Объектно-ориентированное программирование

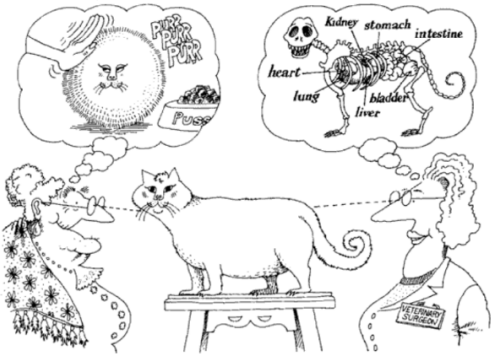
Юрий Литвинов
y.litvinov@spbu.ru

25.02.2022

1. Основные понятия объектно-ориентированного программирования

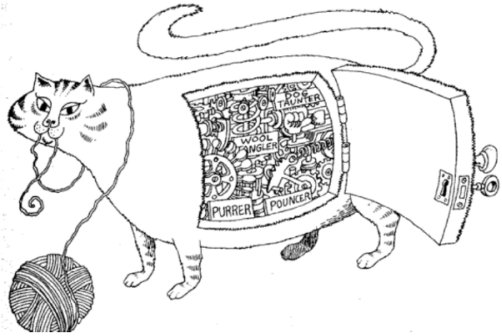
Как, наверное, и так известно, в объектно-ориентированном стиле программа представляется в виде набора объектов. **Объект** — это набор данных и процедур их обработки, вместе представляющий некую независимую сущность. То есть объект имеет своё состояние и своё поведение, и никто извне объекта не вправе делать предположения о том, как именно он устроен. Объекты общаются друг с другом посредством посылки/приёма сообщений. Важное отличие концепции сообщений от вызова функции — объект сам решает, как реагировать на сообщение, и тот код, который вызовется как реакция на какое-то сообщение, вообще говоря, заранее (во время компиляции) неизвестен. Кроме того, у объекта есть **интерфейс** — собственно, набор сообщений, которые он может обрабатывать. То есть, объект можно представлять себе как чёрный ящик, у которого есть какие-то входы, куда можно послать какие-то данные и получить ответ. В ООП такую точку зрения на объект называют **инкапсуляцией**, подразумевая под ней одновременно и сокрытие деталей реализации (наружу видно только интерфейс объекта, или набор интерфейсов), и сбор всех данных и методов их обработки в одном месте — в самом объекте.

Не следует относиться к объектам как к структурам с методами. Объект — это обособленная часть системы, которая обладает каким-то поведением и взаимодействует с остальным миром через какой-то интерфейс. Пожалуй, самое важное понятие для объектно-ориентированного программирования (как и для программирования вообще и для процесса познания ещё более вообще) — абстракция. **Абстракция** — это выделение существенных характеристик чего-то, что существует в предметной области, чтобы с этим было можно работать. Например, рассмотрим картинку из знаменитой книги Гради Буча «Объектно-ориентированный анализ и проектирование»:



Кот, как объект реального мира, обладает бесконечным количеством различных атрибутов, но интересны нам только некоторые. Причём, какие именно — зависит от нашей точки зрения. Например, для бабушки важно, что кота можно гладить, а для ветеринара — что у него есть различные внутренние органы. Кот, как сущность реального мира, может порождать сразу несколько абстракций.

Абстракция хороша тем, что позволяет нам не думать о несущественных деталях. У любой абстракции можно выделить её **контракты** — то, что она должна делать, что можно делать с ней и т.д., и её реализацию — то, как она это должна делать. Инкапсуляция на самом деле разделяет контракты и реализацию объекта, «пряча» от пользователя объекта его внутреннюю сложность. Например, тот же кот может быть очень сложно устроен внутри, но бабушка, когда его гладит, имеет право ничего об этом не знать:



Ещё одним важным понятием для ООП является **инвариант** объекта — некоторый набор условий на состояние объекта, которые выполняются после создания объекта и после вызова каждого его метода всё время жизни объекта. Например, что у квадрата стороны всегда равны, что поле *size* списка должно содержать число, равное количеству элементов списка. Объект сам отвечает за поддержание своего инварианта, и все его методы реализуются в предположении, что инвариант всегда выполняется. Поэтому, в частности, следует избегать публичных полей — они не контролируются самим объектом, так что инвариант может быть нарушен извне, и объект не сможет этому помешать. Если же поле не является частью инварианта (то есть на него нет никаких условий), это является как правило признаком плохого дизайна — непонятно, что поле в таком случае делает в этом объекте. Обратите внимание, что инвариант может нарушаться, пока объект находится посреди

своего метода (например, когда мы добавляем новый элемент в список, мы могли ещё не успеть обновить *size*). Но при выходе из метода инвариант снова должен начать выполняться.

В некоторых ОО-языках программирования (например, в Java, C++ и C#) есть понятие класса. **Класс** — это тип объекта. Класс описывает, какими полями обладает объект и какие методы у него есть. Типы полей определяет класс, а вот значения полей свои для каждого объекта. Бывают ещё **поля класса** — с модификатором *static* в C#, их значение одинаково для всех объектов данного класса, путать их с полями объекта нельзя. Итак, объект — экземпляр класса. Например, есть класс «студент», у которого есть такие поля как номер зачётки, оценки за последнюю сессию и т.д., а есть объект «Иван Иванов» класса студент, у которого номер зачётки вполне определённый. При этом, скажем, ссылка на закон, согласно которому студенты не идут в армию, не зависит от конкретного студента, поэтому её можно считать полем класса.

