки).

Код цвета в языке HTML начинается знаком #, за ним записывают без разделителей три двузначных числа в шестнадцатеричной системе счисления — яркости красной, зелёной и синей составляющих цвета в модели RGB. Например, при R = 123, G = 56 и B = 80 мы должны получить код #7B3850, поскольку $123 = 7B_{16}$, $56 = 38_{16}$ и $80 = 50_{16}$.

[1] Язык HTML (англ. *HyperText Markup Language* — язык разметки гипертекста) используется для оформления информации на веб-страницах.

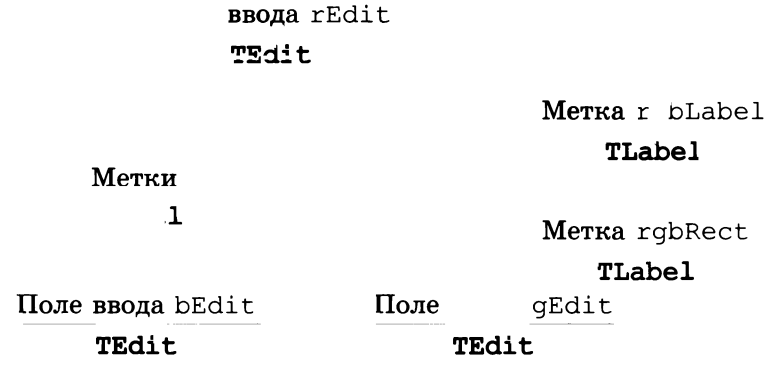


Рис. 1.19

На форме (см. рис. 1.19) расположены:

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

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

Во время работы программы будут использоваться:

- поля ввода rEdit, gEdit и bEdit;
- метка rgbLabel, с помощью которой будет выводиться результат — код цвета;
- метка без текста rgbRect — прямоугольник, закрашенный нужным цветом.

В качестве начальных значений в свойство text полей ввода можно записать любые целые числа от 0 до 255.

При изменении содержимого одного из трёх полей ввода нужно обработать введённые данные, вывести HTML-код цвета в свойство text метки rgbLabel, а также изменить цвет фона для метки rgbRect.

Обработчик события, которое происходит при изменении текста в поле ввода, называется onChange. Так как при изменении любого из трёх полей нужно выполнить одинаковые действия, можно установить один и тот же обработчик события onChange для всех трёх компонентов. Процедура-обработчик может выглядеть так:

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

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

В основной программе создаём объект-приложение с главным окном:

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

и метки слева от полей ввода:

```
fSans = ( "MS Sans Serif", 12 )
lblR = TLabel( app, text = "R =", font = fSans )
lblR.position = (5, 5)

lblG = TLabel( app, text = "G =", font = fSans )
lblG.position = (5, 30)

lblB = TLabel( app, text = "B =", font = fSans )
lblB.position = (5, 55)
```

Мы определили шрифт для меток с помощью параметра font. Его значение — это кортеж fSans, который в данном случае состоит из двух элементов: названия шрифта ("MS Sans Serif") и его размера в пунктах (12).

Далее создадим метку для вывода кода цвета:

```
fCourier = ( "Courier New", 16, "bold" )
rgbLabel = TLabel( app, text = "#000000",
                   font = fCourier, fg = "navy" )
rgbLabel.position = (100, 5)
```

и прямоугольник, который будет закрашен выбранным цветом:

```
rgbRect = TLabel( app, text = "", width = 15, height = 3 )
rgbRect.position = (105, 35)
```

Размеры меток — ширина (англ. *width*) и высота (англ. *height*) — указываются не в пикселях, а в символах.

Шрифт для метки rgbLabel задаётся в виде кортежа fCourier из трёх элементов: названия шрифта ("Courier New"), размера (16) и свойства «жирный» (англ. — *bold*).

Параметр fg (от англ. *foreground* — цвет переднего плана) при вызове конструктора определяет цвет символов ("navy" — тёмно-синий, от англ. *navy* — военно-морской).

Добавляем на форму три поля ввода:

```
rEdit = TEdit( app, font = fSans, width = 5 )
rEdit.position = (45, 5)
rEdit.text = "123"

gEdit = TEdit( app, font = fSans, width = 5 )
gEdit.position = (45, 30)
gEdit.text = "56"

bEdit = TEdit( app, font = fSans, width = 5 )
bEdit.text = "80"
bEdit.position = (45, 55)
```

Для каждого из них устанавливаем один и тот же обработчик onChange, который вызывается при любом изменении текста в поле ввода:

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

После этого запускаем программу:

```
app.run()
```

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

При желании можно переписать программу в стиле ООП, как это мы сделали в конце предыдущего примера. Попробуйте выполнить это задание самостоятельно.

Обработка ошибок

Если в предыдущей программе пользователь введёт в поле ввода не цифры, а другие символы или пустую строку, программа выдаст сообщение о необработанной ошибке и аварийно завершит работу. Хорошая программа никогда не должна завершаться аварийно. Поэтому нужно обрабатывать все ошибки, которые можно заранее предусмотреть.

В современных языках программирования для обработки ошибок применяют *исключения*. Исключение — это исключительная (ошибочная, аварийная, непредвиденная) ситуация, из-за которой программа не может нормально продолжать работу.

Все «опасные» участки кода, при выполнении которых может возникнуть ошибка, нужно поместить в блок **try-except**:

```
try:
  # "опасные" команды
except:
  # обработка ошибки
```

Слово **try** по-английски означает «попытаться», **except** — «исключать». Программа попадает в блок **except** только тогда, когда при выполнении команд между **try** и **except** произошла ошибка.

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

Улучшенный обработчик с защитой от неправильного ввода принимает вид:

```
def onChange( sender ):
  try:
    # получить данные из полей ввода
    s = "#{:02X}{:02X}{:02X}".format(r, g, b)
    bkColor = s
  except:
    s = "?"
    bkColor = "SystemButtonFace"
  rgbLabel.text = s
  rgbRect.background = bkColor
```

Здесь переменная `s` обозначает шестнадцатеричный код цвета (в виде символьной строки), а переменная `bkColor` — цвет прямоугольника.

Если при попытке получить новый код цвета происходит ошибка, то выполняются две команды в блоке **except**: в переменные `s` и `bkColor` записываются значения, сообщающие о неудаче.

В модели RGB все составляющие цвета должны быть не меньше 0 и не больше 255, т. е. должны находиться в диапазоне `range(256)`. Поэтому в обработчик полезно добавить ёщё проверку допустимости введённых чисел:

```
def onChange( sender ):
  s = "?"
  bkColor = "SystemButtonFace"
  try:
    r = int( rEdit.text )
    g = int( gEdit.text )
    b = int( bEdit.text )
    if r in range(256) and \
       g in range(256) and b in range(256):
      s = "#{:02x}{:02x}{:02x}".format(r, g, b)
      bkColor = s
```

```
except:
  pass
rgbLabel.text = s
rgbRect.background = bkColor
```

Здесь в самом начале процедуры-обработчика мы записываем значения по умолчанию в переменные `s` и `bkColor`. В блоке `except` помещён «пустой» оператор **pass**, т. е. ошибки игнорируются. Если пользователь ввёл ошибочные данные (не числа или числа вне отрезка [0; 255]), программа не выполнит команды в теле оператора **if**, и значения переменных `s` и `bkColor` не изменятся.

Выводы

- Для организации диалога с пользователем на форме размещают объекты, называемые виджетами или компонентами.
- Любой компонент может реагировать на некоторый набор событий. С каждым событием, которое нужно обрабатывать, необходимо связать обработчик события.
- Обработчик события — это процедура, которая получает управление каждый раз, когда происходит данное событие.
- Один обработчик события может быть связан с несколькими компонентами. Ссылка на конкретный компонент, который вызвал событие, передаётся в обработчик события как первый параметр.
- Для выбора файлов на диске можно использовать стандартные диалоги из модуля `filedialog` библиотеки `tkinter`.
- Всё приложение с графическим интерфейсом можно оформить как один класс. Компоненты добавляются на форму в конструкторе, а обработчики становятся методами нового класса.
- Для обработки ошибок используют механизм исключений. Команды, которые могут вызвать ошибку, заключаются в отдельный блок **try**. В следующем блоке, который начинается словом **except**, записывают команды, выполняемые в случае ошибки.

Вопросы и задания

1. Что такое компоненты? Зачем они нужны?
2. Объясните, как связаны компоненты и идея инкапсуляции.
3. Что такое родительский и дочерний объекты?
4. Какую роль играет свойство `align` в размещении элементов на форме?
5. Что такое стандартный диалог? Как его использовать?
6. Назовите основное свойство флажка (**TCheckBox**). Как его использовать?
7. Чем отличается метка от элемента **TEdit**?
8. Как обрабатываются ошибки в современных программах? В чём, на ваш взгляд, преимущества и недостатки такого подхода?

9. Почему в последнем варианте обработчика `onChange` (из параграфа) удобно присвоить значения переменным `s` и `bkColor` в начале программы, а не в блоке **except**?
10. *Проект.* Напишите программу для построения RGB-кода цвета в стиле ООП, убрав все действия по созданию формы в конструктор класса. Обработчики событий также должны быть членами класса.
11. *Проект.* Разработайте программу для перевода морских миль в километры (1 миля = 1852 м).
12. *Проект.* Разработайте программу для перевода суммы денег из рублей в другие валюты и обратно.
13. *Проект.* Разработайте программу для перевода чисел из десятичной системы в двоичную, восьмеричную и шестнадцатеричную.
14. *Проект.* Разработайте программу для вычисления информационного объёма рисунка по его размерам и количеству цветов в палитре.
15. *Проект.* Разработайте программу для вычисления информационного объёма звукового файла при известных длительности записи звука, частоте дискретизации и глубине кодирования (числу битов на один результат измерения).
16. *Проект.* Разработайте программу для решения системы двух линейных уравнений. Обратите внимание на обработку ошибок при вычислениях.

Интересные сайты

younglinux.info/tkinter.php — `tkinter`. Программирование графического интерфейса

effbot.org/tkinterbook/tkinter-index.htm — учебник по пакету `tkinter`

pypi.python.org/pypi/Pillow/ — библиотека `Pillow` для работы с изображениями в Python

§ 10
Создание компонентов

Ключевые слова:

- компонент
- наследование
- базовый класс
- поле
- метод
- обработчик события

Зачем это нужно?

Как вы видели в предыдущем параграфе, на практике нередко нужны поля ввода особого типа, с помощью которых можно вводить целые числа. Компонент **TEdit** разрешает вводить любые символы и представляет результат ввода как текстовое свойство `text`. Поэтому для

того, чтобы получить нужное нам поведение (ввод целых чисел), мы добавили обработку ошибок в процедуру onChange, а для перевода текстовой строки в число каждый раз использовали функцию int.

Если такие поля ввода нужны часто и в разных программах, можно избавиться от этих рутинных операций. Для этого создаётся 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.

Компонент для ввода целых чисел

Мы будем совершенствовать компонент **TEdit** (поле ввода), поэтому новый компонент **TIntEdit** будет наследником класса **TEdit**, а класс **TEdit** будет соответственно базовым классом для класса **TIntEdit**:

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

В стандартный класс **TEdit** нужно внести два изменения:

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

Сначала построим конструктор нового класса:

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

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

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

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

Последняя часть этого описания говорит о том, что для чтения введённого числового значения используется лямбда-функция, возвращающая значение скрытого поля __value. Метод записи __setValue сохраняет переданное число в поле __value, а затем записывает в свойство text в виде символьной строки.

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

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

  def __validate( self ):
    try:
      newValue = int( self.text )
      self.__value = newValue
      return True
    except:
      return False
```

При получении сигнала о событии «нужна проверка» обработчик пытается преобразовать новое значение свойства text в число. Если это удаётся, полученное число записывается в переменную __value и возвращается результат True. Если произойдёт ошибка, она будет перехвачена с помощью исключения, и управление будет передано в блок **except**, откуда функция вернёт значение False.

Готовый компонент лучше всего поместить в отдельный модуль, который можно назвать int_edit.py.

Программа с новым компонентом

Построим программу, использующую новый компонент. Она будет переводить целые числа из десятичной системы в шестнадцатеричную (рис. 1.20).

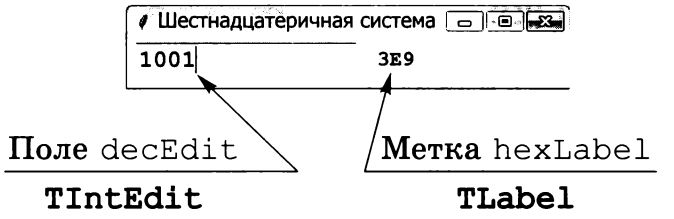


Рис. 1.20

Импортируем компонент **TIntEdit** из модуля int_edit:

```
from int_edit import TIntEdit
```

Создаём объект-приложение:

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

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

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

decEdit = TIntEdit( app, width = 12, font = f )
decEdit.position = ( 5, 5 )
decEdit.text = "1001"
```

Добавим обработчик события onChange для поля ввода:

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

decEdit.onChange = onNumChange
```

Этот обработчик читает значение свойства value объекта, вызвавшего событие, и выводит его на метку hexLabel в шестнадцатеричной системе (формат X). В строке формата можно написать и строчную букву x, тогда в записи шестнадцатеричного числа будут выводиться *строчные* буквы a–f вместо прописных A–F.

Выключатель с рисунком

Построим ещё один новый компонент — выключатель **TSwitch**, который имеет два положения: «включено» и «выключено». Он должен менять своё состояние по щелчку мышью (рис. 1.21 и см. цветной рисунок на обороте обложки).

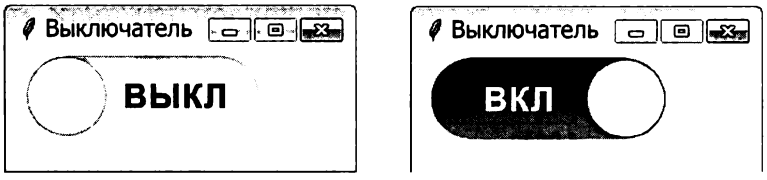


Рис. 1.21

В библиотеке graph есть компонент **TLabel** (метка), который может стать базовым классом для такого выключателя. Перечислим отличия нового компонента-выключателя от стандартной метки:

- содержит не текст, а рисунок;
- имеет состояние (включено или выключено);
- реагирует на щелчок мышью: изменяет состояние и изображение.

Нам нужно создать у компонента **TSwitch** два новых поля: в одном будет храниться состояние выключателя, а во втором — массив из двух изображений («выключено» и «включено»).

Начнём строить новый класс:

```
class TSwitch( TLabel ):
  def __init__( self, parent, images, **kw ):
    TLabel.__init__( self, parent, **kw )     # (1)
    self.__state = False                      # (2)
    self.__images = images                    # (3)
    self.config( image = self.__images[0] )   # (4)
```

Класс **TSwitch** — это наследник класса **TLabel**. Его конструктор принимает два обязательных параметра — ссылку на родительский объект `parent` и массив, состоящий из двух изображений (`images`).

В первую очередь вызываем конструктор базового класса **TLabel** (строка 1). Ему передаются все параметры, кроме изображений.

В строке 2 создаётся новое скрытое поле `__state` (по-английски — состояние), в которое записывается значение `False`. Это означает, что выключатель сразу после создания выключен.

Затем массив изображений, переданный конструктору, записывается в скрытое внутреннее поле `__images` (строка 3). Далее мы будем считать, что `__images[0]` — это изображение выключенного переключателя (соответствующее состоянию `False`), а `__images[1]` — изображение включённого переключателя (состояние `True`).

К счастью, объекты класса **TLabel** умеют работать с изображениями, и наша задача сводится к изменению изображения с помощью метода `config`. В строке 4 мы загружаем изображение с индексом 0 (состояние «выключено»).

Добавим в класс общедоступное свойство (**property**) с именем `state` — состояние выключателя:

```
class TSwitch(TLabel):
  # здесь должен быть конструктор
  def __setState(self, newState):
     if newState != self.__state:
       self.__state = newState
       self.config( image = self.__images[self.__state])
  state = property( lambda x: x.__state,  __setState )
```

Функции `property` передаются два аргумента: функция для чтения свойства и процедура для его изменения. Функция чтения просто возвращает значение скрытого поля `__state`. Процедура записи нового значения `__setState` проверяет, не совпадает ли новое состояние с

текущим, и если не совпадает, то изменяется как значение внутреннего поля, так и изображение. Новое изображение — это элемент массива __images, его номер определяется полем __state. Если переключатель выключен, загружается изображение с индексом 0, потому что при преобразовании логического значения False в целое число получается 0. Аналогично при включении переключателя загружается изображение с индексом 1.

Перехват щелчка мышью

При каждом щелчке мышью по выключателю он должен изменять своё состояние на обратное. Для этого достаточно вызвать уже готовый метод __setState, передав ему новое состояние, обратное текущему:

```
self.__setState( not self.__state )
```

Эту команду нужно применить тогда, когда пользователь щёлкнет по выключателю, т. е. в обработчике события «щелчок мышью» базового класса:

```
class TSwitch( TLabel ):
  def __init__( self, parent, images, **kw ):
    ...
    self.onClick = self.__doClick        # (1)

  def __doClick( self ):                 # (2)
    self.__setState( not self.__state )  # (3)
```

Процедура __doClick (строки 2 и 3) — это скрытый метод класса TSwitch. В строке 1 эта процедура сохраняется в свойстве onClick, т. е. устанавливается в качестве обработчика события «щелчок мышью».

Теперь можно написать основную программу и проверить работу компонента:

```
app = TApplication( "Выключатель" )
app.position = ( 200, 200 )
app.size = ( 300, 300 )

switchImages = [ PhotoImage("off.gif"),
                 PhotoImage("on.gif") ] # (1)
switch = TSwitch( app, switchImages )   # (2)
switch.position = ( 5, 5 )              # (3)
switch.state = True                     # (4)

app.run()
```

После создания главного окна приложения загружаем рисунки в массив switchImages (строка 1). Файлы off.gif и on.gif в текущем

каталоге должны содержать изображения выключенного и включённого переключателей. В строке 2 создаётся объект-выключатель, в строке 3 он размещается на форме. Далее устанавливаем начальное состояние «включено» (строка 4) и запускаем программу.

Новое свойство onChange

Компонент **TSwitch**, который мы построили, хорошо работает сам по себе. В реальных программах при изменении состояния этого элемента нужно выполнять некоторые действия, например включать и выключать какой-то режим, разрешать или запрещать обмен данными по сети и т. п. Для того чтобы позволить программисту, использующему компонент, установить свой обработчик этого события, добавим новое свойство onChange (от англ. *on change* — «при изменении»):

```
onChange = property( lambda x: x.__onChange, __setOnChange)
```

Это свойство связано с внутренним полем __onChange – там будет храниться ссылка на обработчик, который может установить программист. Этот обработчик нужно вызывать при изменении состояния переключателя. При чтении свойства будет возвращаться значение поля __onChange, а при записи — вызываться процедура __setOnChange, которая сохраняет ссылку на новый обработчик:

```
def __setOnChange( self, proc ):
  self.__onChange = proc
```

Обработчик будет вызываться при изменении состояния переключателя, т. е. внутри метода __setState. Все необходимые изменения в описании класса выделены фоном:

```
class TSwitch(TLabel):
  def __init__( self, parent, images, **kw ):
    ...
    self.__onChange = None                          # (1)

  def __setState( self, newState ):
    if newState != self.__state:
      ...
      if self.__onChange:                           # (2)
         self.__onChange()                          # (3)

  def __setOnChange( self, proc ):                  # (4)
    self.__onChange = proc                          # (5)

  onChange = property( lambda x: x.__onChange,
                       __setOnChange)               # (6)
```

В конструкторе (в строке 1) создаётся новое поле для хранения ссылки на обработчик, и в него записывается значение None — пока обработчика нет. В строках 2 и 3 этот обработчик вызывается, если он установлен (не равен None). Строки 4 и 5 — это процедура для изменения обработчика, а строка 6 — описание свойства onChange.

Теперь остаётся в основной программе определить процедуру-обработчик и связать её со свойством onChange объекта-выключателя:

```
def showSwitchState():
  print( switch.state )
...
switch.onChange = showSwitchState
```

Наш тестовый обработчик выводит на экран информацию о каждом изменении состояния выключателя.

Составной компонент

Построим ещё один новый компонент — **TEditPanel**, который объединяет два компонента — метку **TLabel** и поле ввода **TEdit**. Окно для входа пользователя в закрытую часть сайта или базы данных, использующее два таких составных компонента, показано на рис. 1.22.

Рис. 1.22

Метку и поле ввода удобно объединить на панели **TPanel**. Поэтому новый компонент будет наследником класса **TPanel**, а метка и поле ввода будут его внутренними полями:

```
class TEditPanel( TPanel ):
  def __init__( self, parent, label, **kw ):
    TPanel.__init__( self, parent, **kw )
    self.label = TLabel( self, text = label )
    self.label.align = "top"
    self.edit = TEdit( self )
    self.edit.align = "top"
```

У конструктора класса **TEditPanel** два обязательных параметра — ссылка на родительский объект и текст метки (параметр label). Сначала вызывается конструктор базового класса **TPanel**, затем созда-

ются метка и поле ввода, ссылки на них записываются во внутренние поля label и edit. Эти поля нет смысла делать скрытыми, потому что тогда затруднится доступ к данным элементам.

И для метки, и для поля ввода устанавливается режим выравнивания top, т. е. элементы «прижимаются» к верхней границе панели. Сначала размещается метка (она выравнивается первой), а потом — поле ввода. Отметим, что при таком выравнивании ширина всех компонентов будет изменяться автоматически при изменении ширины окна.

Для удобства добавим к этому компоненту свойство text, которое будет связано с одноимённым свойством поля ввода:

```
class TEditPanel(TPanel):
  ...
  def __getText( self ):
    return self.edit.text
  def __setText( self, newText ):
    self.edit.text = newText
  text = property( __getText, __setText )
```

Таким образом, все запросы к свойству text наш компонент без изменений передаёт (*делегирует*) своему внутреннему элементу edit.

Программа, которая проверяет имя пользователя и пароль, может выглядеть так:

```
def tryToEnter( sender ):
  if userEdit.text == "sysdba" and \
     passEdit.text == "masterkey":
     print("Доступ разрешён!")
     app.destroy()

app = TApplication("Вход в систему")
app.position = ( 200, 200 )
app.size = ( 200, 120 )

userEdit = TEditPanel( app, "Пользователь" )
userEdit.align = "top"

passEdit = TEditPanel( app, "Пароль" )
passEdit.align = "top"

btn = TButton(app, text="   Войти   ")
btn.position = ( 70, 85 )
btn.onClick = tryToEnter

app.run()
```

В главное окно программы добавляются два новых компонента **TEditPanel** и одна кнопка с надписью «Войти». Для кнопки установлен обработчик события `onClick`. При щелчке мышью по кнопке вызывается процедура `tryToEnter`, которая проверяет данные пользователя. В случае удачной проверки выводится сообщение «Доступ разрешён!» и программа заканчивает работу.

Выводы

- Новые компоненты обычно строятся на основе существующих с использованием наследования.
- Цель создания новых компонентов — добавить или изменить свойства и методы существующих компонентов или связать несколько компонентов в один составной компонент.
- Составные компоненты могут объединяться в новый компонент с помощью панели **TPanel**.
- Составные компоненты могут передавать запросы своим частям, такой приём называется делегированием.

Вопросы и задания

1. В каких случаях имеет смысл разрабатывать свои компоненты? Сравните такой подход с использованием готовых компонентов.
2. Почему программисты почти никогда не создают свои компоненты «с нуля»?
3. Объясните, как связаны рассмотренные в параграфе классы компонентов **TIntEdit** и **TEdit** с точки зрения объектно-ориентированного программирования. Чем они отличаются друг от друга?
4. Какие функции используются в языке Python для преобразования числового значения в текстовое и обратно?
5. Как получить запись числа в шестнадцатеричной системе счисления?
6. Объясните, как работает свойство `value` у компонента **TIntEdit** в программе из параграфа.
7. Почему мы не обрабатывали возможные ошибки в обработчике `onChange` класса **TIntEdit** в программе из параграфа?
8. Как вы думаете, как можно установить обработчик события во время выполнения программы?
9. Почему можно использовать в программе из параграфа обработчик события `onChange`, который не был объявлен в классе **TIntEdit**?
10. *Проект*. Разработайте компонент, который позволяет вводить шестнадцатеричные числа.
11. *Проект*. Разработайте компонент, который позволяет вводить вещественные числа (они относятся к типу **float**).

12. Изучив программу проверки пароля из параграфа, определите имя пользователя и пароль, при котором проверка проходит успешно. Узнайте в дополнительных источниках, где используются эти данные.
13. *Проект.* Доработайте программу для проверки пароля пользователя так, чтобы после трёх неудачных попыток она завершала свою работу с сообщением «Доступ запрещён!».

§ 11
Модель и представление

Ключевые слова:

- модель
- представление

Зачем это нужно?

Один из важнейших приёмов разработки современных программ — повторное использование написанного ранее кода. Чтобы облегчить решение этой задачи, было предложено использовать ещё одну декомпозицию: разделить *модель*, т. е. данные и алгоритмы их обработки, и *представление* — способ взаимодействия модели с пользователем.

Пусть, например, данные об изменении курсов валют хранятся в виде массива. По этим данным требуется строить зависимости, позволяющие прогнозировать изменение курсов 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. (рис. 1.23). Это *уровень представления*, или интерфейса с пользователем.

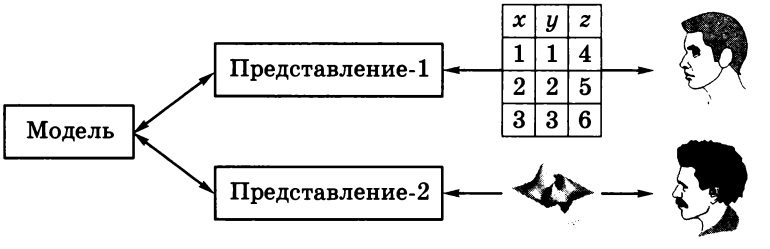


Рис. 1.23

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. Будем считать, что в выражении используются только:

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

Для простоты предполагаем, 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 арифметического выражения.

При вычислении арифметического выражения последней выполняется крайняя справа операция с наименьшим приоритетом. Используя этот факт, можно сформулировать следующий алгоритм вычисления арифметического выражения, записанного в символьной строке `expr` (от англ. *expression* — выражение):

1) найти последнюю операцию с наименьшим приоритетом; предположим, что индекс этого символа записан в переменной `pos`;
2) используя дважды этот же алгоритм, вычислить выражения слева и справа от символа с индексом `pos` и записать результаты вычисления в переменные `n1` и `n2` (рис. 1.24);

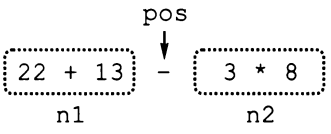


Рис. 1.24

3) выполнить операцию, символ которой записан в `expr[pos]`, со значениями переменных `n1` и `n2`.

Обратите внимание, что в пункте 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 — значение выражения, записанного в этой строке. Вот алгоритм её работы на псевдокоде:

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

Последнюю выполняемую операцию будем искать с помощью функции lastOp, которую напишем немного позже. Если эта функция вернула –1, это значит, что операция не найдена, т. е. вся переданная ей строка — это число.

Теперь можно написать функцию calc на языке Python:

