

# Object Oriented Programming behind the scenes with diagrams!

Hi ! You should be really proud of yourself, you've come a long way from week 1, and now we are going to venture into the world of **Object Oriented Programming (OOP)**. (This post will cover content from lectures in week 5, including instances and Hierarchies)

First, let's discuss **Classes and Instances**:

A class is like a stencil or blueprint that represents something that has attributes and contains functions (methods). We will look at an example in the post.

**This post will be based on an example to make these abstract concepts come to life!**

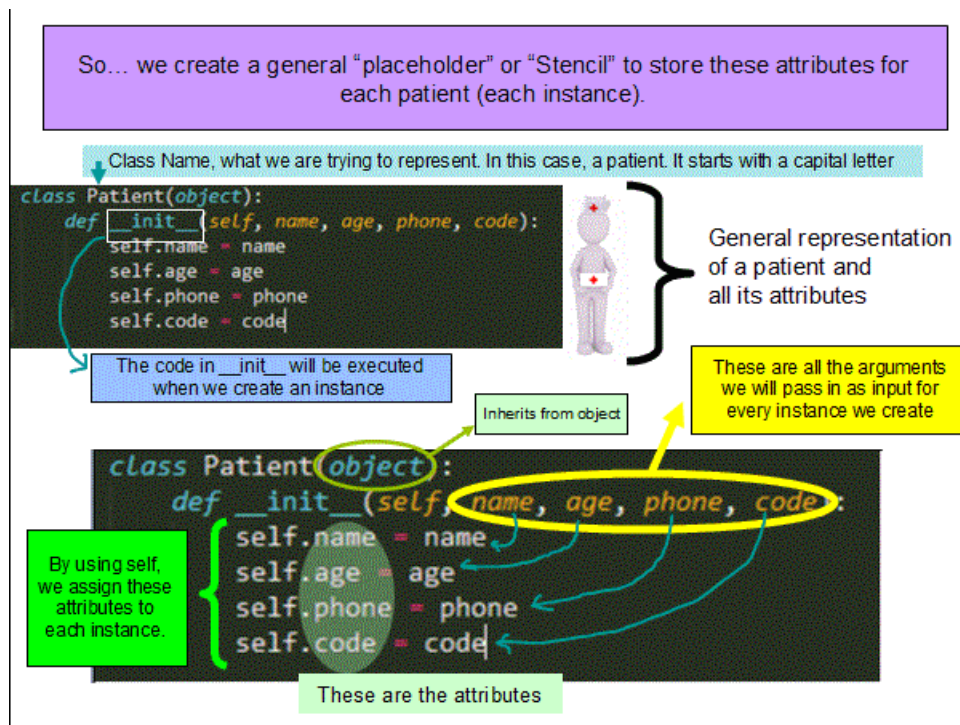
If we want to write code to manage a medical facility, we will need to represent a **Patient**. But... how can we represent the abstract notion of a patient in a general way? Well, all patients are humans, so they share common attributes and perform common actions. How can we do this?? Let me think... **We can represent a patient by using a Class!**

You've learned the syntax for defining a class. We use the keyword `class` followed by the class name (`Patient`) with a capital letter and, in parenthesis we define the inheritance (if it's the first one in the chain, if it doesn't inherit from any custom class, we write `object`). Next, we write a colon. Below this line, we indent our code so python's interpreter knows it belongs to this class definition.

The first thing we will find is the `def` keyword and an `__init__()` method. This is a function that will run every time we create an instance (in just a second we will see what instances are). Next, we have parentheses and inside them we find the keyword `self`, a comma and as many parameters or attributes as we need. Next, we write a colon, indent the code below and add this:

```
self.attribute_name = parameter_name for as many attributes as we need.
```

In this case, a patient has a name, an age, a phone number and a registration code. We can add as many as we need, but for demonstration purposes, let's use only 4.



