```
class fraction(object):

    def __init__(self, n, d):
        self.numerator, self.denominator = fraction.reduce(n, d)


    @staticmethod
    def gcd(a,b):
        while b != 0:
            a, b = b, a%b
        return a

    @classmethod
    def reduce(cls, n1, n2):
        g = cls.gcd(n1, n2)
        return (n1 // g, n2 // g)

    def __str__(self):
        return str(self.numerator)+'/'+str(self.denominator)
```

Using this class:

```
from fraction1 import fraction
x = fraction(8,24)
print(x)
```

```
1/3
```

CLASS METHODS VS. STATIC METHODS AND INSTANCE METHODS

Our last example will demonstrate the usefulness of class methods in inheritance. We define a class "Pet" with a method "about". This method should give some general class information. The class Cats will be inherited in a subclass "Dog" and "Cat". The method "about" will be inherited as well. We will demonstrate that we will encounter problems, if we define the method "about" as a normal instance method or as a static method. We will start by defining "about" as an instance method:

```
class Pet:
    _class_info = "pet animals"

    def about(self):
        print("This class is about " + self._class_info + "!")


class Dog(Pet):
    _class_info = "man's best friends"

class Cat(Pet):
    _class_info = "all kinds of cats"

p = Pet()
p.about()
d = Dog()
d.about()
c = Cat()
c.about()
```

```
This class is about pet animals!
This class is about man's best friends!
This class is about all kinds of cats!
```

This may look alright at first at first glance. On second thought we recognize the awful design. We had to create instances of the Pet, Dog and Cat classes to be able to ask what the class is about. It would be a lot better, if we could just write "Pet.about()", "Dog.about()" and "Cat.about()" to get the previous result. We cannot do this. We will have to write "Pet.about(p)", "Dog.about(d)" and "Cat.about(c)" instead.

Now, we will define the method "about" as a "staticmethod" to show the disadvantage of this approach. As we have learned previously in our tutorial, a staticmethod does not have a first parameter with a reference to an object. So about will have no parameters at all. Due to this, we are now capable of calling "about" without the necessity of passing an instance as a parameter, i.e. "Pet.about()", "Dog.about()" and "Cat.about()". Yet, a problem lurks in the definition of "about". The only way to access the class info `_class_info` is putting a class name in front. We arbitrarily put in "Pet". We could have put there "Cat" or "Dog" as well. No matter what we do, the solution will not be what we want:

```
class Pet:
    _class_info = "pet animals"

    @staticmethod
    def about():
        print("This class is about " + Pet._class_info + "!")


class Dog(Pet):
    _class_info = "man's best friends"

class Cat(Pet):
    _class_info = "all kinds of cats"

Pet.about()
Dog.about()
Cat.about()
```

```
This class is about pet animals!
This class is about pet animals!
This class is about pet animals!
```

In other words, we have no way of differenciating between the class Pet and its subclasses Dog and Cat. The problem is that the method "about" does not know that it has been called via the Pet, the Dog or the Cat class.

A classmethod is the solution to all our problems. We will decorate "about" with a classmethod decorator instead of a staticmethod decorator:

```
class Pet:
    _class_info = "pet animals"

    @classmethod
    def about(cls):
        print("This class is about " + cls._class_info + "!")


class Dog(Pet):
    _class_info = "man's best friends"

class Cat(Pet):
    _class_info = "all kinds of cats"

Pet.about()
Dog.about()
Cat.about()
```

```
This class is about pet animals!
This class is about man's best friends!
This class is about all kinds of cats!
```