```
def calc( expr ):
  pos = lastOp( expr )
  if pos < 0:          # вся строка - число
    return int( expr )
  else:
    n1 = calc( expr[:pos] )    # левая часть
    n2 = calc( expr[pos+1:] ) # правая часть
    # выполнить операцию
    return doOperation( expr[pos], n1, n2 )
```

Обратите внимание, что функция calc — рекурсивная, она дважды вызывает сама себя.

Функция doOperation выполняет допустимые операции:

```
def doOperation( op, n1, n2 ):
  if   op == "+": return n1 + n2
  elif op == "-": return n1 - n2
  elif op == "*": return n1 * n2
  else:           return n1 // n2
```

Осталось написать функцию lastOp, которая должна найти в символьной строке последнюю операцию с минимальным приоритетом. Сначала составим вспомогательную функцию, возвращающую приоритет символа-операции:

```
def priority( op ):
  if op in "+-": return 1
  if op in "*/": return 2
  return 100
```

Сложение и вычитание имеют приоритет 1, умножение и деление — более высокий приоритет 2, а все остальные символы (не операции) — приоритет 100 (условное значение).

Функция lastOp может выглядеть так:

```
def lastOp( expr ):
  minPrt = 50   # любое число между 2 и 100
  pos = -1
  for i in range( len(expr) ):
    prt = priority( expr[i] )
    if prt <= minPrt:
      minPrt = prt
      pos = i
  return pos
```

В условном операторе используется нестрогое неравенство (<=), чтобы найти именно *последнюю* операцию с наименьшим приоритетом.

Начальное значение переменной `minPrt` можно выбрать любым между наибольшим приоритетом операций (2) и условным кодом неоперации (100). Если найдена любая операция, в переменной `pos` сохраняется позиция найденного символа. Если же в строке нет операций, условие в условном операторе всегда ложно, и в переменной `k` остаётся начальное значение −1, которое и вернёт функция `lastOp`.

Цикл, в котором перебираются все элементы символьной строки, можно записать несколько иначе, «в стиле Python»:

```
for i, sym in enumerate(expr):
  prt = priority( sym )
  if prt <= minPrt:
    minPrt = prt
    pos = i
```

Функция `enumerate` перебирает все пары «индекс — символ». При каждом повторении цикла очередной индекс попадает в переменную `i`, а соответствующий ему символ `expr[i]` — в переменную `sym`.

Функции `calc`, `doOperation`, `priority` и `lastOp` удобно объединить в отдельный модуль `model.py` (модуль модели). Таким образом, модель в этой задаче — это функции, с помощью которых вычисляется арифметическое выражение, записанное в символьной строке.

Представление

Теперь построим интерфейс программы. В верхней части окна будет размещён выпадающий список (компонент **TComboBox**), в котором пользователь вводит выражение (рис. 1.25).

Рис. 1.25

При нажатии клавиши *Enter* выражение вычисляется, и его результат выводится в верхней строке обычного списка — компонент **TListBox**.

Выпадающий список, в котором хранятся все вводимые выражения, полезен для того, чтобы можно было вернуться к ранее введённому варианту и исправить его.

Итак, импортируем из модуля `model` функцию `calc` и создаём при-
ние:

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

На форме разместим компонент **TComboBox**. Чтобы прижать его к верху, установим свойство `align`, равное `"top"`. Назовём этот компонент[1] `inp` (от англ. *input* — ввод).

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

При вызове конструктора, кроме родительского объекта мы указали два именованных аргумента: `values` (в переводе с английского — значения) — пустой список, и `height` — высоту компонента, равную одной текстовой строке.

Добавляем второй компонент — **TListBox**, устанавливаем для него выравнивание `"client"` (заполнить всю свободную область) и имя `answers` (в переводе с английского — ответы).

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

Логика работы программы может быть записана в виде псевдокода:

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

Для перехвата нажатия клавиши *Enter* установим обработчик события `"<Key-Return>"` (по-английски — клавиша перевода каретки). Он подключается стандартным способом библиотеки `tkinter` — через метод `bind` (по-английски — связать):

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

Обработчик `doCalc` должен принимать один параметр — структуру, которая содержит информацию о произошедшем событии:

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

Вместо многоточия мы должны записать команды, которые нужно выполнить при нажатии клавиши *Enter*. Во-первых, читаем текст, введённый в выпадающем списке, и вычисляем выражение с помощью функции `calc`:

```
expr = inp.text
x = calc( expr )
```

[1] Конечно, лучше было назвать его `input`, но это имя уже занято — так называется встроенная функция ввода в Python.

Затем добавляем в начало списка вычисленное выражение и его результат, разделив их знаком равенства:

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

Здесь используется метод `insert` (по-английски — вставить), его первый аргумент — это номер вставляемого элемента: если он равен 0, строка вставляется в начало списка.

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

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

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

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

```
def doCalc( event ):
  expr = inp.text
  x = calc( expr )
  answers.insert( 0, expr + "=" + str(x) )
  if not inp.findItem( expr ):
    inp.addItem( expr )
```

Итак, в этой программе мы разделили модель (данные и средства их обработки) и представление (взаимодействие модели с пользователем). Построенный модуль модели можно использовать в любых программах, где нужно вычислять арифметические выражения. При этом представление может быть совершенно различным, модель от него никак не зависит.

Часто к паре «модель — представление» добавляют ещё управляющий блок (контро́ллер, от англ. *controller* — регулятор, управляющее устройство), который, например, обрабатывает ошибки ввода данных. Но часто контроллер и представление объединяются вместе — управление данными происходит в обработчиках событий.

Выводы

- В современных программах принято разделять модель (данные и алгоритмы их обработки) и представление (способ ввода исходных данных и вывода результатов).
- Одну и ту же модель можно использовать с разными представлениями (в разных проектах).
- Разделение на модель и представление позволяет разрабатывать и тестировать эти части отдельно друг от друга.

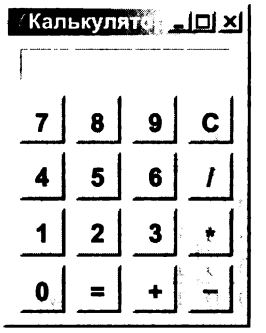
Вопросы и задания

1. Что даёт разделение программы на модель и представление? 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. Пусть требуется изменить программу так, чтобы она обрабатывала выражения со скобками. Что нужно изменить: модель, интерфейс или и то, и другое?
6. *Проект.* Измените программу из параграфа так, чтобы модель осталась та же, а представление было другое.
7. *Проект.* Измените программу из параграфа так, чтобы она вычисляла значения выражений с вещественными числами (для перевода вещественных чисел из символьного вида в числовой используйте функцию `float`).

*8. *Проект.* Добавьте в программу из параграфа обработку ошибок. Подумайте, какие ошибки может сделать пользователь. Какие ошибки могут возникнуть при вычислениях? Как их обработать?

*9. *Проект.* Измените программу из параграфа так, чтобы она вычисляла значения выражений со скобками. Подсказка: нужно искать последнюю операцию с самым низким приоритетом, стоящую вне скобок.

*10. *Проект.* Постройте программу «Калькулятор» для выполнения вычислений с целыми числами:



11. *Проект.* Напишите программу для игры «Змейка» (ru.wikipedia.org/wiki/Snake_(игра)**). Выделите модель задачи и представление.

Глава 2
ПРОГРАММИРОВАНИЕ НА ЯЗЫКАХ C++ И C#

§ 12
Классы и объекты в C++

Ключевые слова:

- класс
- объект
- экземпляр
- модификатор доступа
- конструктор
- инициализация
- метод
- рефакторинг

Новая задача и её анализ

Представьте себе, что в вашей игре нужно моделировать движение машины по дороге. Для упрощения будем считать, что дорога состоит из квадратных участков трёх типов: обочин, препятствий и свободных клеток, где может проехать машина. Так можно построить не только модель одной дороги, но и целую карту местности. Поэтому назовём этот объект *Картой*.

Сама машина занимает один квадрат и при нажатии на клавиши-стрелки может перемещаться по свободной части карты. Наша ближайшая задача — построить классы *Карта* и *Машина* на языке C++ и научиться их использовать.

Так как мы решили составить карту из квадратов, объект класса *Карта* должен где-то хранить информацию о том, что находится в каждом квадрате. Удобнее всего использовать для этого матрицу, размеры которой совпадают с размерами карты. Каждая ячейка этой матрицы (рис. 2.1) будет хранить одно из трёх значений: «свободно» (код 0), «обочина» (код 1) или «препятствие» (код 2).

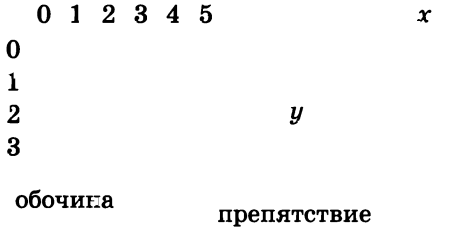


Рис. 2.1

Карта должна уметь перерисовывать себя, поэтому нужно добавить в этот класс метод `show` (в переводе с английского — показать).

Класс *Машина* должен хранить во внутренних полях координаты машины, *x* и *y*, — номера столбца и строки той клетки карты, где сейчас находится машина.

Машина, как и карта, должна уметь перерисовывать себя, поэтому в класс *Машина* тоже нужно добавить метод `show`. Кроме того, машина должна перемещаться при нажатии на клавиши-стрелки, поэтому нужен ещё метод `move` (по-английски — двигаться).

Машина должна как-то получать информацию о том, свободна ли клетка, в которую она перемещается. Поэтому объекты класса *Машина* должны хранить ссылку на карту, по которой машина едет. А в класс *Карта* нужно включить ещё один метод — логическую функцию `isFree`, которая определяет, свободна ли клетка с заданными координатами.

Таким образом, мы описали два класса, которые используются в игре (рис. 2.2).

а)

Карта
`width` (ширина) `height` (высота) `cells` (клетки)
show (показать) **isFree** (свободно?)

б)

Машина

`x`
`y`
`map` (карта)

show (показать) **move** (двигаться)

Рис. 2.2

Свойства карты `width` и `height` — это её ширина и высота, а `cells` — матрица, в которой хранится информация о содержании клеток карты.

Класс CMap

Названия всех классов будем начинать с прописной буквы «С» (от англ. *class* — класс). Класс *Карта* назовём в программе **CMap** (от англ. *map* — карта).

Выделим место под матрицу с запасом, считая, что в нашей игре не встретятся карты размером больше, чем 50 на 50 клеток (вы можете использовать свои максимальные значения). Объявляем две целочисленные константы — максимальные размеры карты, `MAX_WIDTH` (максимальная ширина), и `MAX_HEIGHT` (максимальная высота):

```
const int MAX_WIDTH = 50,
          MAX_HEIGHT = 50;
```

Простейший вариант класса *Карта* выглядит так:

```
class CMap
  {
  int width, height;
  int cells[MAX_WIDTH][MAX_HEIGHT];
  };
```

Обратите внимание, что после закрывающей скобки, которая завершает объявление класса, должна стоять точка с запятой (она выделена фоном).

В отличие от языка Python, в C++ все поля и методы класса нужно заранее объявлять, добавить новое поле или метод во время работы программы не получится. В классе **CMap** мы объявляем два целочисленных поля `width` и `height` (размеры карты — ширину и высоту), а также массив `cells`, в котором будет храниться информация о клетках карты.

Теперь можно построить новый объект — экземпляр класса **CMap**:

```
CMap map;
```

Однако попытка обратиться к полям этого объекта с помощью точечной записи приведёт к ошибке — программа даже не пройдёт компиляцию:

```
map.width = 15; // это ошибка!
```

Дело в том, что по умолчанию все поля и методы объекта в C++ закрытые, или частные (англ. *private*). Наше описание класса равносильно такому:

```
class CMap
  {
  private:
    int width, height;
    int cells[MAX_WIDTH][MAX_HEIGHT];
  };
```

Служебное слово **private**, за которым стоит двоеточие, называется *модификатором доступа*. Эта инструкция определяет правила доступа к записанным далее полям и методам класса.

Инструкция **private** начинает секцию, где перечисляются закрытые (частные) поля и методы класса. А данные, доступные другим объектам, должны быть объявлены в секции, которая начинается модификатором **public** (в переводе с английского — общий, публичный):

```
class CMap
  {
  public:
    int width, height;
    int cells[MAX_WIDTH][MAX_HEIGHT];
  };
```

Теперь можно создать объект и присвоить значения всем полям:

```
CMap map;        // (1)
map.width = 20;
map.height = 25;
```

Объект создаётся в памяти в строке 1, при этом вызывается специальный метод класса — *конструктор*. Если в описании класса конструктор явно не определён (как у нас), компилятор построит конструктор по умолчанию, который просто выделяет в памяти место для объекта. Все поля будут содержать «мусор» — неопределённые значения, использовать которые бессмысленно и опасно.

Пишем свой конструктор

Чтобы случайно не использовать «мусорные» данные, желательно при создании объекта сразу присваивать всем его полям начальные значения. Для этого определим свой конструктор:

```
class CMap
  {
  public:
    int width, height;
    int cells[MAX_WIDTH][MAX_HEIGHT];
    CMap()
       {
       width = 0;
       height = 0;
       }
  };
```

Конструктор (он выделен фоном) — это специальный метод, который создаёт объект в памяти и выполняет его *инициализацию*, т. е. присваивает полям начальные значения. Имя конструктора всегда совпадает с именем класса. Он не возвращает никакого значения, даже **void**.

Можно записать тот же конструктор в краткой форме:

```
class CMap
   {
   public:
     ...
     CMap(): width(0), height(0) {}
   };
```

В заголовке такого конструктора для каждого поля в скобках указано его начальное значение, а тело конструктора — пустое.

В этом конструкторе мы не записали начальные значения в массив `cells` — там пока находится «мусор». Когда размеры карты выбраны, нужно выполнить её *инициализацию* — записать в крайние левый и правый столбцы код 1 («обочина»), а в остальные — код 0 («свободно»). Это будет делать новый метод `init` (от англ. *initialization* — инициализация), который мы добавим в класс:

```
class CMap
  {
  public:
    ...
    void init();    // прототип метода
  };

void CMap::init()  // реализация метода
  {
  for( int x = 0; x < width; x++ )
    for( int y = 0; y < height; y++ )
      if( x == 0  or  x == width-1 )
           cells[x][y] = 1;
      else cells[x][y] = 0;
  }
```

В описании класса метод `init` только объявлен: мы записали его *прототип* — заголовок метода с точкой с запятой в конце. А реализация метода записана отдельно, после объявления класса. «Приставка» `CMap::` в заголовке говорит о том, что это метод класса **CMap**, а не обычная процедура. Другими словами, процедура `init` принадлежит *пространству имён*[1] класса **CMap**. Только поэтому внутри метода можно обращаться к полям `width`, `height` и `cells` класса `CMap`.

В начале основной программы нужно создать объект, выбрать размеры карты и выполнить её инициализацию:

```
CMap map; // вызов конструктора
map.width = 20;
map.height = 20;
map.init();
```

Лучше всего объединить все эти действия: при создании объекта сразу присваивать полям нужные значения и заполнять матрицу `cells`. Для этого можно добавить в класс ещё один конструктор с двумя параметрами:

```
class CMap
  {
  public:
    ...
    CMap(): width(0), height(0) {}
    CMap( int width0, int height0 );
  };
...
CMap::CMap( int width0, int height0 )
```

[1] Таким же способом можно указывать и другие пространства имён. Например, обращение к выходному потоку `cout` из пространства имён `std` можно записать как `std::cout`.

```
{
width = max( 0, min( width0, MAX_WIDTH ) );
height = max( 0, min( height0, MAX_HEIGHT ) );
init();
}
```

В описании класса мы оставили только объявление нового конструктора, а его реализация записана отдельно. Так обычно делают, если метод содержит более одной-двух строк. При этом описание класса становится короче и легче для понимания.

Теперь у класса **CMap** два конструктора. Их может быть и больше, но все они должны различаться по количеству и типам параметров (по *сигнатуре*, от англ. *signature* — подпись). Иначе компилятор не сможет различить, когда какой конструктор вызывается.

Обратите внимание, что в конструкторе с параметрами выполняется проверка правильности входных данных: мы не разрешаем устанавливать ширину карты меньше нуля или больше, чем MAX_WIDTH, а высоту — меньше нуля или больше, чем MAX_HEIGHT. После того как размеры будут установлены, сразу вызывается метод init, и карта готова к работе.

По команде

```
CMap map1;
```

вызывается первый конструктор, поля объекта обнуляются. Второй конструктор срабатывает, если после имени нового объекта в скобках указать размеры карты:

```
CMap map2( 20, 25 );
```

Заметим, что приведённые выше приёмы работают во всех версиях C++, включая стандарт C++98. А в стандарте C++11 (и во всех более новых) разрешено задавать начальные значения по умолчанию в фигурных скобках после имени каждого поля:

```
class CMap
  {
  public:
    int width{20}, height{25};
    int cells[MAX_WIDTH][MAX_HEIGHT]{};
  };
```

Пустые фигурные скобки в последней строке означают, что все ячейки матрицы cells заполняются нулями.

Рефакторинг

Попробуем улучшить описание класса так, чтобы сделать его более простым и понятным (выполним *рефакторинг*).

Во-первых, сделаем так, чтобы в массив cells нельзя было записывать другие значения, кроме 0, 1 и 2. Для этого введём новый тип данных:

```
enum cellType { EMPTY, SIDE, WALL };
```

Такой тип данных называется *перечисляемым*, он объявляется с помощью служебного слова **enum** (от англ. *enumeration* — перечисление).

Переменные типа **cellType** могут принимать только значения EMPTY (в переводе с английского — пустой), SIDE (боковая сторона) и WALL (стена). Фактически это целые числа, если вывести их на экран с помощью оператора

```
cout << EMPTY << SIDE << WALL;
```

вы увидите, что EMPTY = 0, SIDE = 1 и WALL = 2. Но в переменную типа **cellType** компилятор не даст записать значение, не входящее в перечисление:

```
cellType a = 25;    // ошибка!
cellType b = 2;     // ошибка!
cellType c = WALL; // так можно
```

Ограничивая множество допустимых значений, мы даём компилятору возможность автоматически обнаруживать возможные ошибки. Применяя эту идею, выберем тип элементов **cellType** для матрицы cells в классе **CMap**:

```
enum cellType { EMPTY, SIDE, WALL };
class CMap
  {
  public:
    ...
    cellType cells[MAX_WIDTH][MAX_HEIGHT];
  };
```

В методе init нужно избавиться от «магических чисел» 1 и 0, заменив их на SIDE и EMPTY:

```
void CMap::init()
  {
  for( int x = 0; x < width; x++ )
    for( int y = 0; y < height; y++ )
      if( x == 0  or  x == width-1 )
           cells[x][y] = SIDE;
      else cells[x][y] = EMPTY;
  }
```

Добавим в класс новый метод — логическую функцию isFree, которая возвращает значение true, если свободна клетка с переданными ей координатами:

```
class CMap
  {
  public:
    ...
    bool isFree( int x, int y ) const {
      return cells[x][y] == EMPTY;
      }
  };
```

Метод isFree скоро нам понадобится: перед тем, передвинуть машину, нужно проверить, свободна ли клетка, в которой она должна оказаться.

Так как функция isFree получилась короткой, мы разместили её прямо в описании класса. Такой метод компилятор не оформляет как отдельную функцию, а встраивает (англ. *inline*) прямо в машинный код каждый раз, когда этот метод вызывается.

В заголовке метода появилось новое слово **const**. Оно говорит о том, что это *константный метод*, т. е. метод, не изменяющий объект. Служебное слово **const** писать не обязательно, но оно помогает понять, как устроен и как работает класс. Если внутри этого метода случайно изменить данные объекта, компилятор сообщит об ошибке.

Сделаем ещё одно улучшение. Два конструктора класса **CMap** можно объединить в один. Чтобы конструктор с параметрами можно было вызывать без аргументов, для всех параметров нужно задать *значения по умолчанию*:

```
class CMap
  {
  public:
    int width, height;
    cellType cells[MAX_WIDTH][MAX_HEIGHT];
    CMap( int width0 = 10 , int height0 = 15 );
    void init();
  };
```

Теперь, если при создании карты ширина и высота не заданы, ширина будет равна 10, а высота — 15.

Новый конструктор можно вызвать тремя способами:

```
CMap map1;              // 10, 15

CMap map2( 20 );        // 20, 15

CMap map3( 20, 20 );    // 20, 20
```

В первом случае конструктор вызван без аргументов, поэтому применяются оба значения по умолчанию: ширина 10 и высота 15. При вызове с одним аргументом 20 мы задаём только ширину (первый параметр),

а второй параметр (высота) принимает значение по умолчанию — 15. В последнем случае явно заданы оба размера карты.

Нужно помнить, что если какой-то параметр конструктора или другой функции имеет значение по умолчанию, то все следующие параметры тоже должны иметь значения по умолчанию. Например, такое объявление конструктора неверно:

```
class CMap
  {
  public:
    ...                          // это ошибка!
    CMap( int width0 = 10, int height0 );
  };
```

Рисуем карту

Карта состоит из клеток, размер этих клеток на экране лучше задать как константу, чтобы при необходимости его было легко изменить:

```
const int CELL_SIZE = 20;
```

Добавим в класс **CMap** метод show, который рисует карту:

```
class CMap
  {
  public:
    ...
    void show() const;
  };
```

Этот метод тоже не изменяет объект, поэтому объявлен с описателем **const**.

Для рисования будем использовать библиотеку TX Library[1)], разработанную И. Р. Дединским. Основная программа получается очень короткой:

```
const int W = 15, L = 20;                          // (1)
CMap map( W, L );                                  // (2)
txCreateWindow( W*CELL_SIZE, L*CELL_SIZE );        // (3)
map.show();                                        // (4)
```

Размеры карты объявлены как константы W и L внутри основной программы (строка 1). В строке 2 строится новый объект-карта с этими размерами. Окно для вывода графики создаётся в строке 3, а затем мы рисуем карту на экране (строка 4).

Остаётся написать реализацию (код) метода show. Для вывода карты надо перебрать все ячейки массива cells, для каждой клетки выбрать цвет в зависимости от её содержимого и нарисовать на холсте квадрат нужного цвета.

[1)] Библиотеку TX Library можно загрузить с главной страницы сайта И. Р. Дединского **ded32.ru** или **txlib.ru**.

```
void CMap::show() const
  {
  for( int x = 0; x < width; x++ )
    for( int y = 0; y < height; y++ ) {
      COLORREF color;
      if( cells[x][y] == EMPTY )          // (1)
           color = RGB(50, 255, 50);
      else if( cells[x][y] == SIDE )
           color = RGB(80, 0, 0);
      else /* if( cells[x][y] == WALL ) */
           color = RGB(0, 80, 0);
      txSetColor( color );                // (2)
      txSetFillColor( color );            // (3)
      int left = x*CELL_SIZE,             // (4)
          top = y*CELL_SIZE;
      txRectangle( left, top,             // (5)
          left+CELL_SIZE, top+CELL_SIZE );
      }
  }
```

Цепочка условных операторов, которая начинается со строки 1, служит для выбора цвета. Для пустых ячеек мы используем светло-зелёный цвет, для обочин — коричневый (тёмно-красный), для препятствий — тёмно-зелёный. Цвета задаются в модели RGB как яркости трёх составляющих (красной, зелёной и синей), каждая из которых кодируется целым числом от 0 до 255.

В строках 2 и 3 устанавливается одинаковый цвет контура и заливки. Координаты левого верхнего угла квадрата (в пикселях) вычисляются в строке 4, а в следующей строке команда `txRectangle` рисует этот квадрат.

Попробуем сразу выполнить рефакторинг. В методе `show` есть несколько строк (между строками 1 и 2), в которых выбирается цвет клетки. Эти операции можно объединить в новый метод `cellColor` класса **CMap**:

```
class CMap
  {
  public:
    ...
    COLORREF cellColor( int x, int y ) const;
  };

COLORREF CMap::cellColor( int x, int y ) const
  {
  COLORREF color;
  if( cells[x][y] == EMPTY )
    color = RGB(50, 255, 50);
```

```
  else if( cells[x][y] == SIDE )
    color = RGB(80, 0, 0);
  else /* if( cells[x][y] == WALL ) */
    color = RGB(0, 80, 0);
  return color;
  }
```

Метод show сокращается и становится более понятным:

```
void CMap::show() const
  {
  for( int x = 0; x < width; x++ )
    for( int y = 0; y < height; y++ ) {
      COLORREF color = cellColor( x, y );
      txSetColor( color );
      txSetFillColor( color );
      int left = x*CELL_SIZE,
          top = y*CELL_SIZE;
      txRectangle( left, top,
          left+CELL_SIZE, top+CELL_SIZE );
      }
  }
```

Если теперь собрать полную программу и запустить её, она должна нарисовать карту и завершиться при нажатии на любую клавишу.

Выводы

- Описание класса в C++ начинается служебным словом class.
- Для ограничения доступа к полям и методам класса используют модификаторы доступа. Модификатор private обозначает закрытые (частные) данные и методы, обращаться к которым могут только объекты данного класса. Модификатор public обозначает общедоступные данные и методы.
- По умолчанию все поля и методы класса — закрытые.
- Класс может иметь несколько конструкторов. Все они должны различаться по сигнатуре, т. е. по количеству и типам параметров.
- Реализация метода класса может быть вынесена из объявления класса. В этом случае в объявлении класса нужно записать объявление (прототип) метода.
- Данные перечисляемого типа (enum) могут принимать одно из заданного множества значений. Эти значения кодируются целыми числами.

Вопросы и задания

1. Как можно назвать этап решения задачи, который обсуждается в первом пункте параграфа?
2. Найдите ошибку в объявлении класса.