By: kiara-elizabeth

## Instances

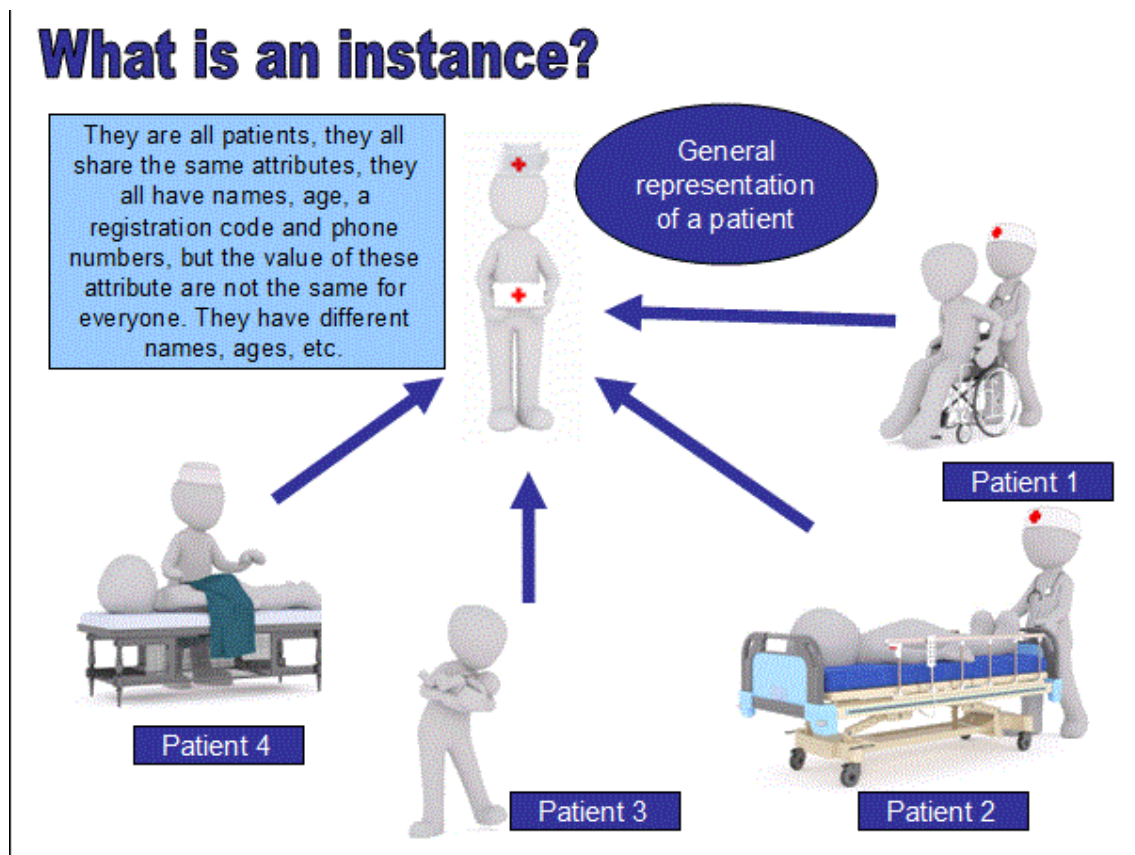
Now (like I promised ;-) ) we will learn about **INSTANCES!**

At first, the concept of an instance might seem very abstract, but let's keep the example of a patient to illustrate this concept.

We've written a class that is a general representation of a patient. It has attributes that all patients share, but they don't actually have values yet, not until we create instances.

If we have the general representation of a patient, every individual patient is an instance of that general notion of a patient. Let me explain... in the diagram below you can see 4 patients. Each one of them has his/her own name, age, phone, and code. We can't expect all of them to have the same values for the attributes we've defined in our general blueprint.

By creating instances, we are using the blueprint to fill the data for each patient in a consistent way, following the same structure, and asking for values for the same parameters in the same order.



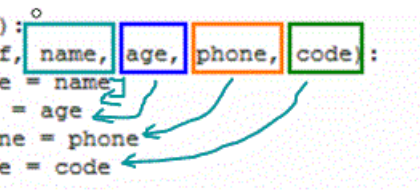
Here you can see how we create an instance of a patient.

```
instance_name = Class_name(parameters)
```

In this case, our new instance is patient1 and his/her attributes are in parentheses in the same order as we defined them in our class, our general blueprint.

```
class Patient(object):
    def __init__(self, name, age, phone, code):
        self.name = name
        self.age = age
        self.phone = phone
        self.code = code
```

```
>>> class Patient(object):
      def __init__(self, name, age, phone, code):
          self.name = name
          self.age = age
          self.phone = phone
          self.code = code
```



```
>>> patient1 = Patient('Clara', 25, 555-5555, 2363)
```







Our new instance patient1

Is a patient whose name is Clara, age is 25...

By: kiara-elizabeth

This way, we can create as many instances as we'd like. In this case, we create 4 instances, one for each patient with their own data.