Классы могут состоять в отношениях **генерализации**, то есть между ними может существовать связь «более общее понятие — более конкретное понятие». Например, всякий студент, я надеюсь, человек. Положим, что у человека есть атрибуты *имя* и *фамилия*, выходит, что и у студента они есть (поскольку любой студент — человек). В таком случае говорят, что класс «студент» является **наследником** класса «человек», наследует его атрибуты и поведение. Это и есть сущность наследования в ООП — объект класса-наследника **является** объектом класса-предка (**is-a**) и может использоваться везде, где может использоваться объект класса-предка. Зачем это надо — для переиспользования кода. Мы можем написать функцию/процедуру/метод, который будет работать с объектами наиболее общего типа, для которого этот код имеет смысл, а обрабатываться будут на самом деле объекты более узких типов, которые могут быть практически полезны. Например, сортировке пузырьком плевать, что сортировать, лишь бы это что-то умело сравнивать себя с другим таким же чем-то. Сортировку пузырьком можно написать над объектами класса «штука, которая имеет метод сравнения», и она будет работать, даже не зная, что именно она сортирует — числа или столбцы матрицы. Ещё переиспользования кода можно добиться, определяя общие атрибуты и методы двух классов в одном общем классе-предке. Например, отчислить можно и студентов и аспирантов, так что вместо того, чтобы описывать, как именно отчислять студентов и аспирантов, можно описать, как надо отчислять объект класса «учащийся», классы «студент» и «аспирант» наследуют метод отчисления автоматически.

Кстати, такого рода переиспользование кода и переиспользование кода предыдущего вида вообще говоря независимы. То, что использует свойства отношения «является», называется в науке о системах типов «сабтайпингом» (или subtype polymorphism, полиморфизмом подтипов) и является довольно важным понятием, на которое завязаны системы типов современных ОО-языков, второе никакого отношения к типам вообще не имеет (напомним, что тип — это набор значений и допустимых операций над ними, как именно реализованы операции и что они делают, систему типов языка не интересует — что реализация описана в предке или в потомке, систему типов не интересует тем более, она проверяет лишь корректность применения операций). В C++ эти два вида наследования явно разделены, во всех нормальных языках наследование влечёт сабтайпинг.

Тут уже должно быть понятно, что один объект может иметь несколько типов сразу — являться экземпляром нескольких классов. Иван Иванов — экземпляр класса «студент» и экземпляр класса «человек», и вообще, экземпляр некоего класса является и экземпля-

ром всех классов-предков. Более того, объект может являться экземпляром нескольких несвязанных классов — например, тот же гипотетический Иван может являться экземпляром класса «студент» и экземпляром класса «любитель дотки». В таком случае объект имеет все атрибуты и первого, и второго класса. Более того, объекты могут вообще менять свои классы во время жизни — например, когда наш Иван закончит университет, он перестанет быть студентом, но не перестанет быть Иваном, то есть не лишится своей идентичности. Такие дела огорчают статическую типизацию, поэтому в распространённых языках не используются. Более того, в нормальных языках объект всегда создаётся как экземпляр некоего известного во время компиляции класса, так что чтобы у нас был объект Иван, нам потребуется сначала класс «студент — любитель дотки», который будет наследовать от классов «студент» и «любитель дотки», зато потом можно создать кучу студентов — любителей дотки. Важно понимать, что каждый класс, экземпляром которого является объект, задаёт интерфейс доступа к этому объекту — к Ивану можно обратиться как к студенту (например, поставить ему зачёт), или как к любителю дотки (например, поговорить с ним о последней катке). Объект будет один и тот же, а вот набор допустимых действий с ним и его поведение могут быть разными. Это же касается классов и в иерархии наследования — интерфейс, по которому можно обратиться к объекту как к экземпляру класса-предка, уже, чем интерфейс объекта как экземпляра класса-потомка.

В нормальных языках (со строгой статической типизацией, как C++, Java и C#) из всего множества типов, которым соответствует объект, выделяется один — тот, с которым этот объект создавался. Этот тип имеет физический смысл «настоящего» типа объекта, того, «чем является объект на самом деле». Конечно, объект на самом деле является объектом любого из типов-предков, но как он реализует свои операции — написано именно в его «настоящем» типе. До реализации операций, как уже говорилось, системе типов дела нет, так что с точки зрения теории всё хорошо, а с точки зрения практики это понятие весьма важно для полиморфных вызовов. Но про это чуть потом, пока надо запомнить, что этот «настоящий» тип объекта называется его **типом времени выполнения** (**runtime type**), а типы объекта, про которые знает компилятор, и операциями которых можно пользоваться в программе — **типами времени компиляции** (**compile-time type**). Разницу можно пояснить на таком примере:

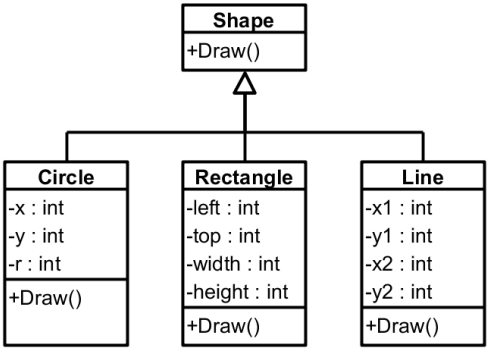
```
Shape a = new Circle();
```

У переменной *a* будет тип времени компиляции *Shape* (и все его предки, да), и вызывать методы мы можем только те, которые объявлены в *Shape* (или его предках, да). А тип времени выполнения у этой же переменной будет *Circle*, но компилятор про это не знает (знает, но забудет строчкой ниже, поскольку переменная объявлена как *Shape*), и, в силу строгой типизированности C#, вызывать методы, специфичные для *Circle*, у переменной *a* будет нельзя. Для тех, кто не в теме — не надо пугаться, про это ещё будет ниже.

