Design Patterns - Python OOP Refresher

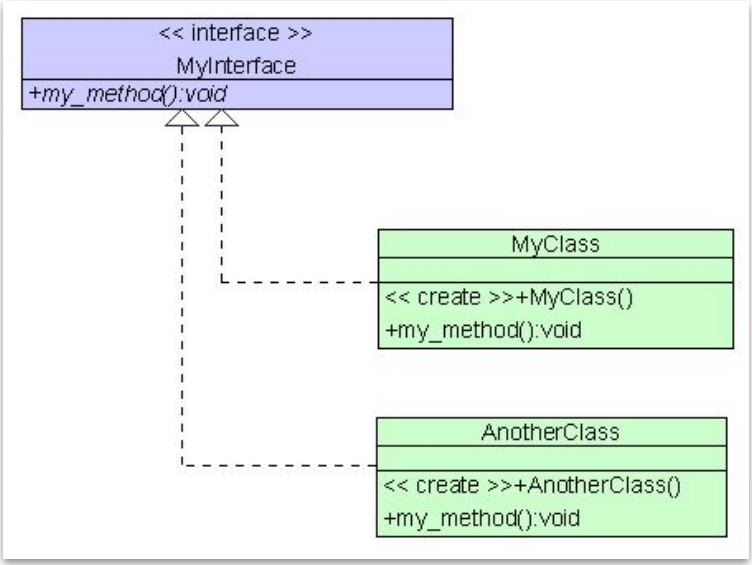
Inheritance: Interfaces

Let's talk about **contracts**.

1. A contract is like a **promise** that you will provide some specific behaviour. In classes this means that you **'promise'** to provide some functionality.
2. One way to create a contract is through a concept of **Interface**.
   a. In **Python** we do not have a strict interface.
   b. We use an **abstract class** with no implementation.

```
# imports needed for abstract classes
from abc import ABC, abstractmethod

# interface contract. Children will have to implement
# all of the abstract methods. In an interface methods
# have no implementations so we use 'pass'
class MyInterface(ABC):
  @abstractmethod
  def my_method(self):
    pass

class MyClass(MyInterface):
  def my_method(self):
    print("my_method implementation in MyClass")

class AnotherClass(MyInterface):
  def my_method(self):
    print("my_method implementation in AnotherClass")

my_obj = MyClass()
my_obj.my_method()

another_obj = AnotherClass()
another_obj.my_method()
```

Design Patterns - Python OOP Refresher




```python
# imports needed for abstract classes
from abc import ABC, abstractmethod

# interface contract. Children will have to implement
# all of the abstract methods. In an interface methods
# have no implementations so we use 'pass'
class MyInterface(ABC):
  @abstractmethod
  def my_method(self):
    pass

class MyClass(MyInterface):
  def my_method(self):
    print("my_method implementation in MyClass")

class AnotherClass(MyInterface):
  def my_method(self):
    print("my_method implementation in AnotherClass")

my_obj = MyClass()
my_obj.my_method()

another_obj = AnotherClass()
another_obj.my_method()
```

Design Patterns - Python OOP Refresher

Inheritance: Interfaces

How do we enforce contracts?

We can use **type hinting** in python to enforce a contract.

```python
# imports needed for abstract classes
from abc import ABC, abstractmethod

# interface contract. Children will have to
# implement all of the abstract methods.
# Methods have no implementations: use 'pass'
class MyInterface(ABC):
  @abstractmethod
  def my_method(self):
    pass
```

```python
class MyClass(MyInterface):
  def my_method(self):
    print("my_method implementation in MyClass")

class NotImplementingInterface:
  def some_method(self):
    print("I am not implementing MyInterface")

# This method expects to be passed an object that
# implements the MyInterface methods.
def process_my_interface(obj: MyInterface):
  obj.my_method()
  print("Correct implementation of MyInterface")

my_obj = MyClass()
process_my_interface(my_obj)

bad_interface = NotImplementingInterface()
process_my_interface(bad_interface)
```