<pre>def __init__(self, name, age, phone, code):</pre> <pre>&gt;&gt;&gt; patient1 = Patient('Clara', 25, 555-5555, 2363)</pre>  <table border="1"> <tr><td>name</td><td>'Clara'</td></tr> <tr><td>age</td><td>25</td></tr> <tr><td>phone</td><td>555-5555</td></tr> <tr><td>code</td><td>2363</td></tr> </table> <p>Patient 1</p>	name	'Clara'	age	25	phone	555-5555	code	2363	<pre>def __init__(self, name, age, phone, code):</pre> <pre>&gt;&gt;&gt; patient2 = Patient('Steven', 45, 134-2344, 2445)</pre>  <table border="1"> <tr><td>name</td><td>'Steven'</td></tr> <tr><td>age</td><td>45</td></tr> <tr><td>phone</td><td>134-2344</td></tr> <tr><td>code</td><td>2445</td></tr> </table> <p>Patient 2</p>	name	'Steven'	age	45	phone	134-2344	code	2445
name	'Clara'																
age	25																
phone	555-5555																
code	2363																
name	'Steven'																
age	45																
phone	134-2344																
code	2445																
<pre>def __init__(self, name, age, phone, code):</pre> <pre>&gt;&gt;&gt; patient3 = Patient('Noris', 1, 778-4563, 2355)</pre>  <table border="1"> <tr><td>name</td><td>'Noris'</td></tr> <tr><td>age</td><td>1</td></tr> <tr><td>phone</td><td>778-4563</td></tr> <tr><td>code</td><td>2355</td></tr> </table> <p>Patient 3</p>	name	'Noris'	age	1	phone	778-4563	code	2355	<pre>def __init__(self, name, age, phone, code):</pre> <pre>&gt;&gt;&gt; patient4 = Patient('Jacob', 67, 234-3553, 2342)</pre>  <table border="1"> <tr><td>name</td><td>'Jacob'</td></tr> <tr><td>age</td><td>67</td></tr> <tr><td>phone</td><td>324-3553</td></tr> <tr><td>code</td><td>2342</td></tr> </table> <p>Patient 4</p>	name	'Jacob'	age	67	phone	324-3553	code	2342
name	'Noris'																
age	1																
phone	778-4563																
code	2355																
name	'Jacob'																
age	67																
phone	324-3553																
code	2342																

By: kiara-elizabeth

## Methods

Now that we presented what instances are, we can dive into Methods.

Methods are functions contained in a class. They are related to the class. In this case, we know patients walk and breath, so we define these methods in our general blueprint so all of our instances "are able to breath and walk" (actually, these functions will print the average human walking speed and respiratory rate).

# What is a method?

```
class Patient(object):
    def __init__(self, name, age, phone, code):
        self.name = name
        self.age = age
        self.phone = phone
        self.code = code

    # A method is a function related to a class. In this case, we have a method
    # that prints the normal respiratory rate for an adult and a method that prints
    # the average human walking speed
    def breath(self):
        print('I breath 12-18 times per minute')

    def walk(self):
        print('I walk at 5 km per hour')

>>> patient1 = Patient('Noris', 1, 778-4563, 2355)
>>> patient1.breath()
I breath 12-18 times per minute
>>> patient1.walk()
I walk at 5 km per hour
>>>
>>> patient2 = Patient('Clara', 25, 555-5555, 2363)
>>> patient2.breath()
I breath 12-18 times per minute
>>> patient2.walk()
I walk at 5 km per hour
```

**TO CALL A METHOD:** instance\_name.method\_name(parameters)

Perhaps you've noticed that the method above only has `self` in parentheses. We can set formal parameters for methods as well, by writing a comma next to the keyword `self` and adding as many parameters as we need to use inside our function separated by commas.

In the picture of the shell below you can see that it now asks for a parameter to execute the function correctly.

## Method with parameters

```
class Patient(object):
    def __init__(self, name, age, phone, code):
        self.name = name
        self.age = age
        self.phone = phone
        self.code = code

    # A method is a function related to a class. In this case, we have a method
    # that prints the normal respiratory rate for an adult and a method that prints
    # the average human walking speed
    def breath(self):
        print('I breath 12-18 times per minute')

    def walk(self, walkRate):
        print('I walk at ' + str(walkRate) + ' km per hour')

>>> patient1 = Patient('Noris', 1, 545-4545, 3456)
>>>
>>> patient1.breath()
I breath 12-18 times per minute
>>> patient1.walk()
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    patient1.walk()
TypeError: walk() missing 1 required positional argument: 'walkRate'
>>> patient1.walk(56)
I walk at 56 km per hour
```