```
class CMarusya { int strength; }
```

3. Сравните работу с полями объектов в языках Python и C++. Какой подход вам больше нравится?

4. Найдите ошибки в программе.

```
class CBobr { int weight; }
int main()
  {
  CBobr b(5);
  b.height = 11;
  }
```

5. Найдите ошибки в программе.

```
class CShip {
  int x, y;
  CShip() { x = y = 0; }
  };
int main() { CShip ship; }
```

6. Найдите ошибку в программе (используйте описание класса **CMap** из параграфа):

```
CMap map;
cout << map->width;
```

7. Может ли класс иметь два таких конструктора?

```
class CBall {
  int R;
  public:
    CBall() { R = 10; }
    CBall( int _R = 0 ) { R = _R; }
  };
```

Проверьте ваш ответ с помощью компьютера.

8. Что такое сигнатура метода?

9. В каких случаях два конструктора класса можно заменить одним?

10. Объясните значение термина «инициализация». Почему лучше выполнять инициализацию полей в конструкторе?

11. Почему метод `init` в классе **CMap** не объявлен с описателем **const**?

12. Как вы думаете, можно ли объявить конструктор с описателем **const**? Обоснуйте ваш ответ, проверьте его экспериментально.

13. Запишите конструктор так, чтобы его тело было пустым.

```
class CPoint {
  int x, y;
  public:
    CPoint( int x0, int y0 )
      { x = x0; y = y0; }
  };
```

14. Зачем при рефакторинге стараются избавиться от «магических чисел»?

15. Почему код (реализация) сложных методов класса обычно записывается отдельно от объявления класса?
16. Найдите ошибку в объявлении класса.

```
class CBall {
  int R, mass;
  public:
    CBall( int _R = 0, int _M ) {
      R = _R; mass = _M;
      }
  };
```

17. Что подозрительное вы заметили в объявлении класса?

```
class CCar {
  public:
    CMap map;
    int x, y;
  };
```

Интересные сайты

cpp.sh — онлайн-среда для программирования на C++

ded32.ru (txlib.ru) — сайт профессиональной проектной работы по информатике для школьников, там же расположена библиотека TX Library

§ 13
Программа с классами (практикум)

Ключевые слова:

- класс
- объект
- ссылка
- конструктор
- метод
- рефакторинг

В этом параграфе мы построим класс *Машина* и закончим простую программу, в которой машина двигается по карте. После этого вы сможете совершенствовать её самостоятельно.

Класс CCar

Данные класса *Машина* (**CCar**) — это координаты машины, x и y. Кроме того, машина должна быть связана с картой, иначе непонятно, что означают эти координаты и как проверить, свободна ли клетка, в которую перемещается машина. Чтобы организовать такую связь, нужно хранить для каждого объекта-машины ссылку на карту, которую машина использует. Таким образом, одно из полей в классе **CCar** — это ссылка на объект класса **CMap**:

```
CMap& map;
```

Ссылка (англ. *reference*) указывает на какой-то объект и позволяет обращаться к его свойствам и методам. Такое отношение между классами называется «использование» (в отличие от наследования).

Ссылка во многом похожа на указатель — переменную особого типа, которая хранит адрес объекта. Но, в отличие от указателя, ссылка не может принимать «нулевое» значение, она всегда указывает на существующий объект. Кроме того, после присваивания начального значения ссылку нельзя изменить. Это значит, что будет невозможно «переставить» машину на другую карту[1].

Вспомним (см. рис. 2.2), что у машины должно быть два метода — show и move. Имя move недостаточно понятное: сложно догадаться, куда и насколько двигается машина. Поэтому заменим его на moveBy (в переводе с английского — передвинуться на) и будем передавать этому методу изменения координат машины.

Введём класс **CCar** следующим образом:

```
class CCar
  {
  public:
    CMap& map;                                          // (1)
    int x, y;                                           // (2)
    CCar( CMap& map0, int x0 = 0, int y0 = 0  ):        // (3)
        map( map0 ), x( x0 ), y( y0 )
        {};
    void show() const;                                  // (4)
    bool moveBy( int dx = 0, int dy = 0 );              // (5)
  };
```

Строки 1 и 2 объявляют данные объекта: ссылку на карту map и целочисленные поля x и y — координаты машины на этой карте.

В строке 3 начинается описание конструктора. Он принимает три параметра — ссылку на карту и начальные координаты. Первый параметр конструктора обязательный, потому что машину обязательно надо «привязать» к какой-то карте. Для остальных заданы нулевые значения по умолчанию. Конструктор присваивает всем полям начальные значения, его тело пустое.

В строках 4 и 5 объявлены два метода — константный метод show для вывода машины на экран и метод moveBy для перемещения машины. Метод moveBy принимает два параметра — смещения по каждой из координат. Координату x нужно увеличить на величину dx, а координату y — на величину dy. По умолчанию оба смещения равны нулю.

[1] Если необходимо связывать машину с разными картами, вместо ссылки нужно использовать указатель на карту.

В основной программе сначала построим объект-карту:

```
const int W = 15, L = 20;
CMap map( W, L );
```

Затем при создании объекта-машины передадим конструктору ссылку на эту карту:

```
CCar car( map, 10, 10 );
```

Будем изображать машину синим ромбом (конечно, вы можете использовать и более сложный рисунок). Метод show можно записать так:

```
void CCar::show() const
  {
  txSetColor( RGB(0,0,255) );
  txSetFillColor( RGB(0,0,255) );
  int xCenter = x*CELL_SIZE + CELL_SIZE / 2,
      yCenter = y*CELL_SIZE + CELL_SIZE / 2;
  POINT carImage[] = { {xCenter-5, yCenter},
                       {xCenter, yCenter-10},
                       {xCenter+5, yCenter},
                       {xCenter, yCenter+10} };
  txPolygon( carImage, 4 );
  }
```

Переменные xCenter и yCenter обозначают координаты центра клетки карты (в пикселях). Массив carImage состоит из структур типа **POINT**, которые описывают точки на плоскости. Этот массив задаёт координаты четырёх углов ромба. Рисование ромба выполняет функция txPolygon, которая принимает два аргумента: массив координат точек многоугольника и количество этих точек.

Теперь напишем код метода moveBy. Прежде чем передвинуть машину в новую позицию, с помощью метода isFree привязанной карты узнаем, свободно ли это место:

```
bool CCar::moveBy( int dx, int dy )
  {
  if( map.isFree( x+dx, y+dy ) ) { // (1)
    x += dx;
    y += dy;
    return true;
    }
  else
    return false;
  }
```

Заметьте, что при определении метода в заголовке не нужно указывать значения параметров dx и dy по умолчанию, они задаются только в прототипе метода в описании класса.

Поскольку поле map — это *ссылка* на объект-карту (а не указатель), для вызова метода isFree применяется обычная точечная запись.

Как видно из условия в строке 1, машина перемещается только тогда, когда клетка карты с новыми координатами свободна. В этом случае метод возвращает логическое значение true («истина») — машина перемещена успешно. Если переместить машину не удаётся, метод возвращает результат false («ложь»).

Снова рефакторинг

Итак, классы вроде бы готовы. Но давайте всё-таки попробуем ещё кое-что улучшить.

Во-первых, функцию выбора цвета клетки карты можно «изъять» из класса CMap и выделить в отдельную подпрограмму. Она определяет цвет клетки не по координатам, а по коду объекта, находящегося в этой клетке:

```
COLORREF cellColor( cellType type )
  {
  COLORREF color;
  if( type == EMPTY )
    color = RGB(50, 255, 50);
  else if( type == SIDE )
    color = RGB(80, 0, 0);
  else // if( type == WALL )
    color = RGB(0, 80, 0);
  return color;
  }
```

Операцию рисования клетки карты оформим как отдельную процедуру, не входящую ни в какой класс. Она принимает три параметра — координаты клетки и код объекта:

```
void drawCell( int x, int y, cellType type )
  {
  COLORREF color = cellColor( type );
  txSetColor( color );
  txSetFillColor( color );
  int left = x*CELL_SIZE,
      top = y*CELL_SIZE;
  txRectangle( left, top,
               left+CELL_SIZE, top+CELL_SIZE );
  }
```

В первой строке для определения цвета клетки вызывается функция cellColor, которую мы только что написали.

Также выделим в отдельную процедуру рисование машины:

```
void drawCar( int x, int y )
  {
  txSetColor( RGB(0,0,255) );
  txSetFillColor( RGB(0,0,255) );
  int xCenter = x*CELL_SIZE + CELL_SIZE / 2,
      yCenter = y*CELL_SIZE + CELL_SIZE / 2;
  POINT carImage[] = { {xCenter-5, yCenter},
                       {xCenter, yCenter-10},
                       {xCenter+5, yCenter},
                       {xCenter, yCenter+10} };
  txPolygon( carImage, 4 );
  }
```

Теперь методы рисования объектов становятся совсем простыми и понятными:

```
void CMap::show() const
  {
  for( int x = 0; x < width; x++ )
    for( int y = 0; y < height; y++ )
      drawCell( x, y, cells[x][y] );
  }
void CCar::show() const
  {
  drawCar( x, y );
  }
```

В результате мы разделили модель и представление (см. § 11). Классы **CMap** и **CCar** представляют собой модель — данные и алгоритмы их обработки. Эти классы «ничего не знают» о том, как карта и машина будут изображены на экране. Меняя процедуры `drawCell` и `drawCar`, мы можем изображать данные как угодно (например, в виде шестиугольных сот), при этом не придётся менять модель.

Основная программа

В основной программе нужно создать объекты (карту и машину) и окно для вывода графики:

```
const int W = 15, L = 20;
CMap map( W, L );
CCar car( map, 10, 10 );
txCreateWindow( W*CELL_SIZE, L*CELL_SIZE );
```

Для анимации будем использовать цикл, работающий до нажатия клавиши *Escape*:

```
while( not GetAsyncKeyState(VK_ESCAPE) ) { // (1)
  txClear();                               // (2)
  if( GetAsyncKeyState(VK_LEFT) )          // (3)
    car.moveBy( -1 );
  if( GetAsyncKeyState(VK_RIGHT) )         // (4)
    car.moveBy( 1 );
  map.show();                              // (5)
  car.show();                              // (6)
  txSleep( 50 );                             // (7)
  }
```

В условии цикла в строке 1 вызывается функция GetAsyncKeyState. Она возвращает истинное значение, если в момент её вызова нажата клавиша *Escape*, код которой равен VK_ESCAPE. При этом обратное условие станет ложно, и цикл завершится.

В строке 2 очищается графическое окно, а в строках 3 и 4 мы проверяем, не нужно ли двигать машину. Если нажата клавиша «влево» (она имеет код VK_LEFT), вызывается метод moveBy. Первый аргумент при вызове этого метода равен –1, а второй не задан, т. е. считается равным нулю (см. прототип метода в классе). Поэтому машина перемещается влево на одну клетку. Аналогично при нажатии кнопки «вправо» (с кодом VK_RIGHT) машина переместится на одну клетку вправо.

В строках 5 и 6 рисуем сначала карту, а потом — машину. Строка 7 обеспечивает задержку выполнения программы на 50 миллисекунд.

Движение машины вверх и вниз при желании вы можете сделать самостоятельно. В некоторых играх машина не двигается по вертикали, вместо этого карта прокручивается сверху вниз.

Разбиение на модули

Программа получилась достаточно большой, поэтому возникает естественное желание разбить её на части, сохранив каждую часть в отдельном файле. Например, в отдельный модуль, который мы назовём cargame.cpp, можно выделить определение обоих классов[1], **CMap** и **CCar**.

Сразу возникает вопрос: как собрать исполняемую программу из нескольких исходных файлов? Для этого нужно создать *проект*, объединяющий несколько файлов. Это умеют все современные среды разработки программ. Кроме того, для сборки проекта из командной строки можно использовать утилиту make.

В C++ используется раздельная компиляция, т. е. все файлы проекта проходят компиляцию независимо друг от друга. Для каждого из

[1] Другой вариант — выделить каждый класс в отдельный файл. При желании вы сможете это сделать самостоятельно, поняв общий принцип.

них компилятор строит отдельный объектный файл (обычно он имеет расширение .o или .obj). На следующем этапе эти объектные файлы обрабатывает другая программа — редактор связей (англ. *linker*), задача которого — собрать из файлов один исполняемый файл, подключив все используемые библиотеки.

Из-за раздельной компиляции возникает ещё одна проблема: обрабатывая файл с основной программой, компилятор не будет знать, что где-то есть файл cargame.cpp с определениями классов **CMap** и **CCar**. Как же сказать ему, что такие классы есть?

Давайте вспомним, как используются функции стандартной библиотеки в языке C++. Чтобы компилятор знал про эти функции, мы включаем в начало файла специальные заголовочные файлы, содержащие объявления (прототипы) функций: iostream, random, cmath и др. Следовательно, нужно написать свой заголовочный файл, в котором будут описания наших классов. Назовём его cargame.hpp (заголовочным файлам на языке C++ обычно дают расширения .h или .hpp) и подключим его в основной программе:

```
#include "cargame.hpp"
```

Двойные кавычки вместо угловых скобок говорят о том, что нужно сначала искать файл в текущем каталоге проекта, а не в системных каталогах, где хранятся заголовочные файлы стандартной библиотеки.

Заголовочный файл cargame.hpp содержит только константы и новые типы данных (в том числе описания классов):

```
enum cellType {EMPTY, SIDE, WALL};

const int MAX_WIDTH = 50,
          MAX_HEIGHT = 50;

class CMap
  {
  public:
    int width, height;
    cellType cells[MAX_WIDTH][MAX_HEIGHT];
    CMap( int width0 = 10, int height0 = 15 );
    void init();
    void show() const;
    bool isFree( int x, int y ) const {
      return cells[x][y] == EMPTY;
      }
  };

class CCar
  {
  public:
    CMap& map;
```

```
    int x, y;
    CCar( CMap& _map, int x0 = 0, int y0 = 0  ):
        map(_map), x(x0), y(y0)
        {};
    void show() const;
    bool moveBy( int dx = 0, int dy = 0 );
  };
```

Реализацию всех методов классов поместим в файл cargame.cpp:

```
#include <algorithm>
#include "cargame.hpp"
using namespace std;
void drawCell( int x, int y, cellType type );
void drawCar( int x, int y );
... // далее – реализация всех методов классов
```

В начале файла подключается библиотека algorithm. В ней объявлены функции min и max, которые используются в конструкторе класса CMap. Затем подключаем заголовочный файл cargame.hpp и записываем прототипы процедур рисования.

Таким образом, мы построили модуль, содержащий классы **CMap** и **CCar**. Он состоит из двух частей — заголовочного файла с расширением .h или .hpp (интерфейса) и файла с расширением .cpp, в котором реализованы методы классов.

А что же осталось в файле с основной программой? Только то, что относится к представлению, т. е. к картинке на экране:

```
#include "TxLib.h"
#include "cargame.hpp"

const int CELL_SIZE = 20;

COLORREF cellColor( cellType type )
  {
  ...
  }
void drawCell( int x, int y, cellType type )
  {
  ...
  }
void drawCar( int x, int y )
  {
   ...
  }
int main()
{
   const int W = 15, L = 20;
   CMap map( W, L );
```

```
    CCar car( map, 10, 10 );

       txCreateWindow( W*CELL_SIZE, L*CELL_SIZE );

    while( not GetAsyncKeyState(VK_ESCAPE) ) {
      ...
      }
}
```

Операторы, которые нужно поставить вместо многоточий, можно посмотреть в середине этого параграфа.

Теперь вы знаете, как разбить объектно-ориентированную программу на модули. В качестве упражнения можно поместить каждый класс в отдельный модуль.

Выводы

- Связь с объектом можно установить с помощью специального поля типа «ссылка». В отличие от указателя, ссылка всегда указывает на существующий объект и не может быть изменена.
- Для обращения к полям и методам объекта по ссылке используется точечная запись.
- Разделение программы на модель данных и представление позволяет легко изменять одну из этих частей, не затрагивая вторую. Кроме того, модель можно использовать в других программах.
- Классы, которые создаёт программист, обычно оформляются в виде модулей. Модуль может содержать один класс или несколько связанных по смыслу классов.
- Модуль состоит из двух частей — заголовочного файла с расширением `.h` или `.hpp` (интерфейса) и файла с расширением `.cpp`, в котором реализованы методы классов.

Вопросы и задания

1. Что произойдёт, если поменять местами строки 5 и 6 в основной программе, приведённой на с. 101?
2. Почему при вызове метода `moveBy` в основной программе из параграфа задаётся только один аргумент, хотя метод имеет два параметра?
3. Что произойдёт, если вызвать метод `moveBy` без параметров (см. программу из параграфа): `car.moveBy();`?
4. Предположите, что произойдёт, если во время работы программы из параграфа одновременно нажать стрелки «влево» и вправо». Что изменится, если два условных оператора в цикле основной программы связать словом **else**?
5. Подумайте, в каком случае вызов метода `isFree` в программе из параграфа может привести к ошибке? Как можно защититься от этой ошибки?

6. Ваш коллега предлагает при разбиении на модули программы из параграфа объединить файлы `cargame.hpp` и `cargame.cpp` в один файл и включить в программу этот объединённый файл с помощью команды **`#include`**. Оцените достоинства и недостатки такого решения.
7. Двигая машину (см. программу из параграфа) с помощью стрелок во время игры, вы можете заметить мигание графического окна. Чтобы убрать мигание, попробуйте использовать функции `txBegin` и `txEnd` из библиотеки `TX Library`. Описание этих функций найдите в справочной системе.
8. Переделайте программу из параграфа, выделив каждый класс в отдельный модуль.
9. Переделайте программу из параграфа так, чтобы снять ограничение на максимальные размеры поля. *Указание*: вместо статического массива `cells` используйте вектор, составленный из векторов:

```
vector< vector<cellType> > cells;
```

*10. *Проект*. Доделайте игру, которая была начата в этом параграфе. Изображение карты должно «ехать» сверху вниз, на карте будут появляться препятствия, которые машина должна объезжать. Чем дольше продержится машина на трассе, тем больше очков получает игрок.

*11. *Проект*. Доработайте игру из предыдущего задания так, чтобы клавиши-стрелки «вверх» и «вниз» изменяли скорость движения машины. Начисляйте очки за скорость прохождения трассы заданной длины.

**12. *Проект*. Придумайте свою простую игру и напишите объектно-ориентированную программу, используя пример из параграфа.

Интересные сайты

ideone.com — онлайн-среда для программирования на различных языках

repl.it — онлайн-среда для программирования на различных языках

§ 14 Инкапсуляция

Ключевые слова:

- инкапсуляция
- поле
- метод
- модификаторы доступа
- интерфейс
- свойство

Объект защищает свои данные

В программе, которую мы написали выше, все поля и методы классов были открытыми — мы объявляли их в секции public. Согласно принципу инкапсуляции, всё внутреннее устройство объектов должны быть «невидимо» для остальных объектов, «заключено в капсулу». Объект тем и отличается от структуры, что сам работает со своими данными, никого к ним «не подпуская».

Для того чтобы разобраться с инкапсуляцией в C++, мы построим класс **CPen** — объект-перо. Для простоты пусть он содержит единственное поле FColor, которое хранит информацию о цвете в виде символьной строки (см. § 4). Запишем объявление класса:

```
class CPen
  {
  private:
    string FColor;
  };
```

Те элементы (поля и методы), которые нужно скрыть, в описании класса помещают в «частную» секцию, которая начинается с модификатора доступа **private**. Таким образом, теперь 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 класса **CPen**), поэтому сейчас невозможно не только изменить внутреннее поле объекта, но и просто узнать его значение.

Чтобы всё же получить доступ к полю FColor, добавим к классу ещё два метода. Один из них, getColor (на жаргоне программистов — «геттер», от англ. *get* — получать), будет возвращать текущее значение поля FColor. Второй метод, setColor («сеттер», от англ. *set* — устанавливать), присваивает полю новое значение.

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

Оба метода находятся в секции **public** (общедоступные), позволяя другим объектам работать с полем FColor.

Конечно, проще всего было сделать поле FColor общедоступным. Но при этом:

- другие объекты (и функции, не относящиеся к объектам) смогут самостоятельно изменять значение этого поля; отследить такие изменения может быть очень сложно;

- другие объекты в своей работе смогут использовать знание о внутреннем устройстве класса **CPen**, а этого лучше не допускать, потому что при изменении класса **CPen** придётся искать в программе все связи, которые могут быть нарушены.

Если мы сделаем поле закрытым, то другие объекты «увидят» это поле только через *интерфейс* — методы `getColor` и `setColor`. Поэтому мы можем сколько угодно менять внутреннее устройство и алгоритмы работы объектов класса **CPen**, не меняя интерфейса, и другие объекты об этом даже не «догадаются».

В простейшем случае метод `getColor` просто возвращает значение поля:

```
string CPen::getColor() const
  { return FColor; }
```

В методе `setColor` мы можем обрабатывать ошибки, не разрешая присваивать полю недопустимые значения. Например, потребуем, чтобы символьная строка с кодом цвета состояла ровно из шести символов. Если это условие не выполняется, будем записывать в поле `FColor` код чёрного цвета «000000»:

```
void CPen::setColor( string newColor )
  {
  if( newColor.size() != 6 )
       FColor = "000000";  // чёрный цвет
  else FColor = newColor;
  }
```

Теперь для установки и чтения цвета пера нужно использовать методы класса **CPen**:

```
CPen pen;
pen.setColor( "FFFF00" );
cout << "цвет пера: " << pen.getColor();
```

Итак, мы скрыли внутренние данные, но одновременно обращение к свойствам стало выглядеть некрасиво: вместо `pen.color="FFFF00"` теперь приходится писать так:

```
pen.setColor("FFFF00");
```

Изменение внутреннего устройства

Поскольку доступ к данным класса **CPen** возможен только через методы, внутреннее устройство может быть любым, и его можно менять как угодно. Для сохранения прежних связей с другими объектами нам нельзя изменять интерфейс. Это значит, что возвращаемое значение функции `getColor` и единственный параметр метода `setColor` должны быть символьными строками заданного формата. Всё остальное не влияет на работу других объектов.

Допустим, мы решили хранить цвет как целое число (такой способ чаще всего и используется на практике). Изменим тип поля FColor на целочисленный (**int**):

```
class CPen
{
  private:
    int FColor;
  ...
};
```

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

Итак, метод getColor должен вернуть символьную строку, в которой записан шестнадцатеричный код цвета, хранящегося в поле FColor как целое число. Преобразовывать число в символьную строку будем с помощью строкового потока (stringstream). В него можно записать число, как в обычный поток вывода, при этом число превратится в последовательность символов, т. е. в строку.

Записать в поток число в шестнадцатеричной системе счисления можно так:

```
#include <sstream>     // (1)
...
stringstream s;        // (2)
s << hex << FColor;    // (3)
```

Для работы со строковыми потоками подключаем библиотеку sstream (строка 1). Формат hex обозначает «вывести число в шестнадцатеричной системе счисления». Однако это немного не то, что нужно. Например, цвет FF_{16} будет выведен как "FF" вместо "0000FF". Необходимо как-то указать, что требуется вывести число в 6 позициях, и все пустые позиции слева заполнить нулями. Это делается с помощью *манипуляторов вывода* из библиотеки iomanip, которую тоже надо подключить в начале программы:

```
#include <iomanip>
...
s << setfill('0') << setw(6) << hex << FColor;
```

Здесь используются два манипулятора вывода: setw(6) устанавливает ширину поля 6 символов, а setfill('0') задаёт заполнение пустых позиций символом 0.

В итоге метод getColor приобретает такой вид:

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

Оператор **return** возвращает символьную строку, записанную в поток s, вызывая его метод str.

Обратный переход — от символьной записи к числовому коду — также может быть выполнен с помощью строкового потока. Записываем в поток символьную строку, а затем читаем число:

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

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

Можно обойтись и без потока, вызвав функцию strtol (от англ. *string to long* — из строки в длинное целое) из библиотеки cstdlib:

```
void CPen::setColor( string newColor )
  {
  if( newColor.size()!= 6 )
    FColor = 0;  // если ошибка, то чёрный цвет
  else
    FColor = strtol( newColor.c_str(), nullptr, 16 );
  }
```

Первый аргумент функции strtol — это символьная строка в формате языка C (с кодом 0 на конце), мы получаем её из строки класса string с помощью метода c_str. Второй аргумент — это адрес указателя, который в результате работы функции strtol будет установлен на первый символ после числа. Мы его не используем, поэтому передаём функции нулевой указатель nullptr. Последний аргумент (16) — основание системы счисления.

Итак, внутреннее устройство объектов класса **CPen** изменилось — информация о цвете хранится теперь как целое число, а не как символьная строка. Однако интерфейс класса — способы вызова методов getColor и setColor — остался прежним. Поэтому никакой переделки объектов, взаимодействующих с классом **CPen**, не требуется.

Фактически пара методов getColor и setColor определяют *свойство* (вспомните свойства объектов в языке Python, § 4). Однако такого понятия в стандарте языка C++ нет.

Свойство «только для чтения»

Иногда нужно позволить другим объектам читать какие-то данные, но запретить изменять их. Другими словами, требуется добавить свойство «только для чтения» (англ. *read-only*). В этом случае просто не нужно создавать метод для записи значения скрытого поля:

```
class CCar {
  private:
    double Fv;
  public:
    CCar( double v0 ): Fv(v0) {}
    double getV() const { return Fv; }
  };
```

Поскольку поле Fv объявлено в секции **private**, оно недоступно остальным объектам. Значение этого поля можно прочитать с помощью общедоступного метода getV, но изменить это поле другие объекты не могут.

Короткий код метода getV записан прямо в объявлении класса, т. е. это встроенный (англ. *inline*) метод. Компилятор подставляет в точку вызова всё тело метода, а не оформляет отдельную подпрограмму в машинных кодах.

Свойства в C#

Мы уже говорили о том, что фактически класс CPen определяет свойство «цвет», для работы с которым используются метод чтения («геттер») getColor и метод записи («сеттер») setColor. Однако понятия «свойство» нет в языке C++, и эти два метода формально никак не связаны!

Во многих современных языках программирования введено понятие свойства. Один из таких языков — язык C#, основанный на идеях C и C++. В C# можно ввести свойство color так (сначала рассмотрим простейший случай):

```
class CPen
  {
  private string FColor; // закрытое поле
  public string color    // открытое свойство
    {
    get { return FColor; }   // метод чтения
    set { FColor = value; }  // метод записи
    }
  }
```

Всё очень похоже на описание класса в C++. У объектов такого класса есть закрытое (**private**) поле FColor и открытое (**public**) свойство color. Для работы с этим свойством используются два метода: «геттер» и «сеттер». Метод, записанный в фигурных скобках после слова get, возвращает текущее значение свойства, а метод, записанный после слова set, устанавливает его новое значение. Параметр, который передаётся процедуре-«сеттеру», всегда называется value (это служебное слово языка C#).

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

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

В методе set можно обеспечить защиту от записи неправильного значения:

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

Мы можем легко изменить внутреннее устройство объекта, сохранив интерфейс. В следующем примере свойство color имеет строковый тип, а данные хранятся как целое число:

```
class CPen
  {
  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 («X», от англ. *hexadecimal* — шестнадцатеричный) в шести позициях.