Ещё одним важным для ООП принципом является то, что потомки могут переопределять поведение предков. Например, обычно если людям рассказывать про ООП, им плевать, а если студентам рассказывать про ООП, они получают некие знания. Заметьте, что человеку можно рассказывать про ООП, следовательно, и студенту тоже, и преподу по сути всё равно, кому рассказывать, главное, чтобы рассказывать было можно — но результат получается разный. Собственно, в этом и состоит большая часть радости от наследования, и возможность переопределения поведения + сабтайпинг обычно и называют **полимор-**

физмом. Объект, как говорилось, реагирует на сообщение, сообщение может быть послано объекту как экземпляру класса-родителя, а как оно будет обработано — решает сам объект, это зависит от его «настоящего» типа. Зачем это нужно — опять-таки, для переиспользования кода, когда в базовом классе определяется некоторое поведение по умолчанию, а в некоторых классах-потомках это поведение переопределяется, или для того, чтобы можно было менять поведение некоего кода прямо во время выполнения. Например, у нас есть код, выводящий матрицу по спирали. Мы хотим в зависимости от желания пользователя вывести результат в файл или на консоль. Ему можно передать объект класса «Выводилка», у которого есть метод «вывести», который, скажем, просто никуда ничего не выводит, от него унаследованы два класса «ВыводилкаНаКонсоль» и «ВыводилкаВФайл», в которых метод «вывести» переопределён соответственно. Метод печати матрицы по спирали получает в качестве параметра объект типа «Выводилка» и вызывает его метод «вывести». А основная программа в зависимости от выбора пользователя создаёт объект либо одного класса, либо другого — и передаёт параметром печаталке спирали. Очень удобно, печаталка даже не знает, что и куда она там выводит.

Или ещё один пример: рассмотрим графический редактор с разными геометрическими фигурами. Можно описать такую иерархию наследования:



Класс *Shape* определяет операцию *Draw*, которую должны реализовать все его потомки, от него наследуются круг, прямоугольник и линия, каждый из которых реализует *Draw* по-разному. И тогда мы можем нарисовать все фигуры на сцене, просто один раз пройдя по списку объектов класса *Shape* — поскольку у нас есть полиморфизм, каждый из этих объектов на самом деле будет какой-то конкретной фигурой, которая реализует метод *Draw* по-своему. И каждая фигура отрисует себя так, как она должна себя рисовать, при этом наш код даже не будет знать, что это была за фигура (более того, конкретный класс-фигура может ещё не существовать в тот момент, когда мы пишем код редактора — например, взяли и добавили стрелочки и звёзды). Вот небольшой пример кода, поясняющий сказанное:

```
List<Shape> shapes = new List { new Circle(0, 0, 5), new Line(0, 0, 10, 10) };

foreach (var shape in shapes)
{
    shape.Draw();
}
```

Для того, чтобы не придумывать реализацию по умолчанию для методов, которые бессмысленны в родительском классе — типа нашего метода *Draw* для абстрактной фигуры *Shape*, придумали понятие абстрактного метода — метода, для которого реализация не определена вовсе. Мы объявляем, что класс имеет такой-то метод с таким-то набором параметров и таким-то возвращаемым значением, но как он реализован — определяют классы-потомки. Понятно, что объект класса с абстрактным методом создать нельзя — он не будет знать, что делать, если абстрактный метод вызовут. А вот если объект имеет «настоящий» тип класс-потомок, у него этот метод переопределён, так что вызвать его можно и существование объекта осмысленно. Классы с абстрактными методами называются абстрактными классами. Есть ещё интерфейсы — это по сути классы, у которых все методы абстрактные. Надо понимать их так — они определяют некий интерфейс, или протокол взаимодействия, которому удовлетворяет объект, его реализующий, и всё, без каких-либо попыток предоставить реализацию. Тогда *Shape* можно сделать интерфейсом, а конкретные фигуры — классы, его реализующие. Код можно писать в терминах интерфейсов, подставляя на их место объекты конкретных классов уже во время выполнения, это повышает уровень абстрактности кода, а значит, и возможностей для его переиспользования.