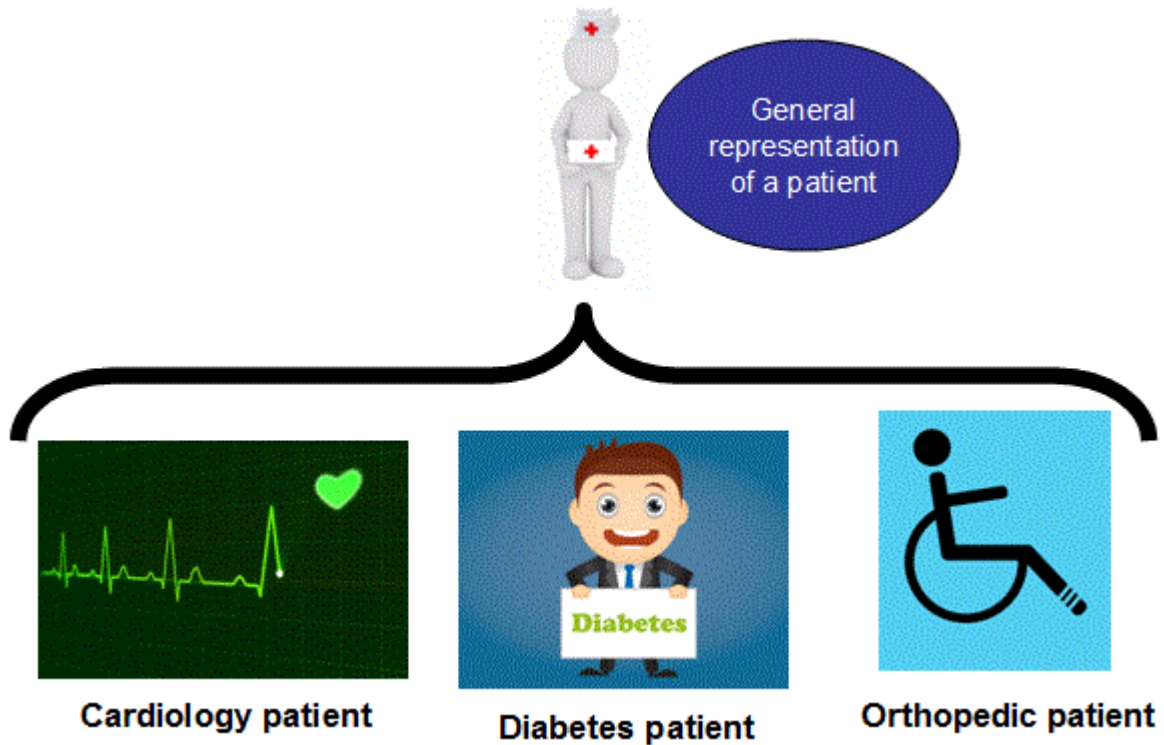
By: kiara-elizabeth

## Inheritance

Let's analyze what happens if our general representation might not model every situation as precisely as we would like. If we need to have custom attributes for a particular type of patient, this general representation might not work so well. What if we need to define a cardiology patient and add a method to check his/her heart rate? Or a diabetic patient and add a blood sugar attribute?

**This is where inheritance comes into play.**

All cardiac, diabetic and orthopedic patients are patients, and share the same attributes as the general class, but they need their own custom structure (methods and attributes)





We define their very own class, but to avoid repeating code we've already written for the general class, we use this code:

```
class Patient(object):
    def __init__(self, name, age, phone, code):
        self.name = name
        self.age = age
        self.phone = phone
        self.code = code

    # A method is a function related to a class. In this case, we have a method
    # that prints the normal respiratory rate for an adult and a method that prints
    # the average human walking speed
    def breath(self):
        print('I breath 12-18 times per minute')

    def walk(self, walkRate):
        print('I walk at ' + str(walkRate) + ' km per hour')
```

```
class CardioPatient(Patient):
    def __init__(self, name, age, phone, code, heartRate):
        Patient.__init__(self, name, age, phone, code)
        self.heartRate = heartRate


    def getHeartRate(self):
        print('My heart rate is: ', self.heartRate)

    def setHeartRate(self, heartRate):
        self.heartRate = heartRate
```

```
class DiabetesPatient(Patient):
    def __init__(self, name, age, phone, code, bloodSugar):
        Patient.__init__(self, name, age, phone, code)
        self.bloodSugar = bloodSugar

    def getBloodSugar(self):
        print('My blood sugar is: ', self.bloodSugar)




    def setBloodSugar(self, bloodSugar):
        self.bloodSugar = bloodSugar
```



```
class OrthopedicPatient(Patient):
    def __init__(self, name, age, phone, code, pain):
        Patient.__init__(self, name, age, phone, code)
        self.pain = pain

    def getPainLevel(self):
        print('My pain is: ', self.pain)

    def setPainLevel(self, pain):
        self.pain = pain
```