Обратный перевод выполняет метод ToInt32 класса Convert. Его второй аргумент (16) — это основание системы счисления, в которой будет записано число.

Выводы

- Данные и методы класса, которые нужно скрыть от других объектов, в языке C++ объявляются в секции **private**, общедоступные данные и методы объявляются в секции **public**.
- Поля, содержащие данные объекта, обычно делают закрытыми. Для доступа к ним в класс добавляют методы, в которых можно выполнять проверку и преобразование данных.
- Если при изменении внутреннего устройства объектов класса сохранился интерфейс (способ вызова общедоступных методов), другие классы переделывать не нужно.
- Если в классе нет метода для установки нового значения закрытого поля, другие объекты не смогут изменить это поле.

Вопросы и задания

1. Каковы достоинства и недостатки инкапсуляции?
2. Чем различаются секции **public** и **private** в описании классов? Как определить, в какую из них поместить какое-либо свойство или метод?
3. Почему рекомендуют делать доступ к полям объекта только с помощью методов?
4. Как вы думаете, когда можно сделать поле открытым (объявить в секции **public**)? Обсудите этот вопрос с коллегами.
5. Зачем нужны свойства «только для чтения»?
6. Как ввести в класс свойство «только для записи» (которое нельзя прочитать)?
7. Что такое интерфейс объекта? Почему при коллективной разработке программы желательно не изменять заранее согласованные интерфейсы?
8. Что означает «сохранить интерфейс класса» для класса **CPen**, разработанного в параграфе?
9. Измените внутреннее устройство объектов класса **CPen** из параграфа так, чтобы три составляющие RGB-кода цвета хранились в виде массива из трёх элементов.

*10. *Проект*. Измените программу моделирования движения машины (см. предыдущий параграф) так, чтобы все поля у объектов были закрытыми.

Интересные сайты

cppstudio.com — программирование для начинающих на C++

ru.cppreference.com — онлайн-справочник по C++

learncs.org — интерактивный учебник по C#

dotnetfiddle.net — онлайн-компилятор C#

§ 15
Наследование

Ключевые слова:

- класс
- иерархия классов
- базовый класс
- наследование
- абстрактный класс
- виртуальный метод
- защищённые поля
- утечка памяти

Моделирование жизни в океане

В этом параграфе мы начнём разрабатывать новую игру, которая моделирует жизнь в океане. На игровом поле «живут» разнообразные объекты:

- *Камни* (препятствия) — не двигаются;
- *Трава* (планктон) — пища для рыб;
- *Рыбы*, поедающие траву; они не обращают внимания на камни;
- *Хищники* — хищные рыбы, поедающие обычных *Рыб*; они не обращают внимания на камни и траву.

Океан живёт своей жизнью. При этом происходят следующие изменения[1]):

- *Трава* растёт (через заданное время её размеры увеличиваются);
- *Рыбы* двигаются в выбранном направлении (или случайно меняют направление движения);
- если *Рыба* проходит через *Траву*, то размер *Травы* уменьшается, а *Рыба* становится сытой;
- если *Рыба* долгое время не ест, она умирает от голода;
- если *Хищник* сталкивается с *Рыбой*, он ест *Рыбу*; при этом *Рыба* погибает, а *Хищник* становится сытым;
- если *Хищник* долгое время не ест, он умирает от голода.

Наша первая (и самая важная) задача — построить систему классов для этой игры.

Иерархия классов

Все объекты, о которых мы говорили, обладают общими свойствами. Эти свойства можно объединить в базовом классе, который мы назовём `COceanObject` (в переводе с английского — класс «океанский объект»). Конечно, создавать такой объект бессмысленно, потому что это просто набор свойств и методов (т. е. *абстракция*). Объектов типа *Океанский объект* в природе (и в нашей игре) не существует. Поэтому, говоря на языке программистов, класс `COceanObject` будет *абстрактным*, он

[1]) Для научного исследования биологических систем используют модели «хищник — жертва». Самая известная из них — модель Лотки–Вольтерры. Этот материал выходит за рамки пособия, но вы можете самостоятельно найти информацию о таких моделях в сети Интернет.

предназначен только для создания классов-наследников, но не конкретных объектов (вспомните материал § 5).

У подвижных объектов должны быть дополнительные свойства и методы. Объединим их в классе **CMovingObject** (в переводе с английского — движущийся объект). Конечно, этот класс будет наследником класса **COceanObject**. Кроме того, такие «подвижные объекты вообще» тоже не существуют, поэтому класс **CMovingObject** тоже абстрактный.

Очевидно, что наследниками класса **CMovingObject** будут классы **CFish** (*Рыба*) и **CHunter** (*Хищник*). А классы **CStone** (*Камень*) и **CGrass** (*Трава*) — это прямые наследники базового класса **COceanObject**.

Таким образом, мы построили иерархию классов в игре (рис. 2.3).

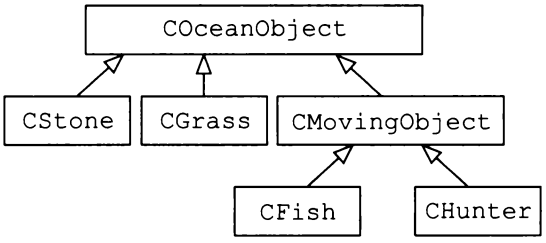


Рис. 2.3

Базовый класс

Введём новый перечисляемый тип данных:

```
enum objectType { STONE, GRASS, FISH, HUNTER };
```

Здесь STONE обозначает *Камень*, GRASS — *Траву*, FISH — *Рыбу* и HUNTER — *Хищника* (от английских слов *stone* — камень, *grass* — трава, *fish* — рыба и *hunter* — охотник).

Для простоты будем изображать все объекты кругами разного цвета: *Камни* — чёрными, *Траву* — зелёными, *Рыб* — синими и *Хищников* — красными.

В базовый класс **COceanObject** добавим три поля, общие для всех объектов: координаты центра круга (x, y) и радиус круга r:

```
class  COceanObject
 {
 public:
   int x, y, r;
 };
```

Теперь определим общие методы, которые должны иметь все океанские объекты. Компьютер выполняет моделирование в дискретном времени, обновляя картинку на экране с некоторым интервалом Δt. В течение очередного интервала моделирования каждый объект может измениться. Могут поменяться размеры или координаты объекта, объект может быть съеден или, наоборот, сам может кого-то съесть. Все эти изменения мы поместим в метод change (по-английски — изменить).

Кроме того, каждый объект должен уметь по команде сам себя перерисовывать. Для этого добавим в базовый класс ещё один метод — show (по-английски — показать).

Эти две операции (изменить и показать) нужно выполнять на каждом шаге моделирования. Поэтому их удобно объединить в новый метод update (по-английски — дополнить). Получается схема базового класса, показанная на рис. 2.4.

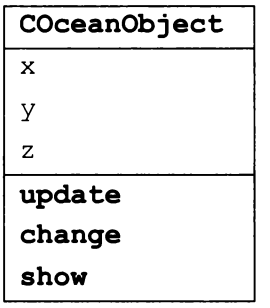


Рис. 2.4

Запишем эту схему в виде класса на языке C++:

```
class COceanObject
  {
  public:
    int x, y, r;
    void update();
    void change();
    void show() const;
  };
```

Метод show не должен изменять объект, поэтому он объявлен с описателем **const**.

Мы не определили конструктор для этого класса, поэтому компилятор создаст конструктор по умолчанию, который просто размещает объект в памяти.

Теперь мы даже можем создать объект этого класса и присвоить значения его полям:

```
COceanObject obj;
obj.x = 100;
obj.y = 100;
obj.r = 10;
```

Конечно, при попытке вызвать метод этого объекта

```
obj.update();
```

мы увидим сообщение об ошибке, ведь реализации этого метода (программного кода) пока нигде нет.

Абстрактный класс

Разбираться с методами класса начнём с метода update. Его содержание написать очень просто: объект изменяется, и затем на экран выводится его новое изображение. Это значит, что сначала вызывается метод change, а потом сразу — метод show. Так будут обновляться объекты всех классов, поэтому метод update мы правильно поместили в базовый класс **TOceanObject**:

```
class COceanObject
  {
  public:
    ...
    void update() {
      change();
      show();
      }
  };
```

Методы change и show есть у всех объектов, поэтому они также должны располагаться именно в базовом классе **COceanObject**. Но каждый наследник будет выполнять их по-разному, т. е. наследники будут *переопределять* методы базового класса. Такие методы обозначаются словом **virtual** (в переводе с английского — виртуальный) в начале объявления метода:

```
class COceanObject
  {
  public:
    ...
    virtual void change();
    virtual void show() const;

  };
```

Обратим внимание на то, что мы *не можем* в базовом классе написать код методов change и show, потому что у всех классов объектов эти методы выполняются по-разному. Мы знаем, как нарисовать камень, траву, рыбу, но никто не знает, как нарисовать «океанский объект вообще».

Конечно, можно добавить в базовый класс методы-«заглушки», которые ничего не делают:

```
class COceanObject
  {
  public:
    ...
    virtual void change() {}
    virtual void show() const {}
  };
```

Пустые фигурные скобки (они выделены фоном) говорят о том, что метод есть, но ни одна команда в нём не выполняется.

Но если мы поставим «заглушки», появится возможность создать и использовать объекты базового класса, что бессмысленно. В этом случае в языке C++ можно применить красивое решение: мы объявим методы в классе, но укажем, что пока они не определены, и поэтому создавать такой класс нельзя. Для этого нужно «приравнять методы к нулю»:

```
class COceanObject
  {
  public:
    ...
    virtual void change() = 0;
    virtual void show() const = 0;
  };
```

Функции, «приравненные к нулю», называются «чистыми» функциями или абстрактными методами. Они обязательно должны быть виртуальными, иначе наследники не смогут их переопределить.

Класс в C++, в котором объявлена хотя бы одна чистая виртуальная функция, называется **абстрактным**. Создавать объекты такого класса нельзя.

Как и в других языках программирования, для того чтобы наследник абстрактного класса **COceanObject** не был абстрактным, он должен переопределить обе чистые виртуальные функции базового класса, `change` и `show`.

Защищённые поля и методы (protected)

Теперь обсудим поля класса **COceanObject**. Сейчас все они открытые, потому что находятся в секции **public**. Как мы уже выяснили в § 14, этого нужно избегать. Конечно, можно сделать их закрытыми (поместить в секцию **private**) и добавить методы для работы с ними («геттеры» и «сеттеры»). Но чтобы не усложнять запись программы (например, не писать `obj.getx()` вместо `obj.x`), мы сделаем по-другому.

В языке C++ кроме **private** и **public** есть ещё один модификатор доступа: **protected** (в переводе с английского — защищённый). Он означает, что поля или методы, размещённые в этой секции, открыты всем наследникам класса, но недоступны объектам всех остальных классов:

```
class COceanObject
  {
  protected:
    int x, y, r;
```

```
  public:
    void update() {
      change();
      show();
      }
    virtual void change() = 0;
    virtual void show() const = 0;
  };
```

Теперь наследники класса **COceanObject** смогут работать с полями базового класса так же, как с открытыми полями, а другие объекты не смогут «увидеть» эти поля.

И тут возникает вопрос: а как же мы будем присваивать начальные значения полям? Имеет смысл делать это сразу при создании объекта, в конструкторе. Следовательно, нам нужно добавить в описание класса конструктор, присваивающий начальные значения полям `x`, `y` и `r`.

Поскольку класс **COceanObject** абстрактный, его конструктор будет вызываться только классами-наследниками. Поэтому конструктор также можно поместить в секцию **protected**:

```
class COceanObject
  {
  protected:
    int x, y, r;
    COceanObject( int x0, int y0, int r0 ):
      x(x0), y(y0), r(r0) { }
...
  };
```

Неподвижные объекты

В нашей программе будут два класса неподвижных объектов — **CStone** (*Камень*) и **CGrass** (*Трава*).

Начнём с класса **CStone**. Как показано на рис. 2.3, он — прямой наследник базового класса **COceanObject**. Поэтому заголовок класса выглядит так:

```
class CStone: public COceanObject
  {
...
  }
```

После названия нового класса **CStone** через двоеточие указан базовый класс **COceanObject**. Служебное слово **public** определяет режим доступа при наследовании. В нашем случае режим доступа не меняется: защищённые (`protected`) данные базового класса остаются защищёнными и для класса-наследника, а открытые — открытыми. В этом пособии мы не будем использовать другие варианты наследования[1)].

[1)] При желании вы легко найдёте эту информацию в литературе или в Интернете.

Приведём полный текст класса **CStone**:

```
class CStone: public COceanObject
  {
  public:
    CStone( int x0, int y0, int r0 ):
      COceanObject( x0, y0, r0 ) {}
    virtual void change() override {}
    virtual void show() const override
      { drawObject( x, y, r, STONE ); }
  };
```

Конструктор класса **CStone** расположен в секции **public**, т. е. его можно вызывать из любой функции. Он принимает три целочисленных параметра — координаты и размер камня, — и сразу вызывает конструктор базового класса **COceanObject**, передавая ему все эти данные. Заметим, что вызов конструктора базового класса записывается через двоеточие сразу после заголовка конструктора класса **CStone**. Сначала всегда выполняется конструктор базового класса, а потом — конструктор производного класса (в данном случае он пустой).

В секции **public** находятся два переопределяемых метода: `change` и `show`. Вообще говоря, слово **virtual** при описании этих методов не обязательно, потому что виртуальные методы базового класса будут виртуальными у всех наследников. Но так программу легче читать — мы сразу видим, что метод виртуальный, не заглядывая в описание базового класса.

Описатель `override` (в переводе с английского — переопределить) говорит о том, что класс **CStone** *переопределяет* метод базового класса, а не добавляет новый метод. Слово `override` тоже можно не писать, но оно ясно выражает намерения программиста и позволяет обнаруживать некоторые ошибки. Если мы попытаемся переопределить метод, которого нет в базовом классе, компилятор сообщит об ошибке.

Обратите внимание, что метод `show` обязательно должен быть объявлен с описателем **const**, потому что такой описатель есть у метода `show` базового класса.

Метод `change` у класса **CStone** — пустой, так как во время анимации никаких изменений с камнем не происходит. А в методе `show` камень выводится на экран с помощью процедуры `drawObject`, которую мы напишем позже. Она должна принимать координаты и размер объекта, а также его тип — значение из перечисления **objectType** (тип объекта для камня — `STONE`).

Класс **CGrass** (*Трава*) строится аналогично, сделайте это самостоятельно[1)].

[1)] Если вам очень хочется поскорее собрать и запустить программу, посмотрите пункты «Вспомогательные процедуры и функции» и «Основная программа» в конце этого параграфа.

Подвижные объекты

Подвижные объекты относятся к классу **CMovingObject**, который наследует все свойства и методы базового класса **COceanObject**:

```
class CMovingObject: public COceanObject
  {
  ...
  }
```

В этом классе к тем полям и методам, которые унаследованы от класса **COceanObject**, добавляются два новых поля — скорость движения (поле v) и курс (поле course), — и новый метод move (в переводе с английского — двигаться):

```
class CMovingObject: public COceanObject
  {
  protected:
    double v,        // скорость
           course;   // курс в градусах

  public:
    CMovingObject( int x0, int y0, int r0,
                   double v0, double course0 = 0 ):
       COceanObject( x0, y0, r0 ),
       v(v0), course(course0)
       {}
    virtual void move()
      {
      double courseRadians = course*M_PI/180;
      x += round(v*cos(courseRadians));
      y -= round(v*sin(courseRadians));
      if( x-r < 0 ) x += SCREEN_WIDTH;
      if( x+r > SCREEN_WIDTH )
        x -= SCREEN_WIDTH;
      if( y-r < 0 ) y += SCREEN_HEIGHT;
      if( y+r > SCREEN_HEIGHT )
        y -= SCREEN_HEIGHT;
      }
  };
```

Конструктор класса **CMovingObject** вызывает конструктор базового класса, а затем записывает в поля v и course дополнительные данные: скорость и курс, которых у базового класса нет. В конструкторе определёно значение по умолчанию для последнего параметра course: если курс не задан, он считается равным нулю.

Новый метод move определяет алгоритм движения объекта. Это виртуальный метод, так что классы-наследники могут его переопределить. Курсовой угол переводится в радианы, затем изменяются координаты x и y (подробно мы обсуждали эти формулы в § 7, см. рис. 1.12).

После этого проверяем, не вышел ли объект за границы игрового поля. Если вышел, то «перекидываем» его на другую сторону поля. Например, если объект вышел за верхнюю границу, он вновь появляется на поле снизу. Здесь используются целочисленные константы — размеры поля:

```
const int SCREEN_WIDTH = 600,
          SCREEN_HEIGHT = 400;
```

Их нужно определить в самом начале программы как глобальные данные.

Обратите внимание, что методы класса **CMovingObject** могут обращаться к полям x, y и r базового класса, так как эти поля — защищённые (**protected**), т. е. доступны всем наследникам класса **COceanObject**.

Заметим, что абстрактные методы change и show базового класса здесь не переопределяются. Это означает, что новый класс **CMovingObject** — тоже абстрактный, и создавать объекты этого класса нельзя. Хотя бы потому, что мы не знаем, как изобразить их на поле.

Рыбы

Рыбы (класс **CFish**) — это подвижные объекты, т. е. наследники класса **CMovingObject**. Класс **CFish** можно описать так:

```
class CFish: public CMovingObject
  {
  public:
    CFish( int x0, int y0, int r0,
           double v0, double course0 ):
      CMovingObject( x0, y0, r0, v0, course0 )
      {}
    virtual void change() override
      { move(); }
    virtual void show() const override
      { drawObject( x, y, r, FISH ); }
  };
```

Конструктор класса просто вызывает конструктор базового класса **CMovingObject**. Метод change вызывает метод move из базового класса, а метод show рисует рыбу с помощью процедуры drawObject, передавая ему последний аргумент FISH.

В описании класса **CFish** определены оба абстрактных метода базового класса — change и show. Мы знаем, как изменять состояние рыбы во время моделирования и как нарисовать её на экране. Поэтому класс **CFish** — не абстрактный класс, и мы можем (и будем) создавать экземпляры этого класса.

Хищники

Займёмся хищниками. Класс **CHunter** (*Хищник*) — это тоже наследник класса **CMovingObject**:

```
class CHunter: public CMovingObject
  {
  ...
  };
```

У нас на поле будет всего один экземпляр класса **CHunter** — им управляет игрок. Хищник должен двигаться с заданной скоростью к той точке игрового поля, на которую указывает курсор мыши.

Определим, куда переместится хищник за один интервал игрового времени Δt. Начальные координаты хищника — x и y, скорость v означает, что за время Δt он проходит v единиц игрового поля (пикселей).

Пусть x_M и y_M — координаты курсора мыши. Тогда расстояние между текущим положением хищника и заданной точкой можно вычислить по теореме Пифагора:

$$d = \sqrt{(x - x_M)^2 + (y - y_M)^2}\,.$$

Далее определим ту часть (долю) этого расстояния, которую пройдёт хищник за один интервал игрового времени Δt:

$$p = \frac{v}{d},$$

и вычислим новые координаты хищника:

$$x_1 = x + p\,(x_M - x), \quad y_1 = y + p\,(y_M - y).$$

Конечно, нужно убедиться, что $d \neq 0$, т. е. хищник ещё не пришёл в нужную точку.

Приведём полностью описание класса **CHunter**:

```
class CHunter: public CMovingObject
  {
  public:
    CHunter( int x0, int y0, int r0, int v0 ):
     CMovingObject( x0, y0, r0, v0 )
      {}
    virtual void change() override
      {
      int xMouse = getMouseX(),
          yMouse = getMouseY();
      double dist = hypot( x - xMouse, y - yMouse );
      if( dist < 1 ) return;
      double part = v / dist;
```

```
      if( part > 1 ) part = 1;
      x += round(part*(xMouse - x));
      y += round(part*(yMouse - y));
      }
    virtual void show() const override
      { drawObject( x, y, r, HUNTER ); }
};
```

В этом классе определяются оба абстрактных метода базового класса: change и show. Поэтому класс **CHunter** — не абстрактный, и мы сможем создать объект-хищник в программе.

В методе change хищник двигается со скоростью v к точке, на которую указывает мышь. Координаты курсора мыши в окне определяются с помощью функций getMouseX и getMouseY, мы их напишем в конце параграфа (пока будем считать, что они есть[1]).

Далее закодирован описанный выше алгоритм. Дистанцию dist между текущим положением объекта и точкой назначения находим с помощью стандартной функции hypot, которая вычисляет гипотенузу прямоугольного треугольника по его катетам. Рассчитанные смещения округляются к ближайшему целому числу с помощью стандартной функции round.

В качестве упражнения попробуйте разобраться, зачем нужна строка

```
if( part > 1 ) part = 1;
```

в методе change и что произойдёт, если её убрать.

Метод show рисует объект на экране, вызывая процедуру drawObject с последним аргументом HUNTER.

Вспомогательные процедуры и функции

Выводить все объекты на экран будет процедура drawObject, которая принимает четыре параметра: координаты объекта x и y (это координаты базовой точки — центра круга), радиус круга r и тип объекта — значение type перечисляемого типа **objectType**:

```
void drawObject( int x, int y, int r,
                 objectType type )
  {
  if( type == STONE )
    drawCircle( x, y, r, RGB(0,0,0) );
  else if( type == GRASS )
    drawCircle( x, y, r, RGB(0,255,0) );
  else if( type == FISH )
    drawCircle( x, y, r, RGB(0,0,255) );
  else /* if( type == HUNTER ) */
    drawCircle( x, y, r, RGB(255,0,0) );
  }
```

[1] Подумайте, какой метод разработки программ мы здесь используем.

Поскольку тип **objectType** включает всего четыре возможных значения, проверку последнего условия (на тип HUNTER) делать не нужно — если первые три не подошли, то остаётся только вариант HUNTER.

Процедура drawCircle должна нарисовать круг с заданными параметрами (последний — это цвет заливки):

```
void drawCircle( int x, int y, int r,
                 COLORREF fillColor )
  {
  txSetColor( RGB(0,0,0) );
  txSetFillColor( fillColor );
  txCircle( x, y, r );
  }
```

Цвет контура выбирается в первой строке — здесь он всегда чёрный (но вы можете всё изменить!).

В методе change класса **CHunter** мы использовали функции getMouseX и getMouseY для определения координат курсора мыши. Эти функции сводятся к вызовам аналогичных функций библиотеки TX Library:

```
int getMouseX() { return txMouseX(); }
int getMouseY() { return txMouseY(); }
```

Для получения случайных целых чисел на отрезке [a; b] мы применим функцию randInt, которая использовалась в предыдущих частях пособия:

```
int randInt( int a, int b ) {
  return a + rand() % (b-a+1);
 }
```

Основная программа

Настало время собрать игру целиком. В начале подключим библиотеку TX Library, введём перечисляемый тип данных **objectType** и константы (размеры поля):

```
#include "TXLib.h"
enum objectType { STONE, GRASS, FISH, HUNTER };
const int SCREEN_WIDTH = 600,
          SCREEN_HEIGHT = 400;
```

Затем запишем все вспомогательные процедуры и функции, которые были только что рассмотрены. После этого поместим описания всех классов, начиная с базового.

В начале основной программы необходимо создать все нужные объекты. Мы покажем, как добавить на поле камни и хищника, остальные объекты вы сможете построить самостоятельно.

Хищник — это объект класса **CHunter**. В нашей программе этот объект будет иметь имя hunter:

```
CHunter hunter( 100, 150, 10, 5 );
```

Здесь вызывается конструктор класса, ему передаются координаты (x = 100, y = 150), размер (r = 10) и скорость хищника (v = 5). Хищник готов.

С камнями получается посложнее. Попытаемся создать массив из 10 объектов класса **CStone**:

```
const int NUMBER_OF_STONES = 10;
CStone stones[NUMBER_OF_STONES];
```

Эти строки вызовут ошибку при компиляции.

Дело в том, что при таком объявлении должен быть вызван конструктор без параметров, которого у нас в классе просто нет! Можно было бы надеяться на конструктор по умолчанию, но компилятор создаёт его только тогда, когда в описании класса вообще нет никакого конструктора. Мы же определили свой конструктор в классе **CStone**, поэтому конструктор по умолчанию не будет построен.

Более того, все данные объекта-камня в базовом классе **COceanObject** защищённые, т. е. недоступные «посторонним» объектам и функциям, в том числе и основной программе. И поэтому задать их значения можно только в конструкторе, при создании объекта.

Но выход есть. Мы создадим массив указателей на объекты[1]. Напомним, что указатель — это переменная, в которой можно хранить адрес данных определённого типа, в том числе и адрес объекта. Объявим массив указателей:

```
CStone* pStones[NUMBER_OF_STONES];
```

Для того чтобы не забыть, что это указатели, имя массива начинается со строчной буквы p (от англ. *pointer* — указатель).

Это ещё не объекты, а переменные, в каждую из которых можно записать адрес какого-то объекта типа **CStone**. Теперь в цикле мы будем на каждой итерации создавать в памяти новый объект-камень и записывать его адрес в очередной элемент массива указателей:

```
for( int i = 0; i < NUMBER_OF_STONES; i++ )
  pStones[i] = new CStone(
                    randInt(0, SCREEN_WIDTH),
                    randInt(0, SCREEN_HEIGHT),
                    randInt(5, 15) );
```

[1] Здесь можно было бы использовать вектор (динамический массив) объектов, в который объекты-камни добавляются по одному. Но вариант с указателями будет нам более удобен при усовершенствовании программы.