2. Объектно-ориентированное программирование в C#

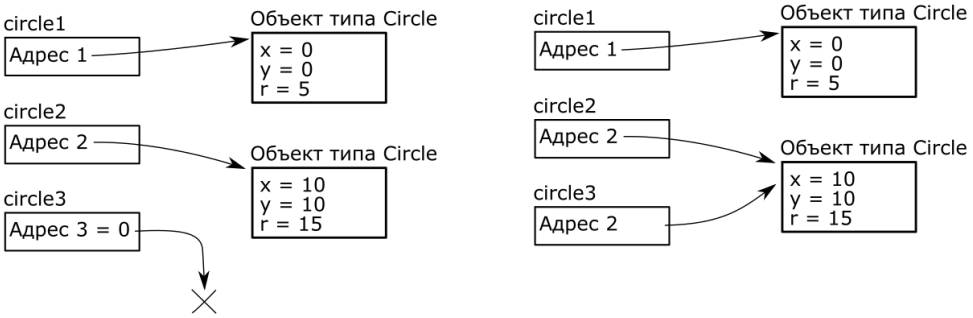
2.1. Ссылочные типы и типы-значения

Если объекты-потомки можно использовать везде, где используются объекты-предки, то их, например, и в массивы можно класть, и копировать. А если они разные по размеру, что делать? Индексация в массивах выполняется путём умножения индекса на размер элемента, так что просто хранить в массиве разные по размеру элементы нельзя. Надо использовать не сами объекты, а указатели. Указателей в C# нет (ну, почти), вместо них используются ссылки. Ссылки в C# почти такие же, как в C++, если кто в курсе, только в C++ если ссылке присвоили значение, то поменять его уже нельзя, только значение объекта, на который она ссылается. В C# (как и в Java, кстати) присваивание ссылочной переменной — это именно присваивание ссылочной переменной, а не объекту, на который она ссылается, в этом смысле ссылочные переменные больше похожи на указатели в C. Если у нас есть такой вот код:

```
var circle1 = new Circle(0, 0, 5);
var circle2 = new Circle(10, 10, 15);
var circle3 = null;

circle3 = circle2;
```

то на картинке это выглядит как-то так:



Очень похоже на указатель в C, только доступа к содержимому самой ссылочной переменной у нас нет. Например, арифметика указателей не получится в принципе, сделать ссылку на ссылку тоже нельзя. Зато и разыменовывать такой «указатель» не нужно: можно писать просто *circle3.x*. Важно только помнить, что «=» не вызывает оператор присваивания класса *Circle*, а присваивает значение ссылочной переменной, а «==» не сравнивает значения объектов, а проверяет, что две ссылочные переменные ссылаются на один объект (если операторы не перегружены, по умолчанию поведение именно такое).

Для ссылочных переменных нет никакого специального синтаксиса объявления, просто все переменные ссылочных типов — ссылочные. Вообще все типы делятся в C# на типы-значения и ссылочные типы (в Java, кстати, тоже). Разница между ними в выделении памяти и в понятии **идентичности** (**identity**) (и связанных с ним заморочек, касающихся поведения операторов копирования и сравнения). Начнём с памяти. Под ссылочные типы память выделяется всегда в куче (под то, на что указывают ссылки, а не под сами ссылки, это важно понимать), всегда с использованием оператора *new*. Это в C мы могли выбирать, на стеке заводить объект или в куче (*Point *a = malloc(sizeof(Point));* или просто *Point a;*), в C# всё решили за нас. Под типы-значения память выделяется всегда на стеке, соответственно, время жизни объекта-значения — пока он не вышел из области видимости. Это удобно, потому что избавляет нас от необходимости думать, когда же наш объект умрёт (хотя тут это не так важно, как в C, потому что в куче работает сборщик мусора, и «забытые» объекты всё равно не проживут долго).

Идентичность некоторым образом связана с выделением памяти. Идентичность ссылочных типов данных определяется их местоположением в куче (просто адресом объекта), так что две ссылки равны, если они указывают на один и тот же объект. Это важно. Если у нас есть два абсолютно одинаковых объекта, с абсолютно одинаковыми полями, то это всё равно будут разные объекты. Например, бывают однофамильцы, бывают люди, у которых совпадает имя и фамилия, и это всё равно разные люди. С типами-значениями всё не так, их идентичность определяется их значением. Два числа «1» равны, глубоко всё равно, что это за единицы и где они физически находятся. То же относится к, скажем, точкам на плоскости — две точки равны, если их координаты совпадают.

Так вот, ссылочные типы — это классы (в том числе, строки, массивы, исключения, делегаты). Типы-значения — это примитивные типы (*int*, *bool*, *double* и т.д.), типы-перечисления и структуры. Некоторая тонкость заключается в том, что формально в C# всё является классом (и наследуется от общего базового класса Object, но об этом потом), ведь даже у примитивных типов есть методы (*1.ToString()* прекрасно будет работать, например), но некоторые классы наследуют от класса *ValueType*, вот они-то и будут типами-

значениями. К счастью, в отличие от той же Java, в C# есть перегрузка операторов, поэтому со ссылочными типами не так неудобно работать, как в той же Java — сравнивать строки (ссылочные типы!) оператором «==» в C# можно, и это приведёт к ожидаемым результатам (в Java можно было сравнить две одинаковые строки и получить, что они не равны, просто потому, что они лежат по разным адресам в памяти, впрочем, в C с *char** было так же; в C# всё будет ок, они будут равны). Для ссылочных типов ещё определено понятие ссылки, которая указывает в никуда — *null* (как в C *NULL*).

Со ссылочными типами связана ещё одна неприятная тонкость. Рассмотрим подпрограмму, которая должна бы возвращать с помощью переменной *s3* сумму строк, хранящихся в переменных *s1* и *s2*:

```
static void Add(string s1, string s2, string s3)
{
    s3 = s1 + s2;
}
```