Design Patterns - Python OOP Refresher

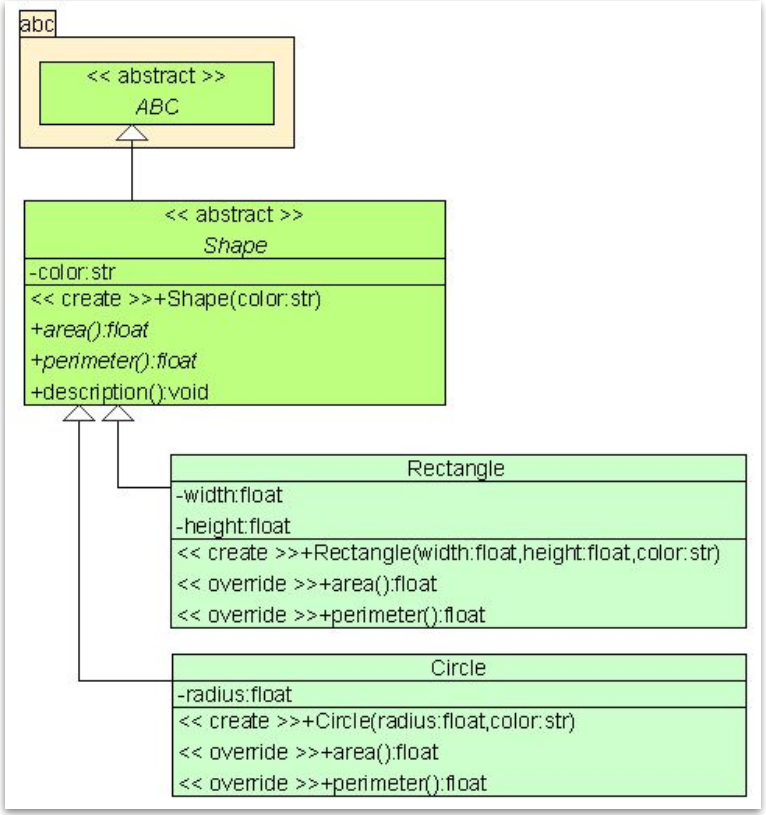
Inheritance: Abstract Classes

Let's talk about partially implemented **contracts** which is about **re-use.**

1. In many cases you have actual functionality not just a signature but actual functionality behind the signature that you would like to have all child classes inherit.
2. This is where you would use an Abstract class.
   a. You can add specific implemented methods
   b. Specific constants or variables.

```python
from abc import ABC, abstractmethod

# Define an abstract class 'Shape'
class Shape(ABC):
  def __init__(self, color):
      self.color = color

  @abstractmethod
  def area(self):
      pass

  @abstractmethod
  def perimeter(self):
      Pass

  def description(self):
    print(f"{self.__class__.__name__} has the
          color: {self.color}")
```

Design Patterns - Python OOP Refresher




```python
from abc import ABC, abstractmethod

# Define an abstract class 'Shape'
class Shape(ABC):
  def __init__(self, color):
      self.color = color

  @abstractmethod
  def area(self):
      pass

  @abstractmethod
  def perimeter(self):
      pass

  def description(self):
    print(f"{self.__class__.__name__} has the
          color: {self.color}")
```

Design Patterns - Python OOP Refresher

Inheritance: Abstract classes

Abstract classes can re-implement methods or keep the existing methods.

```
class Shape(ABC):
    def __init__(self, color):
        self.color = color

    @abstractmethod
    def area(self):
        pass

    @abstractmethod
    def perimeter(self):
        pass

  def description(self):
    print(f"{self.__class__.__name__} has the
          color: {self.color}")
```

```
# Define a concrete class 'Rectangle' that
# inherits from 'Shape'
class Rectangle(Shape):
    def __init__(self, width, height, color):
        super().__init__(color)
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

    def perimeter(self):
        return 2 * (self.width + self.height)

# Create instances of concrete classes and use
# their methods
rectangle = Rectangle(4, 5, "red")
print(f"Rectangle area: {rectangle.area()}")
print(f"Rectangle perimeter:
{rectangle.perimeter()}")
print(f"Rectangle color: {rectangle.color}")
rectangle.description()
```