В цикле вызывается конструктор с тремя параметрами. Координаты камня задаются случайным образом в пределах игрового поля, а его размер (радиус) — случайное целое число из отрезка [5; 15].

Теперь можно создать графическое окно:

```
txCreateWindow( SCREEN_WIDTH, SCREEN_HEIGHT );
```

и запустить цикл моделирования:

```
while( not GetAsyncKeyState(VK_ESCAPE) ) {
  txSetFillColor( TX_WHITE );      // (1)
  txClear();                       // (2)
  for( int i = 0; i < NUMBER_OF_STONES; i++ )
    pStones[i]->update();          // (3)
  hunter.update();                 // (4)
  txSleep( 50 );                   // (5)
  }
```

Как следует из заголовка, этот цикл заканчивает работу при нажатии клавиши *Escape*. Один интервал игрового времени Δt (время ожидания в конце каждой итерации цикла) равен 50 миллисекунд (строка 5).

В теле цикла сначала устанавливается белый цвет заливки (строка 1) и очищается холст (строка 2). Затем в цикле перебираем все указатели на объекты-камни в массиве `pStones`, для каждого из этих объектов вызываем метод `update`. Обратите внимание, что здесь используется не точечная запись, а оператор `->`, потому что элемент массива `pStones[i]` — это адрес объекта, а не сам объект и не ссылка на него.

В строке 4 вызывается метод `update` для хищника. Здесь другая ситуация: переменная `hunter` — это именно объект, а не его адрес, поэтому мы обращаемся к методу объекта с помощью точечной записи.

Интересно, что в классах **CStone** и **CHunter** нет метода `update`, а мы в программе его вызываем, и всё работает! Но этот метод есть в базовом классе **COceanObject**, наследниками которого являются и **CStone**, и **CHunter**. Значит, у классов **CStone** и **CHunter** тоже есть метод `update`, который им «передал по наследству» базовый класс. Классы-наследники получают «по наследству» все данные и методы всех «классов-предков» (кроме закрытых, **private**).

Другие объекты (траву, рыб) вы можете добавить в программу самостоятельно.

Строго говоря, после того как объекты, созданные с помощью оператора `new`, станут не нужны (в конце программы), их нужно удалить из памяти, вызвав оператор **delete**:

```
for( int i = 0; i < NUMBER_OF_STONES; i++ )
  delete pStones[i];
```

Но здесь это делать не обязательно, поскольку при завершении программы вся выделенная ей память освобождается автоматически.

Проблема может возникнуть, если выделение памяти происходит внутри функции, которая часто вызывается (например, 10 раз в

секунду). Если в конце работы функции ненужная память не освобождается, возможна *утечка памяти* (англ. *memory leak*). Это значит, что выделенный блок памяти не освобождён, а указатель на него потерян. Так может произойти, например, когда функция закончит работу и указатель — локальная переменная — выйдет из области видимости. Выделенная память будет помечена как занятая до конца работы программы, и освободить её не получится, так как обратиться к ней невозможно. В результате при каждом вызове функции выделяются (и не освобождаются) всё новые и новые блоки памяти. В конце концов, свободная память будет исчерпана, и программа завершится аварийно.

Выводы

- Классы в программе образуют иерархию. Базовый класс объединяет общие данные и методы всех объектов.
- Класс-наследник обладает всеми свойствами и методами базового класса.
- Класс, который предназначен только для создания классов-наследников, а не экземпляров (объектов), называется абстрактным. Абстрактный класс объединяет общие данные и методы группы классов.
- Виртуальная функция — это метод, который могут переопределять классы-наследники.
- Чистая виртуальная функция только объявляется в классе, но её программный код (реализация) отсутствует.
- Абстрактный класс в C++ — это класс, который содержит хотя бы одну чистую виртуальную функцию.
- Чтобы класс-наследник не был абстрактным, он должен переопределить все чистые виртуальные функции базового класса. После этого можно будет создавать экземпляры класса-наследника.
- Если новые объекты создавались с помощью оператора **new** (динамически), программа должна освободить эту память с помощью оператора **delete**.

Вопросы и задания

1. Чем полезна иерархия классов? Почему бы не использовать в программе отдельные (не связанные между собой) классы?
2. Что такое метод-«заглушка»? Какие достоинства и недостатки он имеет?
3. Объясните, почему бессмысленно создавать объекты абстрактного класса.
4. Что такое виртуальная функция? Почему некоторые виртуальные функции называют чистыми?
5. Можно ли по описанию класса сразу сказать, абстрактный он или нет? Рассмотрите разные случаи.

6. Можно ли метод update базового класса из параграфа объявить с описателем **const**? Почему?
7. Что такое переопределение метода? Зачем оно используется?
8. В каком случае класс-наследник абстрактного класса сам не будет абстрактным классом?
9. Расскажите о разных способах, с помощью которых класс-наследник может обратиться к полям базового класса. Сравните их, обсудите достоинства и недостатки каждого способа.
10. Найдите все ошибки в описании класса (в этом и следующих заданиях используются классы, приведённые в параграфе).

```
class CStone( COceanObject );
  {
    /* какие-то объявления */
  }
```

11. Найдите все ошибки, которую допустил программист:

```
COceanObject fish;
CHunter hunter;
```

Предложите варианты исправления этих ошибок.

12. Сравните свойства данных и методов, описанных в секциях **private**, **protected** и **public**. Как выбрать нужную секцию для нового поля или метода?
13. Как вы думаете, что произойдёт, если конструктор класса объявить в секции **private** или **protected**? Проверьте ваши выводы экспериментально.
14. Зачем при переопределении виртуальных методов в классах-наследниках обычно пишут слово **virtual**, хотя можно этого и не делать?
15. Используя дополнительные источники, выясните, что означает служебное слово **final** и зачем его можно использовать.
16. Какая серьёзная ошибка может случиться, если убрать из метода change класса **CHunter** следующую строку?

```
if( dist < 1 ) return;
```

Проверьте ваш ответ, запустив программу на компьютере.

17. Что произойдёт, если убрать из метода change класса **CHunter** следующую строку?

```
if( part > 1 ) part = 1;
```

Проверьте ваш ответ, запустив программу на компьютере, и объясните результат.

18. Какая ошибка произойдёт, если создать массив объектов класса **CStone** так?

```
CStone stones[10];
```

Чем она вызвана? Как можно доработать программу, чтобы такой способ тоже был возможен?

19. Придумайте свой тип океанских объектов и добавьте его в иерархию на рис. 2.3.

20. Проведите рефакторинг программы. Методы, состоящие из нескольких строк (например, метод `move` класса **`CMovingObject`**), оформите как отдельные подпрограммы вне описания класса.

*21. *Проект*. Вынесите все классы в отдельный модуль. Создайте для него заголовочный файл. Учтите, что все подпрограммы, которые используют функции библиотеки `TX Library` (например, `drawCircle`, `getMouseX`, `getMouseY`), должны находиться в одном файле — там же, где и основная программа с именем `main`.

**22. Предусмотрите такой метод добавления объектов на поле, чтобы объекты не перекрывали друг друга.

**23. Сделайте так, чтобы хищник не мог «проходить сквозь камни». При столкновении с камнем он должен перекатиться вокруг него.

Интересные сайты

cplusplus.com — сайт, посвящённый программированию на языке C++

rsdn.org — сайт, посвящённый разработке программного обеспечения (на разных языках)

§ 16 Полиморфизм

Ключевые слова:

- базовый класс
- класс-наследник
- виртуальный метод
- полиморфизм
- позднее связывание
- таблица виртуальных методов
- деструктор

В § 5 мы познакомились с полиморфизмом. Так называется возможность классов-наследников по-разному реализовать метод базового класса. Виртуальные методы `change` и `show` в построенных ранее классах морских объектов — это как раз пример полиморфизма. В этом параграфе мы увидим, что виртуальные методы особенно полезны при работе с коллекцией объектов разных классов через указатели.

Указатели на базовый класс

Добавляя на игровое поле объекты, вы, скорее всего, записывали адреса объектов каждого типа в отдельный массив. Например, так:

```
const int NUMBER_OF_STONES = 10;
CStone* pStones[NUMBER_OF_STONES];
for( int i = 0; i < NUMBER_OF_STONES; i++ )
  pStones[i] = new CStone( ... );
```

```
const int NUMBER_OF_GRASS = 10;
CGrass* pGrass[NUMBER_OF_GRASS];
for( int i = 0; i < NUMBER_OF_GRASS; i++ )
  pGrass[i] = new CGrass( ... );
```

Здесь многоточия внутри скобок обозначают аргументы конструкторов, которые мы здесь не написали для экономии места.

Однако можно было сделать проще — записывать все указатели в один массив. Это кажется странным, ведь указатель предназначен для хранения адресов объектов конкретного типа. Например, фрагмент кода

```
CStone *p1;
CGrass *p2;
```

вводит два указателя: в указатель `p1` можно записать адрес любого объекта класса **CStone**, а в `p2` — адрес любого объекта класса `CGrass`. В общем случае записать в указатель одного типа адрес объекта другого типа нельзя. Но есть одно исключение.

Объявим указатель так:

```
COceanObject *pObject;
```

Кажется, что это совершенно бессмысленно, ведь в нашей программе **COceanObject** — это абстрактный класс, и создать объект этого класса невозможно: компилятор просто не позволит этого сделать.

Однако в языке C++ такой указатель может хранить адрес *любого наследника класса* **COceanObject**. Это вполне естественно, ведь любой наследник «умеет» делать всё, что «умеет» объект базового класса.

Область памяти, занимаемую объектом класса-наследника (например, класса **CStone**), можно разделить на две части: сначала размещаются данные и методы базового класса **COceanObject**, а затем — данные и методы, определённые в классе-наследнике (рис. 2.5).

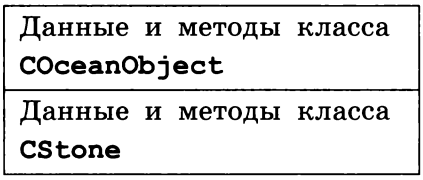


Рис. 2.5

Указатель на любой объект класса-наследника хранит начальный адрес этой области, поэтому через него можно легко «добраться» до всех данных и методов базового класса.

В переменную `pObject`, которая объявлена выше, можно записать адрес нового объекта-камня:

```
pObject = new CStone( 100, 100, 10 );
```

Потом можно удалить этот объект-камень из памяти и в ту же переменную записать адрес нового объекта-рыбы:

```
delete pObject;
pObject = new CFish( 100, 100, 5, 2, 120 );
```


В переменную, объявленную как указатель на объект класса A, можно записать адрес любого объекта-наследника класса A.

Используя эту особенность, построим общий массив указателей для хранения адресов всех океанских объектов. Вот как это выглядит для камней и травы:

```
#include <vector>
using namespace std;
...
vector <COceanObject*> pObjects;

const int NUMBER_OF_STONES = 10;
for( int i = 0; i < NUMBER_OF_STONES; i++ )
  pObjects.push_back ( new CStone(
                randInt(0, SCREEN_WIDTH),
                randInt(0, SCREEN_HEIGHT),
                randInt(5, 15) ) );

const int NUMBER_OF_GRASS = 10;
for( int i = 0; i < NUMBER_OF_GRASS; i++ )
  pObjects.push_back ( new CGrass(
                randInt(0, SCREEN_WIDTH),
                randInt(0, SCREEN_HEIGHT),
                randInt(5, 15) ) );
```

Здесь вместо статического массива заранее определённого размера используется динамический массив (вектор), размер которого можно изменять во время работы программы.

В начале программы подключаем библиотеку **vector** и пространство имён std, в котором объявлены все стандартные функции библиотеки языка C++.

Объявление

```
vector <COceanObject*> pObjects;
```

вводит новый вектор с именем pObjects. В угловых скобках записан тип элементов этого массива: указатель на объект класса **COceanObject**.

Сразу после объявления массив пустой, в нём нет ни одного элемента. Метод push_back добавляет новый элемент в конец массива. В двух циклах мы с помощью этого метода добавляем в массив сначала 10 адресов вновь созданных объектов-камней, а потом — 10 адресов новых объектов типа **CGrass**.

Полиморфизм в действии

Теперь возникает вопрос: что делать с этим массивом, в который записаны адреса объектов *разных* типов, и (это важно!) являющихся

наследниками одного базового класса **COceanObject**. Попробуем в цикле вызвать метод update для каждого объекта:

```
for( int i = 0; i < pObjects.size(); i++ )
  pObjects[i]->update();
```

Запустив программу, вы убедитесь, что все объекты нарисованы правильно! Действительно, метод update — это метод базового класса **COceanObject**:

```
class COceanObject
  {
  ...
  public:
    void update() {
      change();
      show();
      }
  };
```

Но он вызывает методы change и show, которых в базовом классе нет. Все объекты нарисованы правильно (камни — чёрные, трава — зелёная). Это значит, что для объектов класса **CStone** были вызваны методы change и show из класса **CStone**, а для объектов класса **CGrass** — методы с такими же именами из класса **CGrass**.

Фактически программа *во время выполнения* определила, к какому классу относится конкретный объект, и вызвала именно «его» методы.

При вызове *виртуального* метода через указатель или ссылку на объект программа *во время выполнения* определяет фактический класс объекта и вызывает метод того класса, к которому этот объект принадлежит.

Таким образом, полиморфизм позволяет использовать один и тот же интерфейс (способ вызова метода) для управления объектами разных классов.

Немного теории

Полиморфизм можно использовать при обращении к объекту не только через указатель, но и через ссылку. Пусть у нас есть два класса[1]: **CCat** и **CPurringCat**, причём класс **CPurringCat** — наследник класса **CCat**:

```
class CCat
  {
  public:
    virtual void say() {
      cout << "Мяу!" << endl;
      }
  };
```

[1] По-английски *cat* — кошка, а *purring cat* — мурлыкающая кошка.

```
class CPurringCat: public CCat
  {
  virtual void say() {
    cout << "Мурр!" << endl;
    }
  };
```

В этих классах есть виртуальный метод say, который выводит сообщение на экран.

Рассмотрим процедуру, которая принимает один параметр — объект класса **CCat** — и вызывает его метод say:

```
void voice( CCat cat ) {
  cat.say();
  }
```

В основной программе создадим объекты классов **CCat** и **CPurringCat** и вызовем для каждого из них процедуру voice:

```
CCat cat;
CPurringCat purrCat;
voice( cat );            // Мяу!
voice( purrCat );        // Мяу!
```

Программа успешно компилируется, несмотря на то что в последней строке мы передали объект другого класса, **CPurringCat** вместо **CCat**. Это возможно только потому, что класс **CPurringCat** — наследник класса **CCat**. Согласно правилам языка C++, процедуре voice можно передать объект любого класса-наследника **CCat**, поскольку любой такой объект обладает всеми свойствами и методами класса **CCat**.

Вызов процедуры для объекта cat даёт ожидаемый результат — текст «Мяу!». Кажется, что при втором вызове программа должна вывести «Мурр!», но это не так. Дело в том, что объект передаётся в эту процедуру *по значению*. В этом случае в стеке строится временный объект класса **CCat**, и для него вызывается метод say.

Теперь введём вторую процедуру, параметр которой уже не объект, а ссылка на него:

```
void voiceRef ( CCat& cat ) {
  cat.say();
  }
```

и снова вызовем её для обоих объектов:

```
voiceRef( cat );         // Мяу!
voiceRef( purrCat );     // Мурр!
```

Здесь объект передаётся по ссылке (временный объект не создаётся), и это может быть объект любого класса-наследника **CCat**. С одной стороны, мы сообщили компилятору, что параметр cat в процедуре — это ссылка на объект класса **CCat**. Это его *статический тип*, т. е. тип,

известный в момент компиляции. С другой стороны, при вызове процедуры переданная ссылка может указывать на объект любого класса-наследника **CCat**, это *динамический тип* параметра, который может отличаться от статического. Например, в последней строке нашей программы динамический тип параметра cat — это **CPurringCat**. В этом случае, как и при обращении через указатель, программа определяет фактический тип объекта и вызывает метод say класса **CPurringCat**.

Позднее связывание

Если мы хотим использовать полиморфизм при вызове методов через указатели, важно, чтобы эти методы были виртуальными, для невиртуальных методов такой механизм не работает.

Предположим, что в базовом классе **COceanObject** методы change и show *не* объявлены виртуальными (например, мы поставили вместо них «заглушки»). Тогда при компиляции в машинный код сразу записываются команды перехода по адресам в памяти, где расположены эти (обычные, невиртуальные) методы.

Пусть p — указатель на объект базового класса, и мы записали в него адрес класса-наследника:

```
COceanObject* p = new CStone( ... );
```

Теперь вызываем метод update, объявленный в базовом классе:

```
p->update();
```

И тут выясняется, что в машинном коде метода update адреса переходов на методы change и show *уже прописаны*. В итоге программа выполнит методы-«заглушки» базового класса и ничего не нарисует.

У нас же методы change и show — виртуальные. В этом случае компилятор «соображает», что наследники могут переопределить эти методы, и поэтому не записывает в машинный код метода update фиксированные адреса переходов на методы change и show. Вместо этого он добавляет довольно сложные команды, при выполнении которых:

- определяется фактический тип объекта;
- в *таблице виртуальных методов объекта* (англ. *VMT*: *virtual method table*) ищется адрес нужного метода (например, метода change);
- выполняется метод, расположенный по найденному адресу.

Потом так же происходит вызов виртуального метода show.

Такой механизм называется *поздним связыванием*. Это означает, что программа определяет адрес выполняемого метода не при компиляции, а во время выполнения программы.

При позднем связывании:

- вместе с каждым объектом хранится таблица его виртуальных методов;

- выполняются дополнительные действия — поиск нужного метода, а не просто переход по известному адресу.

Позднее связывание всегда замедляет работу программы. Поэтому нужно объявлять виртуальными только те методы, которые наследники действительно могут и будут переопределять, а не все методы всех объектов подряд.

Класс Океан

До этого момента мы предполагали, что в нашей игре все обитатели океана действуют независимо друг от друга. На самом деле они принадлежат единой системе *Океан*. При программировании игр такой объект иногда называют игровым миром.

Если говорить на языке ООП, можно ввести ещё один «большой» объект — *Океан*, — который «владеет» всеми другими объектами. Назовём этот класс **COcean**:

```
class COcean
  {
  };
```

Информацию о его «подчинённых» объектах — камнях, траве и др. — будем хранить как вектор (массив) указателей в закрытом поле объекта с именем pObjects:

```
class COcean
  {
  private:
    vector <COceanObject*> pObjects;
  };
```

В классе **COcean** создадим два метода. Первый метод — addObject — будет добавлять новый объект, а второй — update — будет выполнять обновление всех объектов в конце каждого интервала моделирования. Оба этих метода сделаем общедоступными:

```
class COcean
  {
  private:
    vector <COceanObject*> pObjects;
  public:
    void addObject( objectType type,
                    int x0, int y0, int r0 = 10,
                    int v0 = 0, double course0 = 0 );
    void update();
  };
```

Первый параметр метода addObject — код объекта, который нужно создать. Это величина перечисляемого типа **objectType**. Остальные параметры — координаты и размер объекта, а также его скорость v0 и курс course0. Последние два параметра используются только для подвижных объектов, поэтому для них определены значения по умолчанию: если эти параметры не заданы, они принимаются равными нулю. Размер r0 тоже можно не задавать, если нас устраивает значение по умолчанию, равное 10.

Реализацию (программный код) метода addObject выносим из описания класса:

```
void COcean::addObject( objectType type,
                        int x0, int y0, int r0,
                        int v0, double course0 )
  {
  COceanObject* pNewObj = nullptr;          // (1)
  if( type == STONE )                       // (2)
    pNewObj = new CStone( x0, y0, r0 );
  else if( type == GRASS )                  // (3)
    pNewObj = new CGrass( x0, y0, r0 );
  else if( type == FISH )
    pNewObj = new CFish( x0, y0, r0, v0, course0 );
  else /* if( type == HUNTER ) */
    pNewObj = new CHunter( x0, y0, r0, v0);
  if( pNewObj )                             // (4)
    pObjects.push_back( pNewObj );          // (5)
  }
```

Обратите внимание, что значения параметров по умолчанию здесь указывать не нужно, они задаются в прототипе метода при объявлении класса.

Разберём работу этого метода подробно. В строке 1 создаётся переменная pNewObj — указатель на объект базового класса **COceanObject**. В такой указатель можно записывать адрес любого объекта-наследника — базового класса. Мы сразу записываем в него значение по умолчанию — «нулевой указатель» nullptr.

В строке 2 проверяем переданный тип объекта: если это тип STONE, создаём в памяти новый объект-камень класса **CStone** и записываем его адрес в указатель pNewObj. Объект создаётся с помощью оператора **new**, который вызывает конструктор класса **CStone**. Если переданный тип — не STONE, продолжаем проверку дальше: если тип равен GRASS, строим объект-траву (строка 3), и т. д.

Проверку на тип HUNTER выполнять не нужно (этот условный оператор поставлен в комментарий). Дело в том, что перечисляемый тип **objectType** включает всего четыре варианта. Если первые три не подошли, остаётся последний — HUNTER.

В строке 4 проверяем, успешно ли создан объект. Если ошибок не было, в указатель pNewObj будет записан ненулевой адрес нового объекта, и мы добавляем его в конец вектора pObjects. В океане появился новый объект.

Если произошла ошибка, то в указателе pNewObj останется нулевое значение (nullptr) — признак того, что создать объект не удалось. При этом условие в строке 4 не выполнится, и вектор pObjects не изменится.

Второй метод класса **COcean** занимается обновлением состояния океана:

```
void COcean::update()
  {
  for( int i = 0; i < pObjects.size(); i++ )
    pObjects[i]->update();
  }
```

Здесь для каждого объекта, адрес которого хранится в векторе pObjects, вызывается метод update. Обращение к методу объекта по адресу выполняется с помощью оператора ->.

Используя возможности стандарта C++11, можно записать те же действия более красиво и кратко:

```
void COcean::update()
  {
  for( auto pObj: pObjects )
    pObj->update();
  }
```

Такой цикл перебирает все указатели, входящие в вектор pObjects. На каждой итерации цикла в переменной pObj оказывается адрес очередного объекта, и для этого объекта вызывается метод update. Служебное слово **auto** означает, что тип переменной pObj компилятор определит автоматически. Он будет совпадать с типом элементов вектора pObjects, т. е. pObj — это указатель на объект класса **COceanObject**.

Метод update принадлежит базовому классу **COceanObject** и вызывает ещё два метода: change и show. Они объявлены как виртуальные, поэтому для каждого объекта вызываются «свои» версии этих методов.

Создадим в основной программе объект класса **COcean**:

```
COcean ocean;
```

Добавим в океан камни, траву, рыб. Например, для добавления камней можно использовать такой цикл:

```
const int NUMBER_OF_STONES = 10;
for( int i = 0; i < NUMBER_OF_STONES; i++ )
  ocean.addObject( STONE,
          randInt( 0, SCREEN_WIDTH ),
          randInt( 0, SCREEN_HEIGHT ),
          randInt( 5, 15 ) );
```

Другие типы объектов добавляются аналогично. Наконец, добавим одного хищника:

```
ocean.addObject( HUNTER, 100, 100, 10, 5 );
```

Он тоже будет одним из объектов в общем списке.

В теле цикла анимации теперь не нужно обновлять отдельно каждый объект, достаточно вызвать метод update для главного объекта ocean, а он уже «разберётся» сам со своими обитателями:

```
while( not GetAsyncKeyState(VK_ESCAPE) ) {
  txSetFillColor( TX_WHITE );
  txClear();
  ocean.update();
  txSleep( 50 );
  }
```

Деструктор

В методе addObject класса **COcean** мы создаём новые объекты с помощью оператора new, сохраняя их адреса в массиве pObjects. Чтобы не было утечек памяти, важно сразу определить, где будет освобождаться выделенная память.

Очевидно, что при уничтожении объекта *Океан* все другие объекты, которые находятся в его «владении» (в массиве pObjects), тоже нужно удалить. Поэтому возникает вопрос: как «поймать» момент удаления объекта?

Для этой цели в каждом классе C++ есть специальный метод, который называется деструктором.

Деструктор — это метод, который вызывается автоматически при удалении объекта из памяти.

Если конструкторов у класса может быть много, то деструктор всегда только один. Вызывать деструктор специально не нужно, программа вызовет его автоматически при выходе объекта из области видимости или при удалении с помощью оператора **delete**.

Раньше мы никогда не добавляли деструкторы в описание класса. При этом компилятор сам строит деструктор по умолчанию, который просто освобождает память, занимаемую самим объектом.

Особый деструктор нужен тогда, когда объект захватывает какие-то ресурсы: выделяет память, открывает файл и т. п. Тогда в деструкторе эти ресурсы нужно освободить. Например, конструктор файлового потока

```
ifstream Fin( "input.txt" );
```

открывает файл на чтение. Когда поток выходит из области видимости, деструктор класса ifstream закрывает файл и снимает с него блокировку.

В нашем случае объект *Океан* хранит в векторе pObjects указатели на блоки памяти, выделенные с помощью оператора **new**. При уничтожении объекта все его поля тоже уничтожаются, значит, использовать эти указатели мы больше не сможем. Поэтому не сможем и освободить блоки памяти, на которые они ссылаются[1]. Следовательно, освобождать эту память нужно при уничтожении объекта *Океан*, т. е. в деструкторе класса **COcean**. Таким образом, особый деструктор нам нужен для того, чтобы освободить *захваченные ресуры* — память, выделенную с помощью оператора **new**.

Деструктор — это специальный метод класса, имя которого начинается со знака «~», а за ним записывают название класса:

```
class COcean
  {
  public:
    ...
    ~COcean(); // это деструктор
  }
```

Деструктор, как и конструктор, — это не процедура и не функция, а особый метод. Он ничего не возвращает, даже **void**, и не принимает никаких параметров (в отличие от конструктора).

В теле нового деструктора нужно в цикле перебрать все указатели в массиве pObjects и для каждого из них выполнить оператор **delete** (удалить из памяти):

```
COcean::~COcean()
  {
  for( auto pObj: pObjects )
    delete pObj;
  }
```

При выполнении оператора **delete** объект, на который указывает очередной указатель obj, удаляется из памяти. При этом для него вызывается деструктор. Но какой именно деструктор будет вызван?