На первый взгляд кажется, что работать не должно — мы меняем значение параметра функции, так что извне этого изменения видно быть не должно. Но потом мы принимаем во внимание, что строки — ссылочные типы, так что передаются не сами строки, а ссылки на строки, так что если мы что-то с этими строками делаем, то извне это должно быть видно. В C++, например, значение *s3* вовне бы благополучно поменялось, если бы он передавался как ссылка, с амперсандом. Но потом мы понимаем, что объекты-то передаются по ссылке, но ссылочные переменные передаются по значению, то есть если строку поменять, она честно поменяется, но если ссылочной переменной *s3* присвоить новую строку (результат сложения), то она просто забудет про старую строку, начнёт указывать на новую, и помрёт при выходе из метода (к этому моменту те, кто не в теме, должны уже запутаться, но на всякий случай скажу, что тут-то и проявляется отличие семантики ссылок от C++ — в C++ ссылки немутабельны, начав указывать на какой-то объект, ссылка в C++ указывает на него всегда, а в C# и Java ссылки мутабельны, то есть могут менять объект, на который они указывают). В данном случае, конечно, лучше возвращать результат конкатенации, и вызывающий уже может делать с ним, что хочет, но в отличие от Java, где надо делать только так, и, например, функцию *swap* никогда не написать, в C# есть ключевые слова *ref* и *out*, которые работают как настоящая передача параметров по ссылке в C++. Передача ссылочных переменных по ссылке, так-то. Выглядит это так:

```
static void Add(string s1, string s2, ref string s3)
{
    s3 = s1 + s2;
}

private static void Main(string[] args)
{
    string s1 = "a";
    string s2 = "b";
    string s3 = "c";
    Add(s1, s2, ref s3);
}
```

Или то же самое с *out*:

```
static void Add(string s1, string s2, out string s3)
{
    s3 = s1 + s2;
}

private static void Main(string[] args)
{
    string s1 = "a";
    string s2 = "b";
    Add(s1, s2, out string s3);
}
```

С типами-значениями *ref* и *out*, естественно, тоже работают, и эффект от этого вполне ожидаем. На этом про ссылочные типы пока всё, к конструкциям типа *listElement.next = listElement.next.next;* вполне можно привыкнуть, если делать домашку.

2.2. Конструкторы

Объекты создаются с помощью конструкторов, и только с помощью конструкторов. Конструктор — это такой специальный метод, который не имеет типа возвращаемого значения (вообще, даже *void*), и называется так же, как и класс. Конструктор в классе можно не писать, тогда для него будет определён конструктор по умолчанию, который не принимает аргументов и ничего не делает. Зачем они нужны и как их вызывать: когда мы пишем, например, `Circle circle = new Circle(0, 0, 10);`, то во-первых, под объект класса *Circle* выделяется память на куче, достаточная для хранения всех его полей (включая поля классов-предков), во-вторых, вызывается его конструктор, который должен инициализировать его поля какими-то осмысленными значениями и сделать всё, что нужно, чтобы класс можно было использовать по назначению (ну, не всё так просто, потому что есть ещё поля классов-предков, для инициализации которых используются конструкторы предков, которые вызываются перед нашим конструктором, так что конструирование объекта начинается с вызова конструкторов вверх по цепочке наследования, а затем исполнения конструкторов вниз, до конструируемого объекта, но можно пока не грузиться). В конструкторе можно писать любой код, например, открытие сетевого соединения, чтение из файла, и т.д. Заканчиваться исполнение конструктора должно тем, что для объекта выполнен его инвариант (вспомните, я в начале пары говорил), и объект готов обрабатывать запросы. То есть, если сначала надо создать объект (при этом вызовется конструктор), а потом вызвать какой-нибудь метод вроде *Init* (или, ещё ужаснее, вручную заполнять поля или свойства), то кто-нибудь когда-нибудь наверняка забудет это сделать, и всё сломается. Конструктор же забыть вызвать не получится, иных способов создать объект нет, так что всю инициализацию, если это возможно, надо делать там.

Реализован конструктор может быть, например, так:

```
class Circle
{
    public Circle(int x, int y, int r)
```

```
    {
        this.x = x;
        this.y = y;
        this.r = r;
    }

    private int x;
    private int y;
    private int r;
}
```

Никакой магии, кроме, пожалуй, использования ключевого слова *this* — ссылки на текущий объект. Этой штукой тут надо пользоваться, поскольку имена параметров конструктора перекрывают поля класса, и если бы мы просто написали *x = x;*, то ничего бы не произошло. *this* же указывает на тот объект, у которого был вызван метод, так что *this.x* вернёт нам значение поля *x* того экземпляра класса *Circle*, с которым мы сейчас работаем. Вообще, поля в методах доступны и непосредственно, писать всюду *this* не нужно (хотя и считается хорошим тоном, да). Кстати, все поля классов в C# всегда инициализируются значениями по умолчанию (*0* для целых, *false* для булевых, *null* для всех ссылочных типов и т.д., кстати, строки по умолчанию будут не пустыми, а *null*). Конструкторы можно перегружать, то есть определять в классе несколько конструкторов с разными параметрами, и даже вызывать из одного другой, например, так:

```
class Circle
{
    public Circle(int x, int y, int r)
    {
        this.x = x;
        this.y = y;
        this.r = r;
    }

    public Circle(int r)
        : this(0, 0, r)
    {
    }

    private int x;
    private int y;
    private int r;
}
```

Тут *this* имеет несколько другое значение, но суть та же — тот объект, от которого мы вызываем метод. Можно вызывать конструктор предка с помощью ключевого слова *base*, так же, как и *this*. Если объявлен хоть один конструктор, конструктор по умолчанию не генерируется, но его несложно при желании реализовать руками:

```
Circle()
{
}
```

Порядок вызовов при создании объекта некого класса (будем называть его дочерним классом) такой.

- Создается объект, в котором все поля данных имеют значения по умолчанию (нули на двоичном уровне представления).
- Вызывается конструктор дочернего класса.
- Конструктор дочернего класса вызывает конструктор родителя (непосредственного прародителя), а также по цепочке все прародительские конструкторы и инициализации полей, заданных в этих классах, вплоть до класса *Object*.
- Проводится инициализация полей родительской части объекта значениями, заданными в декларации родительского класса.
- Выполняется тело конструктора родительского класса.
- Проводится инициализация полей дочерней части объекта значениями, заданными в декларации дочернего класса.
- Выполняется тело конструктора дочернего класса.

