CHAPTER 12 INHERITANCE AND POLYMORPHISM

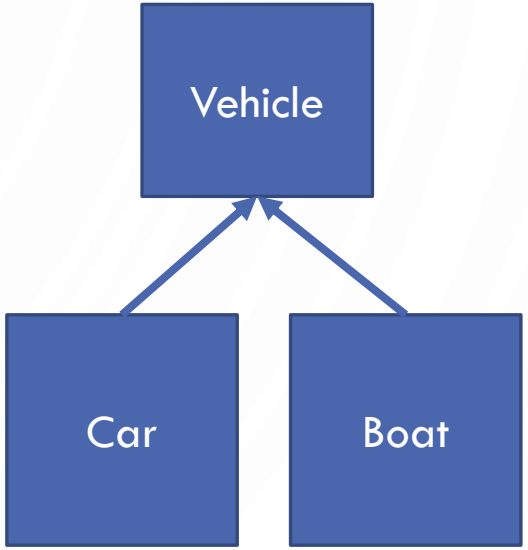
ACKNOWLEDGEMENT: THESE SLIDES ARE ADAPTED FROM SLIDES PROVIDED WITH INTRODUCTION TO PROGRAMMING USING PYTHON, LIANG (PEARSON 2013)

MOTIVATIONS

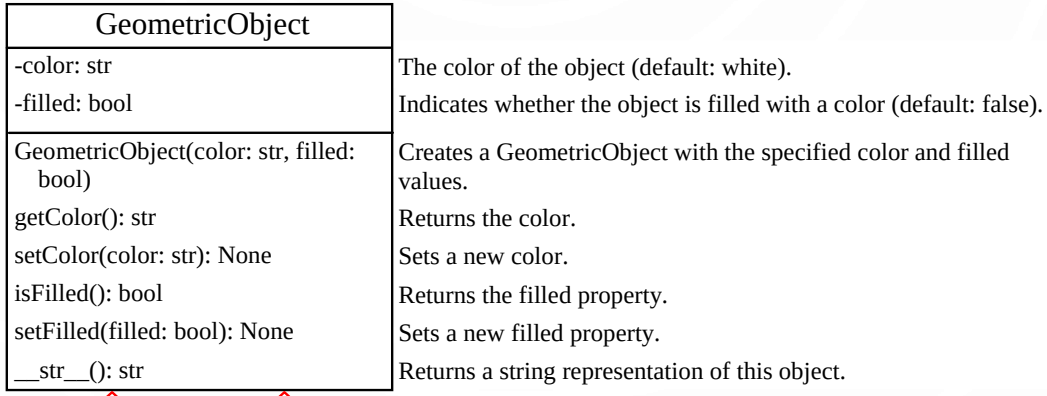
- Suppose you will define classes to model circles, rectangles, and triangles. These classes have many common features. What is the best way to design these classes so to avoid redundancy?
  - Inheritance!

SUPERCLASSES AND SUBCLASSES

- **Inheritance** defines an *IS-A relationship* between two classes to denote a type/sub-type relationship
  - Examples: a car IS-A vehicle and a boat IS-A vehicle – they both have engines but a car more specifically has wheels and a boat has a rudder
- A **superclass** defines an abstract type, whereas **subclasses** define more specific types
  - Superclass stores elements and provides methods that are common to all sub-types, whereas a subclass stores *additional* data and provides *additional* methods that more specialize the object type
  - In the example: vehicle is a superclass and car/boat are subclasses
- All methods/data of the superclass are available to subclass objects



EXAMPLE
GEOMETRIC OBJECTS



INHERITANCE IN PYTHON

- When we say a class **extends** another class, this defines a type/sub-type relationship. The syntax is as follows:

```
class SubclassName(SuperclassName):
```

- Example:

```
class Circle(GeometricObject):
```

OVERRIDING METHODS

- A subclass inherits methods from a superclass.
- However, sometimes it is necessary for the subclass to modify the implementation of a method defined in the superclass. This is referred to as **method overriding**.
- Syntactically, you just define the method in the subclass. For example:

```
class Circle(GeometricObject):
  # Other methods are omitted
  # Override the __str__ method defined in GeometricObject
  def __str__(self):
    return super().__str__() + " radius: " + str(radius)
```

THE OBJECT CLASS

- Every class in Python is descended from the **object** class. If no inheritance is specified when a class is defined, the superclass of the class is object by default.
- There are more than a dozen methods defined in the object class. We have seen quite a few of them already, e.g., `__init__()`, `__str__()`, and `__eq__(other)`

```
class ClassName:
    ...
```

Equivalent

```
class ClassName(object):
    ...
```

__NEW__ AND __INIT__ METHODS

- All methods defined in the object class are special methods with two leading underscores and two trailing underscores.
- The `__new__()` method is automatically invoked when an object is constructed. This method then invokes the `__init__()` method to initialize the object.
- Normally you should only override the `__init__()` method to initialize the data fields defined in the new class.

__STR__ AND __EQ__ METHODS

- The `__str__()` method returns a string representation for the object. By default, it returns a string consisting of a class name of which the object is an instance and the object's memory address in hexadecimal.
- The `__eq__(other)` method returns `True` if two objects are the same. By default, `x.__eq__(y)` (i.e., `x == y`) returns `False` and `x.__eq__(x)` is `True`. You can override this method to return True if two objects have the same contents.

POLYMORPHISM

- The inheritance relationship enables a subclass to inherit features from its superclass with additional new features.
  - A subclass is a specialization of its superclass; every instance of a subclass is also an instance of its superclass, but not vice versa. For example, every circle is a geometric object, but not every geometric object is a circle.
- Therefore, you can always pass an instance of a subclass to a parameter of its superclass type.
- This is the main way **polymorphism** is exhibited in python in which a subclass object *"looks" like* its superclass (e.g., by a parameter pass) but *acts like* its specialization.
- The magic of polymorphism is supported by **dynamic binding** in which when a method is invoked from an instance its most overridden form (closest to the actual type) is used instead of the most generic version

ISINSTANCE FUNCTION

- The **isinstance** provides a handy way to determine is an object instance is an instance of a particular class (e.g., a subclass of a hierarchy).
- Syntax:

```
isinstance(object, className)
```

- Example:

```
o = Circle(5)
isinstance(o, Circle)           # True
isinstance(o, Rectangle)        # False
isinstance(o, GeometricObject)  # True
```