Вспомним, что pObjects — это вектор указателей на объекты базового класса **COceanObject**. Ни в этом базовом классе, ни в его наследниках мы не определяли деструкторы, поэтому компилятор построит деструкторы по умолчанию. Но эти деструкторы — не виртуальные, поэтому для всех объектов будет вызываться деструктор базового класса. Это значит, что разрушается и удаляется из памяти только та часть объекта, за которую «отвечает» базовый класс. Чтобы исправить ситуацию, деструктор базового класса нужно сделать виртуальным, добавив его в базовый класс **COceanObject**:

[1] Если, конечно, мы не запомнили эти адреса где-то в другом месте.

```
class COceanObject
  {
  public:
    ...
    virtual ~COceanObject() = default;
  };
```

Служебное слово **default** после знака = говорит о том, что компилятору нужно построить деструктор по умолчанию, но сделать его виртуальным. В классах-наследниках деструкторы объявлять не нужно: они станут виртуальными автоматически.

Деструкторы классов-наследников выполняются в обратном порядке (по сравнению с конструкторами): сначала — деструктор производного класса, затем — деструктор базового класса.

Выводы

- В переменную, объявленную как указатель на объект класса A, можно записать адрес любого объекта-наследника класса A.
- При вызове виртуального метода через указатель или ссылку на объект программа во время выполнения определяет фактический класс объекта и вызывает метод того класса, к которому этот объект принадлежит.
- При позднем связывании программа определяет адрес выполняемого виртуального метода не при компиляции, а во время выполнения, в зависимости от типа объекта.
- При позднем связывании используется таблица виртуальных методов, которая хранится вместе с каждым объектом.
- Позднее связывание замедляет работу программы.
- Деструктор — это специальный метод, который вызывается при удалении объекта из памяти. Если деструктор не указан, компилятор построит деструктор по умолчанию.
- Деструктор необходимо определять, когда объект захватывает ресурсы, например содержит указатели на блоки памяти, выделенные во время работы программы.
- Если класс содержит виртуальные методы, его деструктор должен быть объявлен виртуальным.

Вопросы и задания

1. Найдите все ошибки в этом фрагменте программы (в этом и следующих заданиях используются объекты, приведённые в параграфе):

```
CFish p;
p = new CStone( 100, 100, 10 );
p.show();
p = new COceanObject( 100, 100, 15 );
p = new CFish( 100, 100, 15, 2, 0 );
p->update;
```

Предложите способ исправления этих ошибок.

2. Оцените преимущества и недостатки использования статического массива и динамического массива (вектора) для хранения адресов игровых объектов.
3. Обсудите достоинства и недостатки обычного («раннего») и позднего связывания.
4. Проверьте экспериментально, может ли класс-наследник переопределить метод базового класса, не объявленный виртуальным. Создайте новый объект и запишите его адрес в указатель на базовый класс, например:

```
COceanObject* pStone = new CStone(100, 100, 5);
```

   Теперь вызовите через этот указатель переопределённый невиртуальный метод. Выясните, какой метод будет вызван: метод базового класса или класса-наследника. Предположите, как связан этот результат с таблицей виртуальных методов.
5. Как вы думаете, зачем нужен класс **COcean**?
6. В классе `COcean` не введен конструктор. Как же создаётся объект?
7. Почему вектор `pObjects` в классе **COcean** объявлен в секции **private**?
8. Имеет ли смысл объявить метод `update` класса **COcean** виртуальным? Обсудите достоинства и недостатки такого решения.
9. Можно ли было объявить метод `update` класса **COcean** с описателем **const**? Как вы думаете, почему мы так не сделали (с учётом будущей доработки программы)?

*10. Найдите в дополнительных источниках информацию об операторе **switch** и перепишите метод `addObject` класса **COcean** с помощью оператора **switch**. Оцените достоинства и недостатки такого решения.

11. Приведите примеры ситуаций, когда нужно обязательно явно определить деструктор класса.
12. Василий хочет записать программный код (реализацию) всех методов своих классов прямо в объявлении класса. Обсудите достоинства и недостатки такого решения.

*13. *Проект.* Измените программу так, чтобы по щелчку мышью хищник переключался в режим «блуждания» — менял направление движения случайным образом. Повторный щелчок мышью по игровому полю должен возвращать его в режим слежения за мышью.

14. Попробуйте добавить в океан несколько хищников. Что при этом будет происходить?

Интересные сайты

ru.stackoverflow.com — сайт вопросов и ответов для программистов

learncpp.com — онлайн-учебник по языку C++

§ 17
Взаимодействие объектов

Ключевые слова:

- столкновения
- «мёртвые» объекты
- итератор
- преобразование типов

Столкновения

Давайте подумаем, а зачем нам вообще нужен класс *Океан* («игровой мир»). Если он просто хранит указатели на все объекты, то особой необходимости в нём нет — можно использовать обычный вектор.

Вспомним, что наши объекты должны взаимодействовать друг с другом. Например, хищник ест рыб, а рыбы едят траву. Организацию этого процесса мы можем поручить классу **COcean**.

Добавим в класс **COcean** два открытых метода, которые обрабатывают столкновения и их последствия:

```
class COcean
  {
  ...
  public:
     void checkCollisions() const;
     void removeDead();
  };
```

Константный метод `checkCollisions` (от англ. «проверить столкновения») проверяет для каждой пары объектов, произошло ли столкновение, а метод `removeDead` — удаляет из океана мёртвые объекты.

Добавим вызовы этих методов в код метода `update`:

```
void COcean::update()
  {
  for( auto pObj: pObjects )
    pObj->update();
  checkCollisions();
  removeDead();
  }
```

Таким образом, после того как все объекты изменят своё положение, мы обрабатываем столкновения между ними и удаляем «мёртвые» объекты.

В методе `removeDead` какие-то объекты могут быть удалены с поля, при этом изменится и вектор `pObjects`. Поэтому метод `update` нельзя объявлять константным.

Перед тем как написать код новых методов, нужно решить, кто (объект-океан или объекты-обитатели) должен уметь «отвечать» на следующие вопросы.

- Столкнулся ли объект *A* с объектом *B*?
- Что произошло с объектом *A* при столкновении с объектом *B*?

Если мы решим, что этим будет заниматься объект-океан, то при появлении нового типа морских обитателей или нового поведения старых обитателей нам придётся менять код класса **COcean**. А это неправильно с точки зрения ООП — океан-то не поменялся! Поэтому вполне разумно передать решение этих вопросов классам объектов, населяющих океан. Отсюда следует, что все объекты — наследники класса **COceanObject** — должны иметь два метода:

- метод `hasCollisionWith` (в переводе с английского — «имеет столкновение с»), который возвращает логическое значение `true`, если этот объект столкнулся с другим объектом, адрес которого ему передали;
- метод `collideWith` (в переводе с английского — «столкнуться с»), в котором описаны изменения, происходящие с этим объектом при столкновении.

Предполагая, что такие методы есть у базового класса **COceanObject**, можно написать код метода `checkCollisions`:

```
void COcean::checkCollisions() const
  {
  for( int i = 0; i < pObjects.size(); i++ )
    for( int j = i+1; j < pObjects.size(); j++ )
      if( pObjects[i]->hasCollisionWith (pObjects[j]) ) {
        pObjects[i]->collideWith( pObjects[j] );
        pObjects[j]->collideWith( pObjects[i] );
        }
  }
```

Вложенный цикл перебирает все пары объектов. Обратите внимание, что переменная `j` внутреннего цикла начинает изменяться со значения `i+1`, т. е. рассматриваются все пары объектов с номерами (`i`, `j`), для которых `j > i`. При этом мы никогда не проверяем столкновение объекта с самим собой (это могло бы произойти при `i = j`).

Сначала выясняем, было ли столкновение — вызываем метод `hasCollisionWith` для первого выбранного объекта (с номером `i`) и передаём ему адрес второго выбранного объекта (с номером `j`). Если эта функция вернула истинное значение (`true`), вызываем метод `collideWidth` для каждого из объектов (оба могут измениться в результате столкновения).

Заметьте, что мы применяем проектирование «сверху вниз», методы `hasCollisionWith` и `collideWidth` пока не написаны, а метод `checkCollisions` класса **COcean** уже готов.

Ещё один новый метод класса **COcean**, названный `removeDead` (в переводе с английского — «удалить мёртвых»), должен удалять из памяти и из списка `pObjects` все объекты, которые погибли в результате столкновений на очередном интервале моделирования.

Будем считать, что объект сам умеет определять, мёртвый ли он, т. е. имеет общедоступный метод isDead. Этот метод должен быть у всех объектов, поэтому объявить его нужно в базовом классе. Метод isDead вернёт значение true, если требуется удалить объект с игрового поля.

Метод removeDead перебирает все объекты в списке pObjects:

```
void COcean::removeDead()
  {
  for( int i = pObjects.size()-1; i >= 0; i-- )
    if( pObjects[i]->isDead() ) {
      delete pObjects[i];                    // (1)
      pObjects.erase( pObjects.begin()+i ); // (2)
      }
  }
```

Если какой-то объект «мёртв», занятая им память освобождается (строка 1), и ссылка на него удаляется из списка pObjects (строка 2). Заметим, что примерно так работает сборщик мусора во многих современных языках программирования, например в языке Python.

Методу erase в строке 2 передаётся *итератор* — указатель специального типа на удаляемый элемент pObjects[i]. Здесь pObjects.begin() — это итератор, указывающий на первый элемент списка (он имеет индекс 0). Если добавить к нему значение целочисленной переменной i, получается указатель на элемент вектора pObjects[i], который нужно удалить.

Обратите внимание, что перебор массива указателей в цикле выполняется в обратном порядке, от последнего к первому. Если использовать стандартный вариант, когда индекс увеличивается после каждого шага, то мы без проверки «проскочим» элементы, следующие за удалёнными, и не удалим их, даже если они будут «мёртвыми».

Например, пусть цикл организован так:

```
for( int i = 0; i < pObjects.size(); i++ )
  ...
```

Предположим, что элемент с индексом 0 оказался «мёртвым» и был удалён. При этом на его место встал элемент, который раньше имел индекс 1, теперь он имеет индекс 0. Но переменная i после первой итерации будет автоматически увеличена и станет равна 1, поэтому новый элемент с индексом 0 мы не будем проверять.

Изменение базового класса

Итак, в базовый класс нужно добавить три новых метода: hasCollisionWith, collideWith и isDead.

Очевидно, что методы collideWith и isDead должны быть виртуальным, потому что каждый наследник класса **COceanObject** может определять их по-своему.

Добавляем новые методы в секцию **public**:

```
class COceanObject
  {
    ...
  public:
    ...
    virtual bool isDead() { return (r == 0); }
    bool hasCollisionWith( COceanObject* pOther ) const;
    virtual void  collideWith( COceanObject* pOther ) {}
  };
```

Мы будем считать, что любой объект-обитатель изображается как круг, и в поле r хранится радиус этого круга. По умолчанию будем считать «мёртвым» объект, радиус которого равен нулю. В этом случае метод isDead, записанный прямо в объявлении базового класса, возвращает истинное значение. В классе-наследнике этот метод можно переопределить, так как он объявлен виртуальным.

Метод collideWith принимает один параметр — адрес объекта, с которым произошло столкновение. Тело метода по умолчанию пустое. Классы-наследники, которые обрабатывают столкновения с другими объектами, могут переопределить этот виртуальный метод.

Напомним, что код всех методов, записанных в объявлении класса, компилятор добавляет прямо в точку вызова, не оформляя его как отдельную программу. Хотя при этом выполнение программы немного ускоряется, размер кода увеличивается. Поэтому рекомендуется применять этот вариант только для очень коротких методов, длиной в одну-две строки.

Третий метод, hasCollisionWith, вынесем из описания класса. В этом случае он будет оформлен как отдельная подпрограмма, а не включён полностью в код в месте вызова.

```
bool COceanObject::hasCollisionWith(
                    COceanObject* pOther ) const
  {
  double distance = hypot( x - pOther->x, y - pOther->y );
  return (distance < r + pOther->r);
  }
```

Здесь проверяется, было ли столкновение двух объектов — текущего (для которого вызван метод) и второго, адрес которого передан методу как единственный параметр pOther.

Предполагается, что все объекты изображаются как круги. Расстояние между базовыми точками объектов (центрами кругов) вычисляем по теореме Пифагора с помощью функции hypot и записываем в локальную переменную distance. Функция hasCollisionWith возвращает истинное значение, если круги пересеклись, т. е. расстояние между центрами кругов меньше, чем сумма их радиусов.

После этих изменений в классах **COcean** и **COceanObject** программу можно запустить на выполнение. К сожалению, никакой реакции на столкновения объектов пока нет. Но это не значит, что проверка не заработала. Дело в том, что мы нигде не переопределили пустой метод collideWith базового класса, поэтому при столкновениях объектов ничего не происходит.

Изменение других классов

Для того чтобы объекты реагировали на столкновение, нужно переопределить виртуальный метод collideWith в классах-наследниках: **CStone**, **CGrass**, **CFish**, **CHunter**. Рассмотрим подробно только одно взаимодействие: при столкновении хищника (**CHunter**) и рыбы (**CFish**) хищник съедает рыбу, и его размер (радиус) увеличивается. А рыба, как ни печально, погибает.

Добавим в классы **CHunter** и **CFish** метод collideWith, переопределив метод базового класса с тем же именем:

```
class CFish: public CMovingObject
  {
  public:
  ...
    virtual void collideWith(
            COceanObject* pOther ) override;
  };
class CHunter: public CMovingObject
  {
  public:
  ...
    virtual void collideWith(
            COceanObject* pOther ) override;
  };
```

При кодировании этих методов возникает следующая задача: мы получили ссылку на другой объект (параметр pOther), но это ссылка на объект базового класса **COceanObject**. Этой информации мало для обработки столкновения. Мы договорились, что пока учитываем только взаимодействие хищника и рыбы, значит, нужно как-то определить, что хищник столкнулся именно с рыбой, а рыба — с хищником.

Для решения этой задачи будем использовать информацию о типах данных, доступную во время выполнения программы. По-английски она называется *RTTI*: *Runtime Type Information*. С её помощью можно узнать, принадлежит ли переданный адрес объекту класса **CFish** или **CHunter**.

Рассмотрим метод collideWith для хищника. Параметр pOther указывает на объект, с которым столкнулся хищник. Попробуем преобразовать его в указатель на объект класса **CFish**. Если это удастся, то второй объект — рыба, а если нет — не рыба.

```
void CHunter::collideWith( COceanObject* pOther )
  {
  CFish* pFish = dynamic_cast <CFish*> (pOther);    // (1)
  if( pFish ) r += 1;                               // (2)
  }
```

В строке 1 создаётся новая локальная переменная pFish — указатель на объект класса **CFish**. В неё сразу же записывается начальное значение. Сразу отметим, что можно было попросить компилятор автоматически определить тип этой переменной:

```
auto pFish = dynamic_cast <CFish*> (pOther);
```

Давайте разберём это сложное выражение по частям. Запись dynamic_cast означает «динамическое преобразование типа», т. е. преобразование во время выполнения программы. Такое преобразование применяют тогда[1], когда при компиляции тип переменной ещё неизвестен. Действительно, ведь мы не знаем, на объект какого класса-наследника **COceanObject** будет ссылаться переданный указатель.

В угловых скобках записан тип (**CFish***), к которому нужно преобразовать переменную, а далее в круглых скобках — имя самой этой переменной (pOther).

Если преобразование удалось, в переменную pFish будет записан адрес, с которым можно работать как с адресом объекта **CFish**. Если же объект, на который указывает pOther, — не рыба, в переменную pFish будет записан нулевой указатель.

В строке 2 мы проверяем значение переменной pFish: если оно ненулевое, то хищник удачно встретился с рыбой и его радиус увеличивается.

Аналогично построим метод collideWith для класса **CFish**:

```
void CFish::collideWith( COceanObject* pOther )
  {
  auto pHunter = dynamic_cast <CHunter*> (pOther);
  if( pHunter ) r = 0;
  }
```

Если рыба встретила хищника, её ждёт печальная участь: радиус становится равным нулю, так что она считается мёртвой и будет удалена из памяти объектом *Океан* при выполнении метода removeDead.

Методы collideWith не зря вынесены из классов, хотя они очень короткие. Дело в том, что оба метода используют как класс **CFish**, так и класс **CHunter**. Поэтому описания обоих классов должно быть выше кода методов collideWith.

---

[1] Если тип переменной, значение которой преобразуется, заранее известен, используют статическое преобразование (static_cast).

Используя этот подход, вы можете самостоятельно добавить в программу остальные варианты взаимодействия объектов разных классов.

Обратите внимание, что объект *Океан* ничего «не знает» о том, как взаимодействуют объекты разных классов, всё определяется методами `collideWith` самих объектов. Океан просто организует такое взаимодействие, вызывая для всех пар объектов методы `hasCollisionWith` и `collideWith` на каждом интервале моделирования.

«Умные» указатели

Неаккуратное использование указателей в программе может привести к очень тяжёлым ошибкам, связанным с выделением и освобождением памяти. К ним относятся:

- обращение по неверному адресу;
- повторное удаление объекта;
- утечка памяти из-за того, что объект не удалён из памяти, а ссылка на него потеряна.

Для того чтобы избавить программистов от этих проблем, в стандарт C++11 были введены «умные» указатели. «Умный» указатель типа **shared_ptr** (от англ. *shared pointer* — разделяемый указатель) в любой момент «знает», сколько всего ссылок есть на связанный с ним объект. Когда такой указатель выходит из области видимости и удаляется из памяти, счётчик ссылок на этот объект уменьшается на единицу. Если после этого счётчик ссылок станет равен нулю, объект становится недоступным, поэтому автоматически удаляется из памяти. Это значит, что программисту не нужно вручную удалять объект, на который ссылается «умный» указатель — объект будет удалён именно тогда, когда станет не нужен.

Для работы с «умными» указателями необходимо подключить библиотеку `memory` (в переводе с английского — память):

```
#include <memory>
```

«Умный» указатель — это переменная типа **shared_ptr**:

```
shared_ptr<COceanObject> pObj;
```

В угловых скобках записан тип объектов, на которые может ссылаться этот указатель. Пока этот указатель пустой, он имеет нулевое значение и нулевой счётчик ссылок.

Значение, которое записывается в «умный» указатель, обычно строится с помощью функции `make_shared`:

```
pObj = make_shared<CStone>( 100, 100, 8 );
```

Здесь создаётся новый объект класса `CStone`, для этого вызывается конструктор **CStone**`(100, 100, 8)`. Затем адрес этого объекта записывается в «умный» указатель `pObj`.

Можно сразу объявить «умный» указатель и присвоить ему начальное значение:

```
auto pObj = make_shared<CStone>( 100, 100, 8 );
```

Служебное слово **auto** говорит о том, что тип переменной pObj определяется автоматически, по тому значению, которое ей присваивается.

Метод use_count позволяет узнать значение счётчика, т. е. количество ссылок на объект:

```
cout << pObj.use_count() << endl;
```

Для нашего примера программа выведет число 1.

Теперь создадим второй «умный» указатель и присвоим ему значение первого:

```
auto pObj2 = pObj;
```

Если теперь вывести на экран значения счётчиков обоих указателей, оба они будут равны двум. Это значит, что в программе есть два указателя на данный объект.

Если один из указателей, pObj или pObj2, выйдет из области видимости, счётчик ссылок другого уменьшится на единицу и станет равен 1. Когда удаляется второй указатель, счётчик ссылок становится равен нулю, и после этого объект автоматически удаляется из памяти.

Применяем «умные» указатели

Сделаем так, чтобы в нашей программе моделирования океана использовались только «умные» указатели. В первую очередь вектор pObjects в классе **COcean** должен теперь состоять из «умных» указателей на объекты-наследники класса **COceanObject**:

```
class COcean
  {
  private:
    std::vector<std::shared_ptr<COceanObject>> pObjects;
  ...
  };
```

В методе addObject тоже нужно перейти на «умные» указатели, которые строятся с помощью функции make_shared:

```
void COcean::addObject( objectType type,
                        int x0, int y0, int r0,
                        int v0, double course0 )
  {
  shared_ptr<COceanObject> pNewObj = nullptr;
  if( type == STONE )
    pNewObj = make_shared<CStone>( x0, y0, r0 );
```

```
  else if( type == GRASS )
    pNewObj = make_shared<CGrass>( x0, y0, r0 );
  else if( type == FISH )
    pNewObj = make_shared<CFish>( x0, y0, r0,
                                  v0, course0 );
  else /* if( type == HUNTER ) */
    pNewObj = make_shared<CHunter>( x0, y0, r0, v0 );
  if( pNewObj )
    pObjects.push_back( pNewObj );
  }
```

Поскольку объекты, связанные с «умными» указателями, удаляются из памяти автоматически, деструктор в классе **COcean** уже не нужен, его можно удалить.

Указатели в нашей программе используются при обработке столкновений. В методах `hasCollisionWith` и `collideWith` базового класса **COceanObject** и его наследников нужно изменить тип параметров на «умные указатели»:

```
class COceanObject
  {
    ...
    bool hasCollisionWith(
       std::shared_ptr<COceanObject> pOther )
       const;
    virtual void collideWith(
       std::shared_ptr<COceanObject> pOther ) {}
  };
```

Тело метода `collideWith` в каждом классе тоже поменяется, например, для класса **CFish**:

```
void CFish::collideWith(
            shared_ptr<COceanObject> pOther )
  {
  auto pHunter =
       dynamic_pointer_cast<CHunter>  (pOther);
  if( pHunter ) r = 0;
  }
```

Обратите внимание, что для преобразования типа «умного» указателя во время выполнения программы используется функция `dynamic_pointer_cast`, а не `dynamic_cast`, как для обычных указателей.

Заметно, что текст программы стал немного сложнее по сравнению с предыдущим вариантом. Однако мы получили важное преимущество: теперь нам не нужно думать о том, как мы будем освобождать выделенную память. Всё произойдёт автоматически и в нужный момент.

Выводы

- Для того чтобы обрабатывать столкновения, каждый объект должен иметь два метода. Один определяет факт столкновения с другим объектом, а второй изменяет объект в результате столкновения (если нужно).
- Вызов этих методов выполняет главный объект («игровой мир», у нас — океан), перебирая во вложенном цикле все возможные пары разных объектов.
- Удалением «мёртвых» объектов с поля занимается главный объект. Все классы должны иметь метод, позволяющий определить, можно ли удалить с поля данный объект.
- Для определения класса объекта по его адресу используется информация о типах данных, доступная во время выполнения программы (*RTTI*: *Runtime Type Information*).
- Для приведения указателя к другому типу во время выполнения программы применяется оператор `dynamic_cast`. Если такое преобразование закончится неудачно, в указатель будет записано нулевое значение.
- В стандарте C++11 введены «умные» указатели, которые поддерживают счётчик ссылок на объект, выделенный в памяти. Когда счётчик ссылок на какой-то объект становится равен нулю, этот объект удаляется из памяти автоматически.

Вопросы и задания

1. Предложите какие-нибудь дополнительные свойства и методы, которые могут быть у объекта *Океан* (можно вспомнить, как меняется состояние реального океана). В этом и следующих заданиях используются объекты, приведённые в параграфе.
2. Пётр хочет сделать так, чтобы всеми взаимодействиями объектов управлял главный объект — «игровой мир». Обсудите достоинства и недостатки такого решения.
3. Константин написал циклы в методе `checkCollisions` класса **COcean** так:
```
for( int i = 0; i < pObjects.size(); i++ )
  for( int j = 0; j < pObjects.size(); j++ )
    ...
```
Обсудите достоинства и недостатки этого варианта.
4. Почему метод `hasCollisionWith` не объявлен виртуальным? В каком случае его нужно будет объявить виртуальным?
5. Мария забыла написать строку
```
delete pObjects[i];
```
в теле метода `removeDead` класса **COcean**. К каким последствиям это может привести?

6. Даниил забыл написать строку
```
pObjects.erase( pObjects.begin()+i );
```
в теле метода `removeDead` класса **COcean**. К каким последствиям это может привести?

7. Почему, на ваш взгляд, в методах `checkCollisions` и `removeDead` мы не использовали циклы такого вида?
```
for( auto p: pObjects ) { ... }
```

8. Можно ли в процедуре `removeDead` поменять местами следующие строки?
```
delete pObjects[i];
pObjects.erase( pObjects.begin()+i );
```
Что при этом может произойти?

9. Используя дополнительные источники, выясните, какие ещё типы «умных» указателей введены в стандарте C++11.

10. Переделайте программу из параграфа так, чтобы она везде использовала «умные» указатели вместо обычных.

*11. *Проект.* Добавьте в модель океана другие взаимодействия между объектами:
- рыбы едят траву;
- если хищник сталкивается с камнем, его размеры уменьшаются.

*12. *Проект.* Вместо хищника, который следит за мышью, добавьте в модель океана несколько хищников, которые двигаются в случайных направлениях.

*13. *Проект.* Добавьте в модель океана изменения объектов с течением времени:
- трава растёт (увеличивается в размерах);
- рыбы размножаются (появляются новые рыбы).

**14. *Проект.* Добавьте в модель океана элементы искусственного интеллекта:
- рыба, обнаружив траву в некоторой зоне рядом с ней, начинает двигаться к ближайшей траве;
- почуяв рядом хищника, рыбы «убегают» от него;
- хищник, обнаружив рыб в некоторой зоне рядом с ним, начинает двигаться к ближайшей рыбе; если рядом рыб нет, хищник двигается в случайном направлении (режим поиска).

Интересные сайты

gamesmaker.ru — сайт о программировании игр на C++

tutorialspoint.com — онлайн-учебники по программированию

§ 18
Простая программа на C#

Ключевые слова:

- RAD-среда
- среда .NET
- язык C#
- виртуальная машина
- проект
- форма
- свойство
- событие
- обработчик события
- компонент
- статический класс

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, правильно подключить обработчики сообщений. Значительную часть своего времени программист занимался рутинной работой, которая была слабо 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 (с помощью мыши), а человек думал бы о сути решаемой задачи, т. е. 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 с помощью мыши *элементов интерфейса* (полей ввода, кнопок, списков) и настройка их свойств;
- создание *обработчиков событий*;
- программирование 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 расчётов 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. Существует свободная RAD-среда *Lazarus* (**lazarus.freepascal.org**), которая во многом аналогична *Delphi*, но позволяет создавать *кроссплатформенные* программы (работающие в разных операционных системах: Windows, Linux, macOS). Как правило, в RAD-средах используется объектно-ориентированный стиль программирования.