2.3. Наследование

Наследование тут описывается очень просто, через двоеточие указывается класс-предок:

```
class Shape
{
    public Shape()
    {
    }

    public Shape(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    protected int x;
    protected int y;
}

class Circle : Shape
{
```

```
    Circle(int x, int y, int r)
    {
        this.x = x;
        this.y = y;
        this.r = r;
    }

    Circle(int r)
        : base(0, 0)
    {
    }

    private int r;
}
```

Заодно можно видеть использование ключевого слова *base* для вызова конструктора предка (если нет, то вызывается конструктор предка по умолчанию, тот, что без параметров, если он есть, иначе ошибка компиляции), и ключевое слово *protected*, означающее, как и следует ожидать, то, что поле/метод будут видны только потомкам. Наследоваться можно только от одного класса, множественного наследования нет. Почему — множественное наследование создаёт некие трудности само по себе. Например, ситуация, называемая ромбовидным наследованием: есть класс A, в котором объявлен метод a, классы B и C, которые переопределяют метод a, причём каждый по-разному, и класс D, который наследует от B и от C — какую из реализаций метода a он получит? В C++ множественное наследование разрешено, в более современных языках — нет. Зато класс может реализовывать несколько интерфейсов.

2.4. Интерфейсы, абстрактные классы

Интерфейс описывается вот так:

```
interface IDrawable
{
    void Draw();
}
```

И реализуется вот так:

```
class Shape : IDrawable
{
    public Shape()
    {
    }

    void Draw()
    {
        Console.Write("Drawing Shape");
```

```
    }

    protected int x;
    protected int y;
}
```

Можно унаследоваться от одного класса и сколько угодно интерфейсов одновременно. Почему — интерфейс всего лишь декларирует методы, которые должны быть у реализующих его объектов, как объект их реализует — его дело. Собственно, если есть два интерфейса с методами с одинаковыми именами и набором параметров, то они оба могут быть реализованы одним методом, и всё будет ок, так что никакого ромбовидного наследования тут не возникает. Все методы в интерфейсе не могут иметь реализацию, то есть являются абстрактными. Кстати, при реализации интерфейса можно явно указать, метод какого именно интерфейса мы реализуем, как раз если есть два интерфейса с одинаковыми методами и мы хотим сделать две разные реализации:

```
class Shape : IDrawable
{
    public Shape()
    {
    }

    void IDrawable.Draw()
    {
        Console.Write("Drawing Shape");
    }

    protected int x;
    protected int y;
}
```

Теперь метод *Draw()* нельзя вызывать просто от объекта, надо сначала привести его к тому интерфейсу, чей *Draw()* мы реализовали (иначе как компилятор узнает, какой из *Draw()* мы имеем в виду, если их много?):

```
var shape = new Shape();
((IDrawable)shape).Draw();
```

Обычному классу тоже можно сделать абстрактный метод (например, не знаем мы, как рисовать просто фигуру, могли бы сделать у *Shape* метод *Draw* абстрактным), для этого перед методом надо написать ключевое слово *abstract*. Абстрактные методы обязаны быть реализованы во всех неабстрактных потомках, то есть, например, *Circle* обязан реализовать *Draw*, либо же оставить его абстрактным, чтобы его реализовали потомки *Circle*. Класс, у которого есть хоть один абстрактный метод, сам абстрактный, и должен быть помечен ключевым словом *abstract*, так:

```
abstract class Shape
{
```

```
    public Shape()
    {
    }

    public abstract void Draw();

    protected int x;
    protected int y;
}
```

Объекты абстрактных классов создавать, естественно, нельзя, иначе в момент вызова абстрактного метода у объекта такого класса наступит печаль и непонимание, что ему делать. Тем не менее, конструкторы в абстрактных классах — это вполне ок, поскольку они будут вызываться при конструировании потомков.

2.5. Полиморфизм

Польза от абстрактных классов, интерфейсов, да и наследования вообще проявляется при использовании полиморфных вызовов. Как уже говорилось, полиморфизм позволяет в классах-потомках переопределять поведение классов-предков. В C# это делается так: если мы хотим, чтобы метод можно было переопределить, мы помечаем его ключевым словом *virtual*, в потомке, в котором мы его хотим переопределить, объявляем метод с точно такой же сигнатурой (именем и набором параметров), и помечаем его ключевым словом *override*. Пример:

```
class Shape
{
    public virtual void Draw()
    {
        Console.WriteLine($"Drawing Shape with coords ({x}, {y})");
    }

    protected int x;
    protected int y;
}

class Circle : Shape
{
    public Circle(int x, int y, int r)
    {
        this.x = x;
        this.y = y;
        this.r = r;
    }

    public override void Draw()
```

```
    {
        Console.WriteLine($"Drawing Circle with radius {r}");
    }

    private int r;
}

class Rectangle : Shape
{
    public Rectangle(int x, int y, int width, int height)
    {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }

    public override void Draw()
    {
        base.Draw();
        Console.WriteLine($"Drawing Rectangle with width={width} and height={height}");
    }

    protected int width;
    protected int height;
}
```

Тут ещё можно видеть использование ключевого слова *base* для вызова метода предка в переопределённом методе у потомка. Теперь пример того, как этими виртуальными методами пользоваться (собственно, магия полиморфизма):

```
private static void Main(string[] args)
{
    var circle = new Circle(0, 0, 10);
    var rectangle = new Rectangle(0, 0, 10, 10);
    var list = new System.Collections.Generic.List<Shape>();
    list.Add(circle);
    list.Add(rectangle);

    foreach (var shape in list)
    {
        shape.Draw();
    }
}
```