By: kiara-elizabeth



Let's do a closeup on the cardioPatient definition. Take a look at the blue rectangle. This line gives the cardioPatient all the attributes a general patient has. And since we've set Patient in parentheses next to the initial cardioPatient class definition, every instance of our cardioPatient class will have access to the Patient class' methods. A new attribute has also been added to illustrate how we can customize these blueprints to make them fit our needs.

```

class Patient(object):
    def __init__(self, name, age, phone, code):
        self.name = name
        self.age = age
        self.phone = phone
        self.code = code

    # A method is a function related to a class. In this case, we have a method
    # that prints the normal respiratory rate for an adult and a method that prints
    # the average human walking speed
    def breath(self):
        print('I breath 12-18 times per minute')

    def walk(self, walkRate):
        print('I walk at ' + str(walkRate) + ' km per hour')

class CardioPatient(Patient):
    def __init__(self, name, age, phone, code, heartRate):
        Patient.__init__(self, name, age, phone, code)
        self.heartRate = heartRate

    def getHeartRate(self):
        print('My heart rate is: ', self.heartRate)

    def setHeartRate(self, heartRate):
        self.heartRate = heartRate
  
```

By: kiara-elizabeth

```

>>> cardioPatient1 = CardioPatient('Noris', 25, 456-3456, 2345, 60)
>>> cardioPatient2 = CardioPatient('Ana', 57, 465-5677, 2305, 80)

>>> diabetesPatient1 = DiabetesPatient('Gino', 13, 456-3567, 2356, 120)
>>> diabetesPatient2 = DiabetesPatient('Maria', 27, 456-3523, 1356, 100)

>>> orthoPatient1 = OrthopedicPatient('Brickert', 90, 152-2743, 480, 'high')
>>> orthoPatient2 = OrthopedicPatient('jason', 90, 157-2343, 3467, 'low')
  
```

Now we can access the attributes and methods in these classes that inherit from Patient. They will have all the attributes and can access all methods in Patient, but they will have their own attributes and methods as well. This is a great way to implement "Do not repeat yourself" since we are reusing code with this patient "template"

One last thing... Orthopedic patients may not walk like a general representation of a patient, so we would need to overwrite the method walk in Patient class. This is done by writing a method walk in the orthopedicPatient class. If we have two methods that are named the same, the method that is lowest in the chain will be executed. In this case, we have a child orthopedicPatient that inherits from Patient, so the method found first will be the one in orthopedicPatient, as you can see in the shell.

## Overwriting the Parent's method

```
class Patient(object):
    def __init__(self, name, age, phone, code):
        self.name = name
        self.age = age
        self.phone = phone
        self.code = code

    # A method is a function related to a class. In this case, we have a method
    # that prints the normal respiratory rate for an adult and a method that prints
    # the average human walking speed
    def breath(self):
        print('I breath 12-18 times per minute')

    def walk(self, walkRate):
        print('I walk at ' + str(walkRate) + ' km per hour')
```

```
class OrthopedicPatient(Patient):
    def __init__(self, name, age, phone, code, pain):
        Patient.__init__(self, name, age, phone, code)
        self.pain = pain
```

```
    def getPainLevel(self):
        print('My pain is: ', self.pain)
```

```
    def setPainLevel(self, pain):
        self.pain = pain
```

```
    def walk(self):
        print("I am an orthopedic patient, currently I can't walk")
```

```
>>> orthoPatient1 = OrthopedicPatient('Brickert', 90, 152-2743, 480, 'high')
>>> orthoPatient1.walk()
I am an orthopedic patient, currently I can't walk
```

Orthopedic patients won't have the same walk rate as our general representation of a patient, we need to create a new walk method for orthopedic patients!



If two methods in the chain have the same name, the child's method will overwrite the parent's method.

By: kiara-elizabeth

I hope it helps! If you have any questions please ask the on the forums!

Estefania (kiara-elizabeth).

Attributions: -All complementary images used to make the diagrams (patients, ecg, diabetes and orthopedic patient) were taken from [pixabay.com](https://pixabay.com), licensed under public domain