* [ОБЗОР КУРСА](https://lyceum.yandex.ru/courses/123/groups/568)

[Урок Наследование №1](https://lyceum.yandex.ru/courses/123/groups/568/lessons/674)

**ООП. Наследование**

**План урока**

1

[Введение](https://lyceum.yandex.ru/courses/123/groups/568/lessons/674/materials/1714#1)

2

[Иерархия классов](https://lyceum.yandex.ru/courses/123/groups/568/lessons/674/materials/1714#2)

3

[Механизм наследования](https://lyceum.yandex.ru/courses/123/groups/568/lessons/674/materials/1714#3)

4

[Наследование на примере геометрических фигур](https://lyceum.yandex.ru/courses/123/groups/568/lessons/674/materials/1714#4)

**Аннотация**

*В этом уроке объясняется понятие наследования, показывается его связь с уже известными понятиями объектно-ориентированного программирования (инкапсуляция и полиморфизм), описывается синтаксис и семантика наследования в Python.*

**1. Введение**

На прошлых уроках мы познакомились с основами объектно-ориентированного программирования. Научились определять классы и создавать объекты. Узнали о том, что хорошо продуманный набор методов (иначе говоря, **интерфейс**) позволяет добиться **инкапсуляции** (т. е. скрытия информации о внутреннем устройстве объекта) и **полиморфизма** (т.е. возможности писать код, работающий одинаково с разными классами).

Часто бывает так, что классы в программе имеют не только общий интерфейс, но и похожую реализацию. Для примера рассмотрим несколько классов, представляющих геометрические фигуры: круг (**Circle**) и прямоугольник (**Rectangle**). Пусть интерфейс этих классов пока состоит из единственного метода **area()**, возвращающего площадь фигуры.

**from** math **import** pi

**class** Circle:

**def** \_\_init\_\_(self, radius):

self.r = radius

**def** area(self):

**return** pi \* self.r \*\* 2

**class** Rectangle:

**def** \_\_init\_\_(self, a, b):

self.a = a

self.b = b

**def** area(self):

**return** self.a \* self.b

Допустим, что теперь нам нужно реализовать класс для квадрата (**Square**). Конечно, мы можем сделать это непосредственно:

**class** Square:

**def** \_\_init\_\_(self, a):

self.a = a

**def** area(self):

**return** self.a \* self.a

Но мы знаем, что квадрат — это частный случай прямоугольника, у которого ширина равна высоте. Чем нам это может помочь?

Представьте, что мы хотим добавить в интерфейс наших классов ещё один метод **perimeter()** для вычисления периметра. Сейчас нам нужно добавить его во все три класса — но это лишняя работа, ведь периметр квадрата вычисляется так же, как и периметр прямоугольника. А любая лишняя работа не только отнимает время у программиста, но и увеличивает вероятность допустить ошибку. Конечно, вы вряд ли ошибетесь в написании метода **perimeter()** для квадрата, ведь это займет одну строку; но в реальности методы классов могут занимать десятки (иногда даже сотни) строк, а логика работы этих методов гораздо сложнее.

Резюмируем наши наблюдения:

* Класс **Square** является частным случаем класса **Rectangle**;
* Если бы имелся способ явно запрограммировать это отношение, то наш код получился более коротким и, что важно, **согласованным** (если периметр прямоугольника вычисляется правильно, то периметр квадрата **автоматически** вычисляется правильно).

Такой способ, конечно, существует — он называется «наследование».

**Наследование**

**Наследование** — это механизм, позволяющий запрограммировать отношение вида «класс B является частным случаем класса A». В этом случае класс A также называется **базовым** классом, а B — **производным** классом.

Наследование является способом переиспользования кода между классами без необходимости нарушения инкапсуляции. Это достигается за счёт того, что производный класс может пользоваться атрибутами и методами базового класса (иными словами, прозводный класс «наследует» атрибуты и методы базового класса).

Наследование — важная концепция объектно-ориентированного программирования наряду с инкапсуляцией и полиморфизмом.

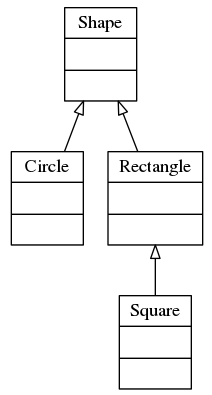
**2. Иерархия классов**

Используя отношение «частный случай», можно строить иерархии классов. Добавим к нашим классам ещё один класс, представляющий произвольную геометрическую фигуру (**Shape**):

**class** Shape:

**pass**

Тогда иерархия классов будет выглядеть так:



В этом и следующем уроках мы узнаем, как отразить эту иерархию в коде.

Чтобы наследовать класс B от класса A (то есть, чтобы запрограммировать отношение «класс B является частным случаем класса A»), нужно написать так:

**class** A:

**pass**

**class** B(A):

**pass**

**3. Механизм наследования**

**Вопрос**: Что происходит, когда класс B наследуется от класса A (т.е. когда мы пишем class B(A))?

**Ответ**: Меняется процедура поиска методов и атрибутов в классе B.

Разберём, как именно.