Деталями использования генерика-списка можно пока не грузиться, суть в том, что мы побросали в список шейпов разные фигуры, а потом просто прошлись и отрисовали их

все, не заморачиваясь их типом времени исполнения. Поскольку все фигуры наследуют от *Shape*, то у них у всех всё равно должен быть метод *Draw*, так что его можно вызвать, имея ссылку с типом времени компиляции *Shape*. Обратите внимание, что если мы добавим ещё фигур в нашу иерархию, то код отрисовки менять будет вообще не надо! В этом и есть основное преимущество при использовании объектно-ориентированного подхода, мы можем создавать каркасы приложений, которые расширяются с помощью классов-наследников, тогда сам каркас трогать не надо (он может нам поставляться даже без исходников), а мы просто дописываем немного кода, и всё работает. Именно так устроены средства разработки пользовательских интерфейсов в объектно-ориентированных языках.

Что получится:

```
Drawing Circle with radius 10
Drawing Shape with coords (0, 0)
Drawing Rectangle with width=10 and height=10
Для продолжения нажмите любую клавишу . . .
```

То же самое касается абстрактных методов:

```
abstract class Shape
{
    public abstract void Draw();

    protected int x;
    protected int y;
}

class Circle : Shape
{
    public Circle(int x, int y, int r)
    {
        this.x = x;
        this.y = y;
        this.r = r;
    }

    public override void Draw()
    {
        Console.WriteLine($"Drawing Circle with radius {r}");
    }

    private int r;
}
```

Если *override* не написать, то компилятор выдаст предупреждение — вы действительно объявляете новый метод или пытаетесь переопределить метод класса предка? Если хотите именно новый, это надо явно сказать с помощью немного неоригинального ключевого слова new:

```
class Shape
{
    public virtual void Draw()
    {
        Console.WriteLine($"Drawing Shape with coords ({x}, {y})");
    }

    protected int x;
    protected int y;
}

class Circle : Shape
{
    public Circle(int x, int y, int r)
    {
        this.x = x;
        this.y = y;
        this.r = r;
    }

    public new void Draw()
    {
        Console.WriteLine($"Drawing Circle with radius {r}");
    }

    private int r;
}
```

В чём разница с *override*: при полиморфном вызове вызовется метод предка (ну, типа потомок его не переопределяет, *Circle.Draw* — это другой метод, не имеющий отношения к *Shape.Draw*).

Ещё есть ключевое слово *sealed*, говорящее, что метод нельзя дальше переопределять, или, если оно применено к классу, от класса нельзя наследоваться. Это используется в системных библиотеках, чтобы, например, злоумышленники не унаследовались от строки и не вставили туда код, отправляющий пароли на сервер, после чего подсунули такие строки вместо обычных строк в пользовательскую программу. У вас таких проблем пока быть не должно, так что не буду подробно.

2.6. Модификаторы

Помимо *new* и *sealed* у методов, полей и самих классов могут быть модификаторы видимости. Видимость определяет, откуда можно использовать класс, его метод или поле. Модификаторы видимости в C# такие:

- `public` — применяется к типам и членам (то есть полям, методам, свойствам, событиям), доступ без ограничений;

- `protected` — применяется только к членам, доступ в типе и его потомках;
- `internal` — применяется к типам и членам, доступ только внутри сборки (то есть одного проекта, который собирается в одну .dll-ку или .exe-файл);
- `protected internal` — применяется к типам и членам, доступ внутри сборки **или** в потомках;
- `private` — применяется к типам и членам, доступ только внутри типа;
- по умолчанию для типов `internal`, для членов — `private`.

Бывает ещё модификатор *partial*, который применяется к классам и говорит компилятору, что в этом файле есть только часть описания класса, и может быть ещё один или несколько файлов, определяющих другие члены класса. Используется это в основном для того, чтобы один класс мог содержать рукописные и автоматически сгенерированные части и чтобы при перегенерации рукописный код не затирался (например, так работает дизайнер пользовательских интерфейсов). Использовать *partial* для логической группировки полей и методов — плохая идея, если возникла такая потребность, вам, скорее всего, нужно просто несколько разных классов.

2.7. Вложенные классы

Классы могут содержать внутри себя описания типов, в частности, других классов (уровней такой вложенности может быть много). Например,

```
class Circle
{
    private readonly Point pos;
    private readonly int r;

    private class Point
    {
        public int x;
        public int y;
    }

    public Circle(int x, int y, int r)
    {
        pos = new Point {x = 10, y = 10};
        this.r = r;
    }

    public void Draw() =>
        Console.WriteLine($"({pos.x}, {pos.y}), radius {r}");
}
```

Тут класс *Point* объявлен внутри класса *Circle* как *private* вложенный класс, поэтому доступен он только внутри класса *Circle* (где и используется как тип одного из полей). Полезно для обеспечения инкапсуляции.