Самая известная современная профессиональная RAD-система — *Microsoft Visual Studio* — поддерживает несколько языков программирования. Одна из её версий, *Visual Studio Community*, распространяется бесплатно.

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

Язык C# и среда .NET

В этом и следующих параграфах мы познакомимся с программированием в RAD-средах на примере бесплатной среды *Visual Studio Community*[1)] для операционной системы Windows. Все основные приёмы, которые вы изучите, применимы также и в других средах визуальной разработки программ, например в *Delphi* и *Lazarus*.

Для выполнения примеров мы будем использовать язык C# (читается как «Си шарп»), разработанный компанией Microsoft. Он опирается на многие идеи языков C и C++. Поэтому правила записи команд в языках C# и C++ очень похожи.

С помощью языка C# создаются программы для платформы .NET (читается как «дот нет», от англ. *dot* — точка), разработанной компанией Microsoft для операционной системы Windows. Программы на C# компилируются не в команды процессора, а в код на специальном языке *CIL* (*Common Intermediate Language* — «общий промежуточный язык»). Этот код выполняет специальная программа — *виртуальная машина* среды .NET, которая называется *CLR* (*Common Language Runtime* — общая среда выполнения для языков).

В код на языке CIL можно скомпилировать программы на языках C++, C#, Visual Basic .NET, F#, PascalABC.NET и др. Это позволяет объединять в одной программе модули, написанные на разных языках.

Однако нужно помнить, что для выполнения таких программ необходимо установить среду .NET (англ. *.NET Framework*), причём опре-

[1)] Добавка *Community* (в переводе с английского — сообщество) указывает на бесплатность этой версии. Скачать среду можно на сайте **visualstudio.microsoft.com/ru/vs/community/**.

делённой версии, для которой написана программа. Установочные файлы всех версий среды .NET для Windows можно свободно загрузить с сайта компании Microsoft.

В UNIX-подобных операционных системах (Linux, macOS) программы, использующие .NET, могут выполняться в среде *Mono*.

Проект в C#

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

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

Проекты в *Visual Studio* объединяются в *решения* (англ. *solution*). 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 на C# находится в файле `Program.cs` (здесь и далее указываются имена файлов 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 из них хранятся данные о расположении и свойствах элементов интерфейса. В названиях этих файлов есть слово `Designer`, например `Form1.Designer.cs`. Второй файл (`Form1.cs`) содержит программный код обработчиков событий, связанных с этой формой.

Одна из форм — главная, она появляется на экране при запуске программы. Когда пользователь закрывает главную форму, работа программы завершается.

Первый проект

Для создания проекта с графическим интерфейсом нужно выбрать пункт меню *Файл — Создать проект* и в появившемся окне отметить вариант *Приложение Windows Forms*. При этом автоматически создается работоспособная программа, которую сразу же можно запустить на выполнение клавишей **F5** (рис. 2.6). Все приведённые далее снимки экрана («скриншоты») были сделаны в среде *Visual Studio Community 2017*.

Рис. 2.6

В верхней части окна *Visual Studio* (см. рис. 2.6) расположены меню и панель инструментов — кнопки для быстрого вызова команд. В рабочей области размещаются несколько окон, их состав может меняться. Мы будем использовать следующие окна:

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

На *Панели элементов* размещены готовые объекты (кнопки, поля ввода, списки и т. п.), которые можно добавлять на формы с помощью мыши.

Обозреватель решений позволяет выбрать для редактирования любой файл проекта (решения).

Вся программа на C# состоит из объектов. Их свойства настраиваются с помощью окна *Свойства*. В нём две основные вкладки:

-  *Свойства*, где можно изменить общедоступные свойства объекта,
-  *События*, где из списка подходящих методов выбираются обработчики событий, связанных с объектом.

Форма — это тоже объект в программе. На форме можно располагать другие объекты, обеспечивающие интерфейс с пользователем, — кнопки, поля ввода, метки и т. п. Они называются *компонентами* или *элементами*.

После двойного щелчка мышью по названию файла `Form1.cs` в окне *Обозреватель решений* открывается *Конструктор формы*. В этом режиме можно мышью перетаскивать компоненты с *Панели элементов* на форму, изменять их размеры и расположение, настраивать свойства. 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[1]:

```
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 WindowsFormsApp1 {
  public partial class Form1: Form {
```

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

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

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

```
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 `WindowsFormsApp1` (такое имя имеет проект по умолчанию):

```
namespace WindowsFormsApp1 {
  ...
  }
```

В этом пространстве имён 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(); }
```

Процедура `InitializeComponent` устанавливает свойства формы и расположенных на ней элементов. Она находится во втором файле, связанном с формой, — `Form1.Designer.cs`, который строится автоматически. Менять его вручную не рекомендуется, но посмотреть интересно и полезно.

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

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;

namespace WindowsFormsApp1 {
 static class Program {
```

```
    static void Main() {
      Application.EnableVisualStyles();
      Application.SetCompatibleTextRenderingDefault( false );
      Application.Run( new Form1() );
    }
  }
}
```

В уже знакомом нам пространстве имён проекта `WindowsFormsApp1` 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 (нет *состояния*). Таким образом, статический класс — это просто набор функций.

Все методы статического класса также объявляются статическими. В нашем случае — это единственный метод с именем `Main` — точка входа, с которой начинается выполнение основной программы.

В методе `Main` выполняются начальные установки для объекта (статического класса) **Application**. Это объект-приложение, самый главный объект в нашей программе. В последней строке вызывается метод `Run`, запускающий цикл обработки сообщений. Этому методу передаётся ссылка на главную форму, которая появляется на экране при запуске программы[1]. В данном случае главная форма — это новая форма класса **Form1**, созданная с помощью оператора **new**.

Свойства объектов

Панель *Свойства* позволяет просматривать и изменять свойства и обработчики событий для выделенного объекта, например для формы (рис. 2.7).

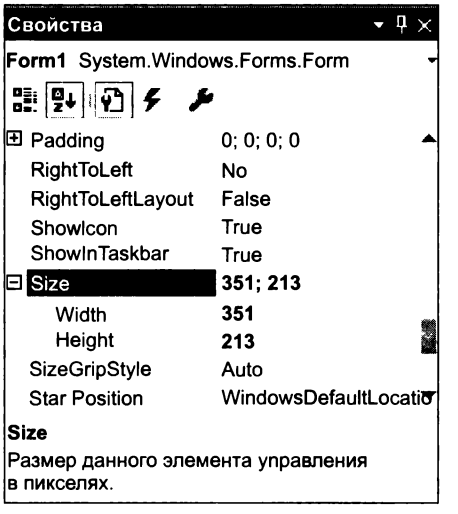


Рис. 2.7

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

Можно заметить, что при изменении границ формы с помощью мыши изменяются её свойства Width (в переводе с английского — *ширина*) и Height (*высота*), объединённые в группу Size (*размер*). Эти величины можно изменить и с клавиатуры, вводя новые значения в соответствующие области на панели *Свойства*. При этом размеры формы также меняются.

Свойство Name (в переводе с английского — *имя*) — это название объекта-формы в программе. В имени можно использовать только латинские буквы, цифры и знаки подчёркивания.

Если изменить имя формы в окне *Свойства*, скажем, на MainForm, то это название автоматически изменится и в тексте модуля этой формы. Более того, название класса в файле Form1.Designer.cs тоже сразу изменится на MainForm. Таким образом, многие изменения вносятся в код программы автоматически.

Перечислим ещё некоторые важные свойства формы, с которыми интересно поработать:

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

Установленные вами свойства формы **Form1** автоматически сохраняются в файле Form1.Designer.cs. Там находится код метода InitializeComponent, который вызывается при запуске программы и выполняет все начальные установки. Например, в нём можно найти такие строки:

```
this.ClientSize = new System.Drawing.Size(800, 600);
this.Name = "MainForm";
this.Text = "Первая программа";
```

Слово this (в переводе с английского — *этот*) — это ссылка на саму форму. Все свойства, записанные через точку после this, — это свойства формы. В первой строке задаются размеры формы (свойство ClientSize), во второй — имя формы в программе (свойство Name), в третьей — текст в заголовке окна (свойство Text).

Обработчики событий

На вкладке  *События* в окне *Свойства* (рис. 2.8) перечислены все события, которые может обрабатывать форма.

Чтобы создать обработчик, нужно дважды щёлкнуть мышью на поле справа от названия события. При этом открывается окно редактора и в текст модуля автоматически добавляется пустой обработчик события (шаблон), в который остаётся только вставить нужные команды.

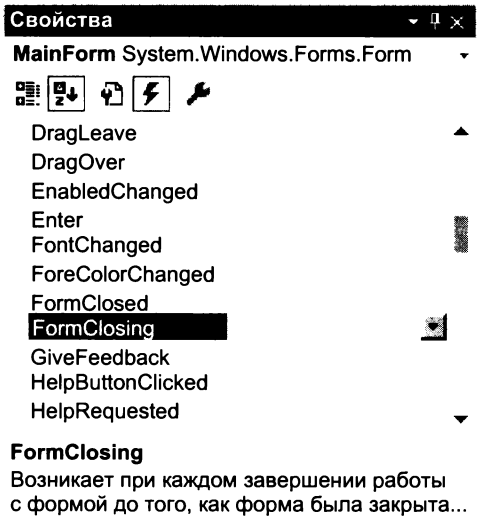


Рис. 2.8

Рассмотрим простой пример. Вы знаете, что многие программы запрашивают подтверждение, когда пользователь завершает их работу. При этом происходит событие, которое называется *Закрытие формы* (`FormClosing`). В обработчике этого события мы можем вывести на экран сообщение для пользователя, получить его ответ и, если нужно, отменить закрытие окна.

Для создания обработчика выполним двойной щелчок мышью на поле справа от названия события `FormClosing`. В результате в файле `Form1.cs` создаётся шаблон обработчика:

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

}
```

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

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

- `sender` — ссылка на объект, от которого пришло сообщение о событии (в данном случае такой объект — это сама форма);
- `e` — изменяемый объект, с помощью которого можно разрешить или запретить закрытие формы.

Установив значение поля `e.Cancel` равным `true` («да»), мы отменяем команду на закрытие формы. Значение `false`, которое записано по умолчанию, разрешает закрыть форму и завершить работу программы.

Для того чтобы решить вопрос о закрытии окна, нужно задать пользователю вопрос и получить на него ответ. В библиотеке C# есть класс **MessageBox**, который выводит на экран диалоговое окно с несколькими кнопками (рис. 2.9) и возвращает ответ пользователя — код нажатой кнопки.

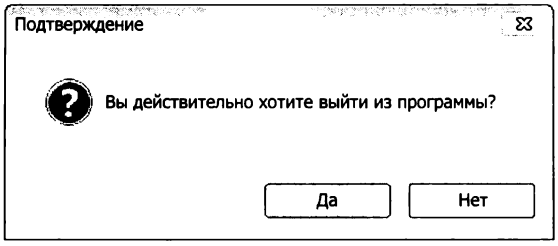


Рис. 2.9

В тело обработчика добавим условный оператор, который присвоит свойству `e.Cancel` значение `true`, если пользователь подтвердит выход из программы:

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

Здесь вызывается метод `MessageBox.Show`, и его результат записывается в переменную `result` типа **DialogResult**. Если это значение совпадает с константой `DialogResult.No` (т. е. пользователь нажал кнопку «Нет»), в свойство изменяемого параметра `e.Cancel` записывается значение `true`. В этом случае команда закрытия окна отменяется, иначе (при подтверждении выхода) программа завершается.

Разберём подробно вызов метода `MessageBox.Show`. Ему передаются несколько аргументов:

- сообщение пользователю (`"Вы действительно хотите выйти из программы? "`);
- заголовок окна (`"Подтверждение"`);
- набор 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 работоспособную программу с графическим интерфейсом на C# и познакомились со средой *Visual Studio*. В следующем параграфе вы узнаете, как работать с компонентами.

Выводы

- Для быстрой разработки программ применяют системы визуального программирования, в которых интерфейс строится без «ручного» написания программного кода. Как правило, в них используется объектно-ориентированный стиль программирования.
- Программы на языке C# компилируются не в коды команд процессора, а в программы на промежуточном языке CIL. Программу на языке CIL выполняет виртуальная машина среды .NET.
- Проект — это набор файлов, которые используются для сборки исполняемой программы (файла с расширением .exe).
- В программе с графическим интерфейсом может быть несколько окон, которые называют формами. С каждой формой связаны два файла, один из которых (`.Designer.cs`) строится автоматически.
- Готовые объекты, которые можно использовать в программе (в том числе кнопки, поля ввода, надписи и т. п.), называются компонентами или элементами.
- Интерфейс программы на C# можно строить с помощью мыши.
- В окне *Свойства* можно изменить свойства выделенного объекта и назначить обработчики связанных с объектом событий.

Вопросы и задания

1. Какие причины сделали необходимым создание сред быстрой разработки программ? В чём достоинства этих сред?
2. Чем отличается разработка программ в RAD-среде от использования классических сред программирования?
3. Какие достоинства и недостатки, на ваш взгляд, имеют программы, написанные для среды .NET?
4. В каком смысле используется термин «проект» в программировании?
5. Из каких файлов состоит проект на языке C#?
6. Сравните термины «проект» и «решение» (*solution*).
7. Что такое форма? Почему для описания формы в *Visual Studio* используются два файла?

8. Покажите, что программа, написанная с помощью *Visual Studio*, состоит из объектов.
9. Где расположена основная программа (*точка входа*) в проекте *Visual Studio* на языке C#?
10. Почему в основной программе не виден цикл обработки сообщений?
11. Назовите некоторые важнейшие свойства формы. Какими способами можно их изменять?
12. Приведите примеры автоматического построения и изменения кода в RAD-среде.
13. Как создать новый обработчик события? Проверьте, можно ли ввести текст обработчика вручную.
14. Что означает параметр `sender`, который передаётся в обработчик события? В каких случаях он может понадобиться?
15. Предположите, как можно вывести сообщение об ошибке на экран. Проверьте вашу догадку экспериментально.
16. Попробуйте изменить какие-нибудь свойства формы, построив обработчик ещё одного события (например: `Shown` — вывод формы на экран; `Click` — щелчок мышью; `Resize` — изменение размеров).
17. Выясните, какие изменения вносятся автоматически в файл `.Designer.cs` при добавлении обработчика события.

Интересные сайты

visualstudio.com/ru/vs/community/ — бесплатная среда разработки *Visual Studio Community*

docs.microsoft.com/ru-ru/dotnet/csharp/ — руководство по языку C# от компании Microsoft

bit.ly/2HnSmJo — учебный курс по языку C# от компании Microsoft

§ 19
Использование компонентов

Ключевые слова:

- компонент
- свойство
- событие
- обработчик события
- родительский объект
- дочерний объект
- поле ввода
- метка
- статический метод
- обработка ошибок
- исключение

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

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

Программа для просмотра рисунков

Используя готовые компоненты, мы построим простую программу для просмотра рисунков.

В верхней части окна программы разместим кнопку для загрузки файла и флажок, который изменяет масштаб рисунка так, чтобы рисунок вписывался в окно и был полностью виден (рис. 2.10 и см. цветной рисунок на обороте обложки).

Рис. 2.10

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 кнопку **Panel** на форму или щёлкнуть по этой кнопке и нарисовать на форме прямоугольник, ограничивающий панель. Для изменения размеров панели перетащите маркеры на её границах. Можно также изменить значения свойств Width и Height на панели *Свойства*.

Панель можно перетаскивать по форме мышью за значок . Нам нужно, чтобы панель была всё время прижата к верхней границе окна, а её ширина должна быть равна ширине окна. Для того чтобы обеспечить такое поведение, установим свойство Dock (по-английски — *состыковать*) равным Top (*верх*).

На панели нужно разместить кнопку (компонент Button) и флажок (компонент **CheckBox**) — рис 2.11.

На кнопке должна быть надпись Открыть файл (свойство Text), а справа от флажка — текст По размерам окна (это свойство компонента **CheckBox** тоже называется Text).

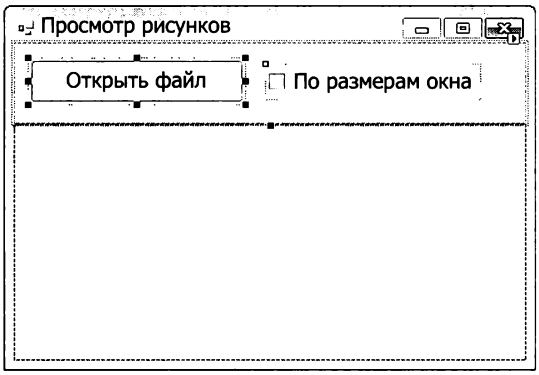


Рис. 2.11

Размеры и расположение компонентов изменяются с помощью мыши. Имена компонентов (свойства Name) тоже можно заменить на более понятные, 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 в методе InitalizeComponent они создаются:

```
this.panel1 = new System.Windows.Forms.Panel();
this.openBtn = new System.Windows.Forms.Button();
this.sizeCb = new System.Windows.Forms.CheckBox();
```

Слово this в методе формы — это ссылка на саму форму, а запись this.panel1 говорит о том, что панель (как и другие компоненты) принадлежит форме.

Компоненты на форме связаны отношениями «родитель — потомок». Главный объект — это форма MainForm. Она является *родительским объектом* для панели panel1, а панель — *дочерним компонентом* для формы. Это означает, что при перемещении формы панель перемещается вместе с ней. Кроме того, «родитель» отвечает за прорисовку всех дочерних компонентов на экране. В свою очередь, панель — это родительский объект для кнопки openBtn и флажка sizeCb.

Теперь добавим на форму специальный компонент **PictureBox**, который «умеет» выводить рисунки различных форматов. Для того чтобы он заполнял всё свободное пространство формы (кроме панели), для свойства Dock выберем значение Fill (по-английски — заполнить).

Изменим название компонента **PictureBox** на `img`. Для него, как и для панели `panel1`, родительский объект — это сама форма **MainForm**. Иерархия связей «родитель — потомок» для нашей формы показана на рис. 2.12.

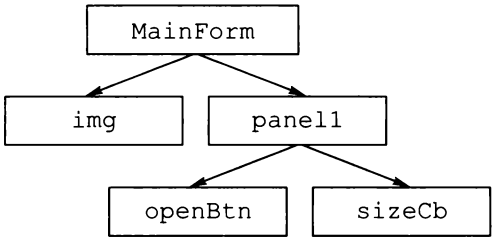


Рис. 2.12

Остаётся запрограммировать два действия:

- выбор файла и его загрузку в компонент `img`;
- подгонку размера рисунка под размер формы (при включённом флажке `openCb`).

К счастью, для этого достаточно использовать готовые возможности компонентов библиотеки языка C#.

В группе *Диалоговые окна* есть компонент для выбора файла, он называется **OpenFileDialog**. Перетащим его в любое место формы. Это невизуальный компонент — его не будет видно во время выполнения программы. Поэтому он добавляется не на форму, а в нижнюю часть окна конструктора.

Для краткости изменим название компонента на `openDlg` и очистим свойство `FileName` (имя файла по умолчанию). В свойство `Filter` запишем такую строку:

```
"Рисунки|*.jpg;*.jpeg;*.gif;*.png;*.bmp"
```

Это фильтр для отбора файлов: файлы, не подходящие ни под одну из масок, будут скрыты. В данном случае нас интересуют только файлы с указанными расширениями. Слева от символа «|» записано название этого фильтра (любой текст).

Выбор файла нужно выполнять тогда, когда пользователь щёлкнет по кнопке `openBtn`. Щелчок по кнопке — это событие, которое называется `Click`. Для того чтобы добавить обработчик этого события, нужно выделить кнопку, перейти на вкладку *События* в окне *Свойства* и выполнить двойной щелчок в поле справа от названия события `Click`. После этого откроется окно с шаблоном обработчика, который был создан автоматически. Тело процедуры-обработчика пока пустое. Запишем в него команды

```
if( openDlg.ShowDialog() == DialogResult.OK )
  img.Image = new Bitmap( openDlg.FileName );
```

В этих двух строках происходит довольно много операций, которые скрыты за вызовами готовых методов:

1) вызывается метод ShowDialog — пользователь увидит стандартный диалог выбора файла;
2) если результат работы метода ShowDialog равен значению, обозначенному как DialogResult.OK, это означает, что пользователь успешно выбрал файл; при этом имя выбранного файла можно получить через свойство FileName объекта-диалога;
3) в памяти создаётся новый объект класса Bitmap (по-английски *bitmap* — битовая карта, рисунок); в него загружается изображение из выбранного файла (openDlg.FileName);
4) адрес нового объекта присваивается свойству Image компонента img; при этом рисунок сразу загружается в компонент, и изображение на форме обновляется.

Сейчас уже можно запустить программу и проверить, как работает загрузка файлов.

Обратите внимание, что сейчас изображение выводится с исходными размерами независимо от размера окна. Флажок *По размерам окна* можно включать и выключать, но на работу программы он никак не влияет.

Изменить масштаб изображения можно с помощью свойства SizeMode (режим подгонки размера), которое есть у всех компонентов класса **PictureBox**, в том числе и у нашего компонента img. Если этому свойству присвоить значение Zoom, компонент сам выполнит подгонку размеров рисунка под размер области, которую он занимает. По умолчанию установлен режим Normal — не изменять размеры.

Переключение режима вывода рисунка будем выполнять при изменении состояния флажка sizeCb. У него есть свойство Checked (в переводе с английского — отмечен) — логическое значение, определяющее состояние компонента. Если флажок включён, это свойство равно true, если выключен, то false.

При изменении свойства Checked происходит событие CheckedChanged (в переводе с английского — изменение свойства Checked). В обработчике этого события нужно устанавливать режим вывода рисунка в зависимости от состояния флажка. Создадим шаблон обработчика (так же, как мы делали раньше) и добавим в него следующий код:

```
if( sizeCb.Checked )
     img.SizeMode = PictureBoxSizeMode.Zoom;
else img.SizeMode = PictureBoxSizeMode.Normal;
```

Режимы Zoom и Normal относятся к стандартному перечисляемому типу **PictureBoxSizeMode**.

Теперь программа полностью готова. Отметим следующие важные особенности:

- программа целиком состоит из объектов и основана на принципах объектно-ориентированного программирования;
- мы построили полезную 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. Обычно для ввода данных применяют поля ввода — компоненты класса **TextBox**. Введённую символьную строку можно получить через свойство `Text` (по-английски — текст).

Шрифт текста в поле ввода задаётся сложным свойством `Font` (шрифт). Это объект, у которого есть свои «подсвойства», их список можно увидеть, если щёлкнуть по значку ⊞ слева от названия свойства. Например, подсвойство `Size` — это размер шрифта в пунктах, а логические значения `Bold` (жирный), `Italic` (курсив) и `Underline` (подчёркнутый) определяют стиль шрифта: каждое из них может быть истинно или ложно.

Если вручную установить шрифт для какого-то объекта, например для формы, все дочерние компоненты по умолчанию будут иметь такой же шрифт.

Программа, которую мы напишем, будет строить шестнадцатеричный код цвета, используемый в языке HTML, по значениям красной, зелёной и синей составляющих цвета в модели RGB (см. § 9) — рис. 2.12.

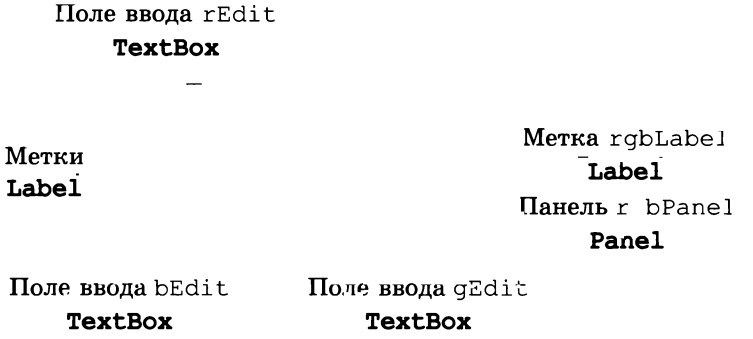


Рис. 2.13

На форме расположены:

- три поля ввода (в них пользователь может задать значения составляющих цвета в модели RGB);
- прямоугольная панель (компонент **Panel**), цвет которой изменяется согласно введённым значениям;
- несколько меток (компонентов **Label**).

Метки — это надписи, которые пользователь не может редактировать. Однако из программы можно изменять их содержимое через свойство `Text`.

Для того чтобы пользователь не мог изменить размеры окна программы (в данной задаче это не нужно!), мы установим свойство формы `FormBorderStyle` (по-английски — стиль границы формы), равное `FixedSingle` (фиксированная, одиночная).

Во время работы программы будут использоваться поля ввода rEdit, gEdit и bEdit, метка rgbLabel для вывода кода цвета, и панель rgbPanel. Начальные значения полей ввода — любые целые числа от 0 до 255 (их нужно записывать в свойство Text).

При изменении содержимого любого из трёх полей ввода будем выводить HTML-код полученного цвета в свойство Text метки rgbLabel, а также устанавливать новый цвет заливки панели rgbPanel.

Событие, которое происходит при изменении текста в поле ввода, называется TextChanged. Так как при изменении любого из трёх полей нужно выполнить одинаковые действия, для всех полей ввода мы установим один и тот же обработчик события TextChanged. Удерживая клавишу *Shift*, выделим щелчками мышью три поля ввода. После этого на вкладке *События* окна *Свойства* двойным щелчком мышью создадим обработчик события TextChanged. Этот обработчик 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. Это делает метод Parse (в переводе с английского — разбор строки) класса int (целые числа). Возможно, вы заметили, что обычно метод вызывается для какого-то объекта, а здесь слева от точки записано название типа данных (класса) int. Это особый метод, который называется *методом класса* или *статическим методом*.

У панели есть свойство BackColor, которое задаёт цвет заливки внутренней области. Нужный цвет определяется по значениям составляющих модели RGB с помощью метода FromArgb класса Color.

В последней строке обработчика события строится символьная строка, содержащая шестнадцатеричный код цвета. Для перевода значений в шестнадцатеричную систему применяем метод ToString (в переводе с английского — в строку), второй аргумент "X2" указывает на то, что число нужно записать как два знака в шестнадцатеричной системе счисления ("X").