Мы знаем, что у «простого» объекта (который ни от кого не наследуется) нельзя вызвать несуществующий метод или прочитать несуществующий атрибут:

**class** C:

**def** foo(self):

**print**('foo')

c = C()

c.foo() *# ok*

c.bar() *# error*

------------------------------------------

AttributeError Traceback (most recent call last)

<ipython-input-4-de63cbd4e47c> in <module>()

6 c = C()

7 c.foo() # ok

----> 8 c.bar() # error

AttributeError: 'C' object has no attribute 'bar'

**Важно**

Если класс наследован от другого класса, то проверка существования метода (или атрибута) осуществляется так:

* сперва метод ищется в исходном (производном) классе;
* если его там нет, он ищется в базовом классе;
* предыдущие шаги повторяются до тех пор, пока метод не будет найден, или пока процедура не дойдет до класса, который ни от кого не наследуется.

Это означает, что производному классу доступны не только собственные методы, но и методы базового класса. В этом случае говорят, что производный класс «наследует» методы базового класса.

**class** BaseC:

**def** bar(self):

**print**('bar')

**class** C(BaseC):

**def** foo(self):

**print**('foo')

c = C()

c.foo() *# ok -- этот метод есть в производном классе*

c.bar() *# ok -- этот метод есть в базовом классе*

c.baz() *# error -- этого метода нет ни в производном, ни в базовом классе*

Как видим, в производные классы можно не только добавлять новые методы, но и пользоваться методами базового класса. В этом и состоит польза наследования методов — оно позволяет обойтись без дублирования кода и при этом не нарушает инкапсуляцию.

**4. Наследование на примере геометрических фигур**

Рассмотрим механизм наследования подробнее на примере нашей иерархии геометрических фигур.

Для начала исследуем, как производные классы могут пользоваться методами базового класса. Для этого реализуем в классе **Shape** метод **describe**, который будет печатать название собственного класса:

**class** Shape:

**def** describe(self):

*# Атрибут \_\_class\_\_ содержит класс или тип объекта self*

*# Атрибут \_\_name\_\_ содержит строку,*

*# в которой написано название класса или типа*

**print**("Класс: {}".format(self.\_\_class\_\_.\_\_name\_\_))

Наследуем от **Shape** классы **Circle** и **Rectangle** и убедимся, что метод **describe** работает и для них тоже:

**from** math **import** pi

**class** Circle(Shape):

**def** \_\_init\_\_(self, radius):

self.r = radius

**def** area(self):

**return** pi \* self.r \*\* 2

**class** Rectangle(Shape):

**def** \_\_init\_\_(self, a, b):

self.a = a

self.b = b

**def** area(self):

**return** self.a \* self.b

shape = Shape()

shape.describe()

circle = Circle(1)

circle.describe()

rectangle = Rectangle(1, 2)

rectangle.describe()

------------------------------------------

Класс: Shape

Класс: Circle

Класс: Rectangle

Теперь рассмотрим, как добавить в производный класс новый метод, которого нет в базовом классе.

А именно, добавим в класс **Circle** метод **square**, который решает знаменитую задачу **квадратуры круга** — возвращает квадрат (в нашем случае — объект **Rectangle** с равными сторонами), который по площади равен площади исходного круга.

Математики много столетий бились над задачей [квадратуры круга](https://ru.wikipedia.org/wiki/%D0%9A%D0%B2%D0%B0%D0%B4%D1%80%D0%B0%D1%82%D1%83%D1%80%D0%B0_%D0%BA%D1%80%D1%83%D0%B3%D0%B0) и в итоге доказали, что такое построение нельзя выполнить с помощью циркуля и линейки.

Зато это можно сделать с помощью Python:

**class** Circle(Shape):

**def** \_\_init\_\_(self, radius):

self.r = radius

**def** area(self):

**return** pi \* self.r \*\* 2

**def** square(self):

side = pi \*\* 0.5 \* self.r

**return** Rectangle(side, side)

circle = Circle(1)

square = circle.square()

**print**("Площадь круга: {}".format(circle.area()))

**print**("Площадь квадрата: {}".format(square.area()))

**print**("Радиус круга: {}".format(circle.r))

**print**("Длина стороны квадрата: {}".format(square.a))

------------------------------------------

Площадь круга: 3.141592653589793

Площадь квадрата: 3.1415926535897927

Радиус круга: 1

Длина стороны квадрата: 1.7724538509055159

[Справка](https://yandex.ru/support/lyceum-students)

Исключительное право на учебную программу и все сопутствующие ей учебные материалы, доступные в рамках проекта «Яндекс.Лицей», принадлежат АНО ДПО «ШАД». Воспроизведение, копирование, распространение и иное использование программы и материалов допустимо только с предварительного письменного согласия АНО ДПО «ШАД».

© 2018 – 2020  ООО «[Яндекс](https://yandex.ru/)»

Чаты