2.8. Приведение типов

Есть ещё явное и неявное приведение типов. От потомка к предку преобразование возможно всегда (потому что потомок и так является предком, да), так что выполняется автоматически. От предка к потомку каст (преобразование, от англ. cast) не всегда возможен: если *Circle* всегда *Shape*, то *Shape* может оказаться либо *Circle*, либо *Rectangle*. Однако если мы уверены относительно типа времени выполнения, мы можем сделать явный каст вручную:

```
Shape shape = new Circle();
Circle circle = (Circle)shape;
```

Если мы неправы, программа упадёт с исключением. Есть оператор *as*, который делает то же самое, но в случае неудачи тихо возвращает *null*:

```
Shape shape = new Circle();
Circle circle = shape as Circle;
```

Есть ещё оператор проверки типа времени выполнения *is*:

```
Shape shape = new Circle();
if (shape is Circle)
{
    Circle circle = (Circle)shape;
}
```

Или даже так, сразу же с объявлением переменной приведённого типа:

```
Shape shape = new Circle();
if (shape is Circle circle)
{
    ...
}
```

Эту же конструкцию можно использовать внутри *switch*, чтобы делать что-нибудь в зависимости от типа переменной:

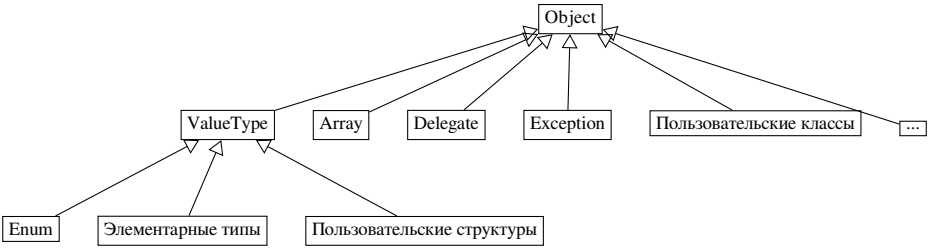
```
switch (shape)
{
    case Circle c:
        WriteLine($"circle with radius {c.Radius}");
        break;
    case Rectangle s when (s.Length == s.Height):
        WriteLine($"{s.Length} x {s.Height} square");
        break;
    case Rectangle r:
        WriteLine($"{r.Length} x {r.Height} rectangle");
        break;
```

```
    default:
        WriteLine("<unknown shape>");
        break;
    case null:
        throw new ArgumentNullException(nameof(shape));
}
```

Каст от предка к потомку называется понижающим кастом, каст от потомка к предку называется повышающим кастом. Использование понижающих кастов свидетельствует о непродуманной архитектуре системы — если мы точно знаем тип времени выполнения переменной, то почему бы не сделать его типом времени компиляции? Однако же, в совокупности с оператором *is* он бывает весьма полезен, и всё-таки используется в промышленном коде.

2.9. System.Object

Корнем иерархии наследования в C# является класс *System.Object*. От него наследуются все типы в языке вообще, не только классы. Вот иерархия наследования основных типов в C#:



Тип *ValueType* имеет специальное значение для компилятора — все его наследники считаются типами-значениями. Впрочем, отнаследоваться от него вручную не получится, это ошибка компиляции. Фактически, наследование от него — это объявление своей структуры. От *ValueType* же наследуются все enum-ы и все элементарные типы (*int*, *double*, *bool* и т.д.). Все пользовательские классы напрямую или опосредованно наследуются от *System.Object*.

В самом *System.Object* определено всего шесть методов и два *private*-поля. Методы такие:

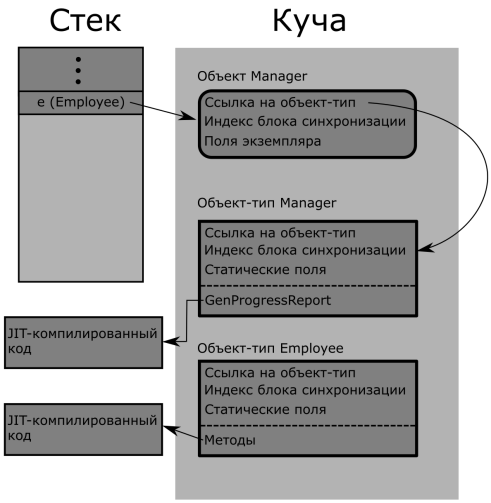
- *Equals* — виртуальный метод, предназначенный для сравнения объектов; напомню, что оператор «==» по умолчанию сравнивает адреса объектов в памяти, Equals как раз нужен для того, чтобы можно было сравнивать объекты по значениям их полей;
- *GetHashCode* — виртуальный метод, возвращающий хеш-значение для объекта в виде целого числа (любого), реализация по умолчанию возвращает просто адрес объекта для ссылочных типов и значение для типов-значений;
- *ToString* — виртуальный метод, который должен возвращать строковое представление объекта, для удобства отладки и логирования;

- *GetType* — невиртуальный метод, возвращающий объект-тип объекта, нужен для рефлексии;
- *MemberwiseClone* — невиртуальный защищённый метод, создающий копию объекта путём копирования всех его полей, нужен для сериализации/десериализации;
  - обратите внимание, этот метод создаёт объект, не вызывая конструктор;
- *Finalize* — виртуальный защищённый метод, говорящий, что делать, когда объект собирается удалить сборщик мусора.

В памяти каждый объект представляется следующим образом. Положим, у нас есть код

```
void Example()
{
    Employee e =
        new Manager();
    e.GenProgressReport();
}
```

Тогда в памяти всё будет выглядеть так:



На стеке вызовов находится ссылка на объект, сам объект лежит на куче. В участке памяти на куче назодится ссылка на объект-тип (она есть у всех объектов, даже у объектов-типов, такие дела), индекс блока синхронизации (штука, нужная для многопоточного программирования, это просто число) и значения полей объекта. В объекте-типе помимо всего вышеперечисленного лежат ещё статические поля класса, к которому принадлежит наш объект, и ссылки на скомпилированный код всех его методов.