Вы можете заметить, что сразу после запуска программы код цвета в метке rgbLabel и цвет панели rgbPanel не соответствуют значениям

составляющих цвета в полях ввода. Дело в том, что обновление метки и цвета панели не произойдёт, пока не будет вызван обработчик TextChanged.

Чтобы исправить ситуацию, нужно при запуске программы сымитировать изменение одного из полей ввода. Для этого мы используем событие Load, которое происходит сразу после создания формы, но до того, как она появится на экране. Выделим форму и создадим (так же, как мы делали раньше) обработчик события Load. В нём вызовем готовый обработчик TextChanged:

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

Первый аргумент, который передаётся любому обработчику, — это ссылка на объект-источник события. Здесь источником события «назначено» поле ввода rEdit, но вместо него можно было использовать ссылку на любой компонент или даже нулевой объект null. Поскольку значения параметров в нашем обработчике TextChanged не используются, можно передать ему два нулевых объекта:

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

Обработка ошибок

Если в предыдущей программе пользователь введёт не цифры, а другие символы (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 программе.

В современных языках программирования для обработки ошибок применяют *механизм исключений*. В языке C# все «опасные» участки кода (на которых может возникнуть ошибка) нужно поместить в блок **try — catch**:

```
try {
  // "опасные" команды
  }
catch {
  // обработка ошибки
  }
```

Слово *try* в переводе с английского означает «попытаться», *catch* — «поймать». В данном случае мы ловим *исключения* (исключительные или ошибочные, непредвиденные ситуации). Программа выполнит блок **catch** только тогда, когда между **try** и **catch** произойдёт ошибка.

В нашей программе «опасные» команды — это операторы преобразования данных из текста в числа (вызовы метода `int.Parse`). Вот улучшенный обработчик с защитой от неправильного ввода:

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

В случае ошибки мы выведем вместо кода цвета знак вопроса, а панель будет невидима — её цвет будет совпадать со стандартным цветом элементов управления операционной системы; этот цвет обозначается как `SystemColors.Control`.

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` (в переводе с английского — «нажатие клавиши»):

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

Обратите внимание, что в языке C# используются классические обозначения логических операций, пришедшие из языка C: операция И обозначается как `&&`, ИЛИ — как `||`, а НЕ –восклицательным знаком. Обозначения **and**, **or** и **not**, часто применяемые в языке C++, считаются синтаксической ошибкой.

Обработчик события `KeyPress` получает блок данных типа `KeyPressEventArgs` с именем `e`, в составе которого есть поле `KeyChar` — символ, соответствующий нажатой клавише.

При вводе целых неотрицательных чисел используются только цифры и клавиша `BackSpace` (код символа 8) для удаления ошибочно введённых символов. Принадлежность символа к цифрам определяется с помощью метода `Char.IsDigit`, который возвращает логическое значение `true` для любой цифры и `false` для всех остальных символов.

Если полученный символ не входит в допустимый набор, свойство `e.Handled` (по-английски — *обработано*) устанавливается равным `true`. Этим мы сообщаем системе, что событие уже обработано и никаких дополнительных действий не требуется. Так для всех «посторонних» символов отключается стандартная процедура обработки, при вводе они игнорируются.

Выводы

- Интерфейс программы строится с помощью мыши из готовых компонентов, которые размещены на *Панели элементов*.
- Свойства компонентов изменяются с помощью панели *Свойства*. На вкладке *События* можно установить обработчики событий для выделенного компонента.
- Родительский компонент отвечает за перемещение и вывод на экран всех дочерних компонентов.
- Невизуальные компоненты, например диалог выбора файла, не изображаются на форме.
- Поля ввода позволяют пользователю вводить данные.
- Метки — это надписи на форме, которые не может изменить пользователь. Текст метки можно изменять из программы.
- Для нескольких компонентов можно установить один и тот же обработчик события. Параметр `sender`, который получает каждый обработчик, — это ссылка на объект-источник события.
- Статический метод (метод класса) — это метод, для вызова которого не нужно создавать объект класса.
- Для обработки ошибок в современных программах используют механизм исключений.

Вопросы и задания

1. Что такое компоненты? Зачем они нужны?
2. Объясните, как связаны компоненты и идея инкапсуляции.
3. Что такое родительский объект? Что такое дочерний объект?
4. Объясните роль свойства `Dock` в размещении элементов на форме.
5. Назовите основное свойство флажка. Как его использовать?
6. Что такое стандартный диалог? Как его использовать?
7. Объясните, что такое сложное свойство (на примере свойства `Size` или `Font`).
8. Какой шрифт устанавливается для компонента по умолчанию?
9. Сравните свойства метки и поля ввода.
10. Как можно установить один обработчик события для нескольких компонентов? Как в этом обработчике выяснить, какой компонент был источником события?
11. Зачем в программе из параграфа используются методы `int.Parse` и `Color.FromArgb`?
12. Как обрабатываются ошибки в современных программах? В чём, на ваш взгляд, преимущества и недостатки такого подхода?

13. Как при вводе блокировать некорректные символы?
14. Используя дополнительные источники, выясните, какие ещё элементы входят в перечисляемый тип `DialogResult`.
15. Используя дополнительные источники, выясните, как преобразовать строку в вещественное число.
16. *Проект*. Измените программу для просмотра рисунков из параграфа так, чтобы кнопка и флажок размещались в нижней части окна.
17. *Проект*. Добавьте в программу для построения HTML-кода цвета (из параграфа) защиту от ввода слишком больших чисел (больших, чем 255).
18. Изучите метод `InitializeComponent` в файле `Form1.Designer.cs`. Выясните, где указано, что форма — это родительский объект для кнопки и флажка.

*19. *Проект*. Разработайте свою программу, использующую компоненты (например, можно написать программу для шифрования и дешифрования текста с помощью шифра простой замены).

Интересные сайты

ulearn.me — интерактивные онлайн-курсы по программированию на C#

professorweb.ru — программирование для среды .NET

metanit.com/sharp/ — сайт о программировании (в том числе на C#)

§ 20
Создание новых классов

Ключевые слова:

- модуль
- модель
- представление
- класс
- компонент
- статический класс
- методы класса
- свойство

В этом параграфе мы создадим две программы, в которых используются модули — файлы, содержащие функции и описания новых классов объектов.

Сначала построим программу, которая вычисляет арифметическое выражение, записанное в символьной строке (см. § 11).

Вычисление арифметических выражений: модель

Модель в этой задаче — это функции для вычисления арифметического выражения, записанного в символьной строке. Будем использовать те же функции, что и в § 11.

Основная функция calc принимает один параметр — символьную строку — и вычисляет значение записанного в ней арифметического выражения. Напомним алгоритм её работы (на псевдокоде):

```
pos = позиция символа, соответствующего
      последней операции
if ( pos < 0 )
  результат = перевести всю строку в число
else {
  n1 = результат вычисления левой части
  n2 = результат вычисления правой части
  результат = применить операцию к n1 и n2
  }
```

Для того чтобы найти последнюю выполняемую операцию, функция calc вызывает другую функцию, lastOp, а та, в свою очередь, использует функцию priority для определения приоритета операций. Четвёртая функция, doOperation, вызывается из calc: она выполняет заданную операцию с переданными ей данными. Все эти функции мы хотим поместить в отдельный файл (*модуль*).

Новый класс

Создадим новый проект StringCalculator. Для того чтобы выделить модель задачи в отдельный файл, нужно построить новый класс. Конечно, возникает вопрос: почему нельзя сделать отдельный модуль, содержащий только функции, не относящиеся к какому-то классу? Но это действительно сделать невозможно, потому что в языке C# любая функция обязательно должна быть методом какого-то класса.

Выберем в меню *Visual Studio* пункт *Проект — Добавить класс*, введём в диалоговом окне название класса — **Calculator**. После этого в окне *Обозреватель решений* в списке файлов появляется новый файл Calculator.cs (рис. 2.14).

Если открыть его двойным щелчком мышью, мы увидим заготовку, построенную программой автоматически:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace StringCalculator
{
  class Calculator
    {
    }
}
```

Рис. 2.14

Итак, в отдельном файле создан новый класс **Calculator**, этот файл добавлен к проекту, и он относится к тому же пространству имён (namespace) StringCalculator, что и главная форма программы.

Новый класс нужен нам только для того, чтобы добавлять в него функции, а не создавать объекты этого класса. Такой класс называется *статическим*, перед его объявлением нужно добавить слово **static**:

```
static class Calculator
{
}
```

Статический класс — это просто набор функций, объединённых под общим именем класса.

Методы класса

В класс **Calculator** мы поместим все функции, которые используются для вычисления выражения (см. § 11):

```
static class Calculator
  {
  public static int calc( string expr ) {
    ...
    }
  static int lastOp( string expr ) {
    ...
    }
```

```
  static int priority( char op )  {
    ...
    }
  static int doOperation(char op, int n1, int n2) {
    ...
    }
}
```

В этом объявлении интересны два момента. Во-первых, все функции объявлены с описателем **static**, поскольку все методы статического класса **Calculator** должны быть статическими, т. е. не привязанными к конкретному объекту. Статические методы называются *методами класса* (в отличие от методов объектов).

Во-вторых, для метода `calc` указан дополнительный описатель **public**. Такой метод будет доступен не только в классе **Calculator**, но и вне класса, например в методах главной формы. А остальные методы — закрытые (**private**), их могут вызывать только методы самого класса **Calculator**, в частности метод `calc`.

Напишем сначала код метода `calc`:

```
public static int calc( string expr )
  {
  int pos = lastOp( expr );                    // (1)
  if( pos < 0 )
    return int.Parse( expr );                  // (2)
  int n1 = calc( expr.Substring(0, pos) );     // (3)
  int n2 = calc( expr.Substring( pos+1 ) );    // (4)
  return doOperation( expr[pos], n1, n2 );     // (5)
  }
```

Обратите внимание, что функция `calc` — рекурсивная, она дважды вызывает сама себя (в строках 3 и 4).

В строке 1 вызывается функция `lastOp`, которая возвращает позицию последней выполняемой операции. Если эта функция вернула –1 («операция не найдена»), в строке 2 преобразуем всю символьную строку в число с помощью метода `Parse` и возвращаем это число как результат функции.

Строки 3–5 обрабатывают случай, когда операция найдена. В этом случае вычисляем значения выражений слева и справа от знака операции, а потом выполняем с этими значениями нужную операцию, вызвав функцию `doOperation`.

Для того чтобы выделить подстроку из строки, применяется метод `Substring` класса `string`. Первый аргумент этой функции — начальная позиция, второй — длина подстроки. Вызов

```
expr.Substring( 0, pos )
```

возвращает подстроку строки `expr`, которая начинается с символа `expr[0]` и имеет длину `pos`. Если второй аргумент не задан, метод

возвращает все символы с указанной позиции до конца строки. Например, подстрока, полученная в строке 4:

```
expr.Substring( pos+1 )
```

содержит все символы после найденного знака операции.

Остальные методы практически повторяют функции, написанные на языке Python в § 11:

```
static int lastOp( string expr )
  {
  int minPrt = 50, pos = -1;
  for( int i = 0; i < expr.Length; i++ ) {
    int prt = priority( expr[i] );
    if( prt <= minPrt ) {
      minPrt = prt;
      pos = i;
      }
    }
  return pos;
  }
static int priority( char op )
  {
  if( op == '+' || op == '-') return 1;
  if( op == '*' || op == '/') return 2;
  return 100;
  }
static int doOperation( char op, int n1, int n2 )
  {
  if (op == '+') return n1 + n2;
  else if (op == '-') return n1 - n2;
  else if (op == '*') return n1 * n2;
  else return n1 / n2;
  }
```

В коде эти функций есть два «новшества»:

- длина строки с C# определяется с помощью свойства `Length`; это именно свойство, а не метод, поэтому круглые скобки в конце записывать не нужно;
- оператор `||`, использованный в функции `priority`, выполняет логическую операцию ИЛИ[1)].

Таким образом, мы разработали модель задачи — класс `Calculator`, с помощью которого вычисляется арифметическое выражение, записанное в строке.

[1)] Такое же обозначение применяется в языках C, Java, JavaScript. А в C++ вместо `||` можно писать слово **or**.

Вычисление арифметических выражений: представление

Теперь построим интерфейс программы (рис. 2.15).

Рис. 2.15

В верхней части окна находится выпадающий список — компонент **ComboBox**. Под ним размещёно поле вывода результата — многострочный редактор текста. Это компонент **TextBox**, у которого установлено свойство Multiline, равное True.

Пользователь вводит выражение в окне выпадающего списка. При нажатии на клавишу *Enter* выражение вычисляется и его результат выводится в последней строке поля вывода. Кроме того, каждое новое выражение добавляется в верхний выпадающий список. С помощью этого списка можно вернуться к введённому ранее выражению и исправить его.

Добавляем компонент **ComboBox** на форму, назовём его input (в переводе с английского — ввод). Чтобы прижать список к верхней границе окна, установим свойство Dock равным Top («прижать вверх»).

Для компонента **TextBox** выбираем выравнивание Fill (заполнить всю свободную область), имя answers (в переводе с английского — ответы) и свойство Multiline равное True («да», это многострочный редактор).

Чтобы пользователь не мог изменять поле вывода, у компонента answers устанавливаем логическое свойство ReadOnly (в переводе с английского — только для чтения), равное True («да»). При этом фон компонента станет серым. Если вам это не нравится, нужно изменить свойство BackColor.

Логика работы программы может быть записана в виде псевдокода:

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

Перехватим нажатие клавиши *Enter* с помощью обработчика события KeyPress компонента input. Клавиша *Enter* имеет код 13, поэтому условие «если нажата клавиша *Enter*» запишется так:

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

Значение арифметического выражения будем вычислять с помощью функции calc:

```
int x = Calculator.calc( input.Text );
```

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

Содержимое поля вывода **TextBox** хранится как свойство Text, для разбивки на строки используется сочетание двух служебных символов: \r (возврат в начало строки) и \n (перевод строки). Допишем введённое выражение и результат его вычисления в конец текста в виде отдельной строки:

```
answers.Text += input.Text + "=" +
                x.ToString() + "\r\n";
```

Результат вычислений (x) переведён в символьный вид с помощью метода ToString.

Строки, входящие в выпадающий список, доступны как свойство-список Items объекта input. Метод FindString служит для поиска строки в списке и возвращает индекс найденного элемента или значение –1, если образец не найден. Поэтому добавить в список новую строку можно следующим образом:

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

Метод Insert добавляет строку в указанное место списка. Первый аргумент — это позиция новой строки в списке. Нумерация элементов списка начинается с нуля, поэтому позиция 0 — это самое начало списка.

Приведём полностью обработчик события 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 );
    }
  }
```

Новый компонент — зачем это нужно?

На практике удобно использовать специальные поля ввода, с помощью которых можно вводить целые числа. Например, такие поля очень

пригодились бы, когда мы писали программу для работы с кодами цвета в модели RGB.

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 в своей программе:

- добавили обработчик события `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 создаётся *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 поле ввода стандартной библиотеки — компонент **TextBox**. Это значит, что наш компонент (назовём его **IntTextBox**) будет наследником класса **TextBox**, а класс **TextBox** будет соответственно базовым классом для *нового* класса **IntTextBox**.

Перечислим изменения, которые нужно внести в класс **TextBox**:

- 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` (в переводе с английского — значение) целого типа.

Добавляем новый компонент

Начнём новый проект — программу, которая будет переводить целые числа из десятичной системы в шестнадцатеричную (рис. 2.16).

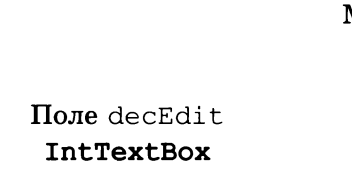


Рис. 2.16

Для формы запретим изменение размеров: установим свойство `FormBorderStyle` равным `FixedSingle`.

Теперь создадим новый компонент — наследник класса **TextBox**. Для этого нужно добавить к проекту новый класс: выбрать пункт меню *Проект — Добавить класс* и ввести имя этого класса — **IntTextBox**. После этого в окне обозревателя решений появится новый исходный файл `IntTextBox.cs`.

Откроем этот файл двойным щелчком и укажем, что новый класс является наследником класса **TextBox**:

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

Если теперь попытаться запустить программу, вы увидите сообщение об ошибке: класс **TextBox** неизвестен. Чтобы исправить эту ошибку, нужно в начале файла IntTextBox.cs подключить библиотеку, в которой описан класс **TextBox**:

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

После сборки программы в верхней части окна *Панель элементов* появляется новый компонент **IntTextBox** (рис. 2.17), который можно добавлять на форму так же, как и стандартные компоненты. Правда, пока он ничем не отличается от компонента **TextBox**.

Рис. 2.17

Изменение поведения компонента

Добавим в класс **IntTextBox** (в файле IntTextBox.cs) новые методы. Сначала создадим 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** нам уже знакомо: оно означает, что мы переопределяем метод базового класса с таким же именем. Описатель **protected**

говорит о том, что этот метод будет доступен всем наследникам класса **IntTextBox** и недоступен для остальных классов.

В последней строке вызывается метод OnKeyPress базового класса:

```
base.OnKeyPress(e);
```

Здесь base — это служебное слово языка C#, которое используется для обращения к базовому классу из методов класса-наследника. Мы вызываем метод базового класса для того, чтобы не отключить действия, которые там выполняются (и о которых мы не знаем!).

Теперь добавим в описание класса новое свойство:

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

В первой строке объявляется свойство с именем Value. Значение свойства — это целое число (int). Свойство будет доступно всем объектам, на это указывает слово **public**.

Свойство в C# определяется двумя методами — «сеттером», который всегда называется set, и «геттером», за ним закреплено имя get.

Метод set служит для изменения значения свойства. Новое значение, которое *всегда* называется value (все буквы строчные!), преобразуется в символьную строку и сразу копируется в свойство Text нашего компонента.

В методе get (чтение значения свойства) пытаемся преобразовать строку в целое число[1)], и в случае неудачи возвращаем 0.

Вот полное описание нового класса:

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

[1)] 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.

```
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 **IntTextBox**, назовём его decEdit. Кроме того, добавим метку **Label** с именем hexLabel для вывода шестнадцатеричного значения.

Для нового компонента decEdit необходимо определить обработчик события TextChanged — при изменении содержимого поля ввода нужно показать соответствующее шестнадцатеричное число с помощью метки hexLabel. Выделим поле ввода и добавим 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 («X») с помощью метода ToString.

Выводы

- Для того чтобы объединить несколько функций в отдельный модуль, нужно создать статический класс. В языке C# нельзя создавать функции, не относящиеся к какому-то классу.
- Статический класс — это класс, который содержит только методы, у него нет данных. Объекты такого класса не создаются.
- Все методы статического класса должны быть статическими. Их нужно вызывать с помощью точечной записи, но вместо имени объекта указывать имя класса.
- Новые классы компонентов обычно строятся на основе существующих (как классы-наследники).
- При сборке программы новые компоненты автоматически добавляются в *Панель элементов*. После этого с ними можно работать так же, как и со стандартными компонентами.

- При переопределении методов базового класса в новом компоненте нужно вызывать одноимённый метод базового класса.
- В языке C# объекты могут иметь свойства. Свойства определяются двумя методами: метод `get` вызывается при чтении значения свойства, а метод `set` — при записи значения.

Вопросы и задания

1. Что такое пространство имён? Как пространства имён используются для связи между модулями программ?
2. Что означает описатель **static** при объявлении классов и методов?
3. Почему в классе **Calculator** из параграфа только функция `calc` объявлена с описателем **public**?
4. Зачем используют свойство `ReadOnly`?
5. Как в C# преобразовать числовое значения в текстовое и обратно? Как перевести число в шестнадцатеричную систему счисления?
6. *Проект*. Измените программу `StringCalculator` из параграфа так, чтобы она вычисляла выражения с вещественными числами. Для перевода вещественных чисел из символьного вида в числовой можно использовать метод `double.Parse`.
7. *Проект*. Добавьте в программу `StringCalculator` из параграфа обработку ошибок. Подумайте, какие ошибки может сделать пользователь. Какие ошибки могут возникнуть при вычислениях? Как их обработать?
8. Изучите исходные тексты файлов программы `StringCalculator` из параграфа. Сделайте предположение, как связаны форма и класс `Calculator` (откуда форма «знает» такой класс и его методы).
9. Пусть требуется изменить программу `StringCalculator` из параграфа так, чтобы она обрабатывала выражения со скобками. Что нужно изменить: модель, интерфейс или и то, и другое?

*10. *Проект*. Измените программу `StringCalculator` из параграфа так, чтобы она вычисляла выражения со скобками. *Подсказка*: нужно искать последнюю операцию с самым низким приоритетом, стоящую вне скобок.

**11. *Проект*. Измените программу из предыдущего задания так, чтобы в выражении можно было использовать функции `abs`, `sin`, `cos`, `sqrt`.

12. *Проект*. Постройте программу «Калькулятор» для выполнения вычислений с целыми числами (см. рисунок справа).

13. *Проект*. Постройте программу «Калькулятор» для выполнения вычислений с вещественными числами.

*14. *Проект.* Постройте программу «Римский калькулятор» для выполнения вычислений с числами в римской системе счисления.

15. В каких случаях имеет смысл разрабатывать свои компоненты?

16. Как вы думаете, почему программисты редко создают свои компоненты «с нуля»?

17. Объясните, как связаны классы компонентов **IntTextBox** (из параграфа) и **TextBox**. Чем они различаются?

18. Что означают описатели **protected** и **override** при описании метода?

19. Объясните, как работает свойство `Value` у компонента **IntTextBox** из параграфа.

20. Зачем в обработчике события `OnKeyPress` нового компонента (из параграфа) вызывается метод базового класса с тем же именем? Установите для вашего компонента обработчик события `KeyPress`. Попробуйте убрать вызов метода базового класса. Проверьте, работает ли обработчик.

21. *Проект.* Разработайте компонент, который позволяет вводить шестнадцатеричные числа.

*22. *Проект.* Разработайте компонент **FloatTextBox**, который позволяет вводить вещественные числа и обрабатывает ошибки.

**23. *Проект.* Придумайте свой компонент и напишите программу, в которой он используется.

Интересные сайты

csharp.net-informations.com — справочник по языку C#

tutorialspoint.com/csharp/ — онлайн-учебник по C#

rextester.com — онлайн-компилятор C#

ЗАКЛЮЧЕНИЕ

Вся наша жизнь всё в бóльшей степени зависит от программ, которые управляют роботами, «умными домами», автомобилями, самолётами. Поэтому роль программистов в современном мире очень велика. Можно сказать, что они во многом определяют, насколько комфортным и безопасным будет дальнейший путь развития человечества.

Одна из главных задач современного программиста — написать надёжную программу, которая содержит минимальное число ошибок. Для этого программа должна быть хорошо спроектирована, т. е. правильно разбита на составные части, которые можно отлаживать отдельно друг от друга.

Многолетний опыт показал, что разбиение на объекты и использование ООП позволяет успешно разрабатывать очень сложные системы, содержащие сотни тысяч и даже миллионы строк исходного кода.

В этой части пособия мы познакомились с ООП на примере современных языков программирования: Python, C++ и немного C#. Конечно, мы изучили лишь небольшую часть возможностей этих языков. Хочется надеяться, что вы этим не ограничитесь и будете дальше совершенствовать свои знания. Для этого нужно изучать дополнительную литературу, справочники, источники в сети Интернет. Небольшой список литературы приведён в конце этой книги.

И самое главное — не забывайте, что для того чтобы научиться программировать, нужно программировать. Пишите самые разные программы: программируйте игры, анализируйте «большие данные», обрабатывайте информацию с веб-сайтов, попытайтесь с помощью своих программ автоматизировать операции, которые вы обычно выполняете вручную.

Успехов вам на этом интересном, но трудном пути!

Литература для дальнейшего изучения

Язык Python

1. *Доусон М.* Программируем на Python: Пер. с англ. — СПб.: Питер, 2014.
2. *Мэтиз Э.* Изучаем Python. Программирование игр, визуализация данных, веб-приложения.: Пер. с англ. — СПб.: Питер, 2017
3. *Любанович Б.* Простой Python. Современный стиль программирования.: Пер. с англ. — СПб.: Питер, 2016.

4. *Прохоренок Н. А.*, Python 3 и PyQt. Разработка приложений. — СПб.: БХВ-Петербург, 2012.
5. *Саммерфилд М.* Программирование на Python 3. Подробное руководство.: Пер. с англ. — СПб.: Символ-Плюс, 2009.
6. *Лутц М.* Изучаем Python, 4-е изд.: Пер. с англ. — СПб.: Символ-Плюс, 2011.
7. *Рамальо Л.* Python. К вершинам мастерства.: Пер. с англ. — М.: ДМК Пресс, 2016.

Язык C++

1. *Страуструп Б.* Программирование: принципы и практика с использованием C++, 2-е изд.: Пер. с англ. — М.: ООО «И. Д. Вильямс», 2016.
2. *Липпман С. Б., Лажойе Ж., Му Б. Э.* Язык программирования C++. Базовый курс, 5-е изд.: Пер. с англ. — М.: ООО «И. Д. Вильямс», 2014.
3. *Шилдт Г.* C++. Базовый курс, 3-е изд.: Пер. с англ. — М.: ООО «И. Д. Вильямс», 2010.
4. *Лафоре Р.* Объектно-ориентированное программирование в C++, 4-е изд.: Пер. с англ. — СПб.: Питер, 2004.
5. *Мюссер Д., Дердж Ж., Сейни А.* C++ и STL: справочное руководство, 2-е изд.: Пер. с англ. – М.: ООО «И. Д. Вильямс», 2010.

Язык C#

1. *Дрейер М.* C# для школьников: Учебное пособие: Пер. с англ. — М.: Интернет-Университет Информационных Технологий; БИНОМ. Лаборатория знаний, 2009.
2. *Хейлсберг А., Торгерсен М., Вилтамут С., Голд П.* Язык программирования C#. Классика Computers Science. 4-е изд.: Пер. с англ. — СПб.: Питер, 2012.
3. *Троелсен Э.* Язык программирования C# 5.0 и платформа .NET 4.5, 6-е изд.: Пер. с англ. — М. : ООО «И. Д. Вильямс», 2013.
4. *Шилдт Г.* C# 4.0. Полное руководство: Пер. с англ. — М.: ООО «И. Д. Вильямс», 2011.

ОГЛАВЛЕНИЕ