

Chapter 12 Inheritance and Class Design

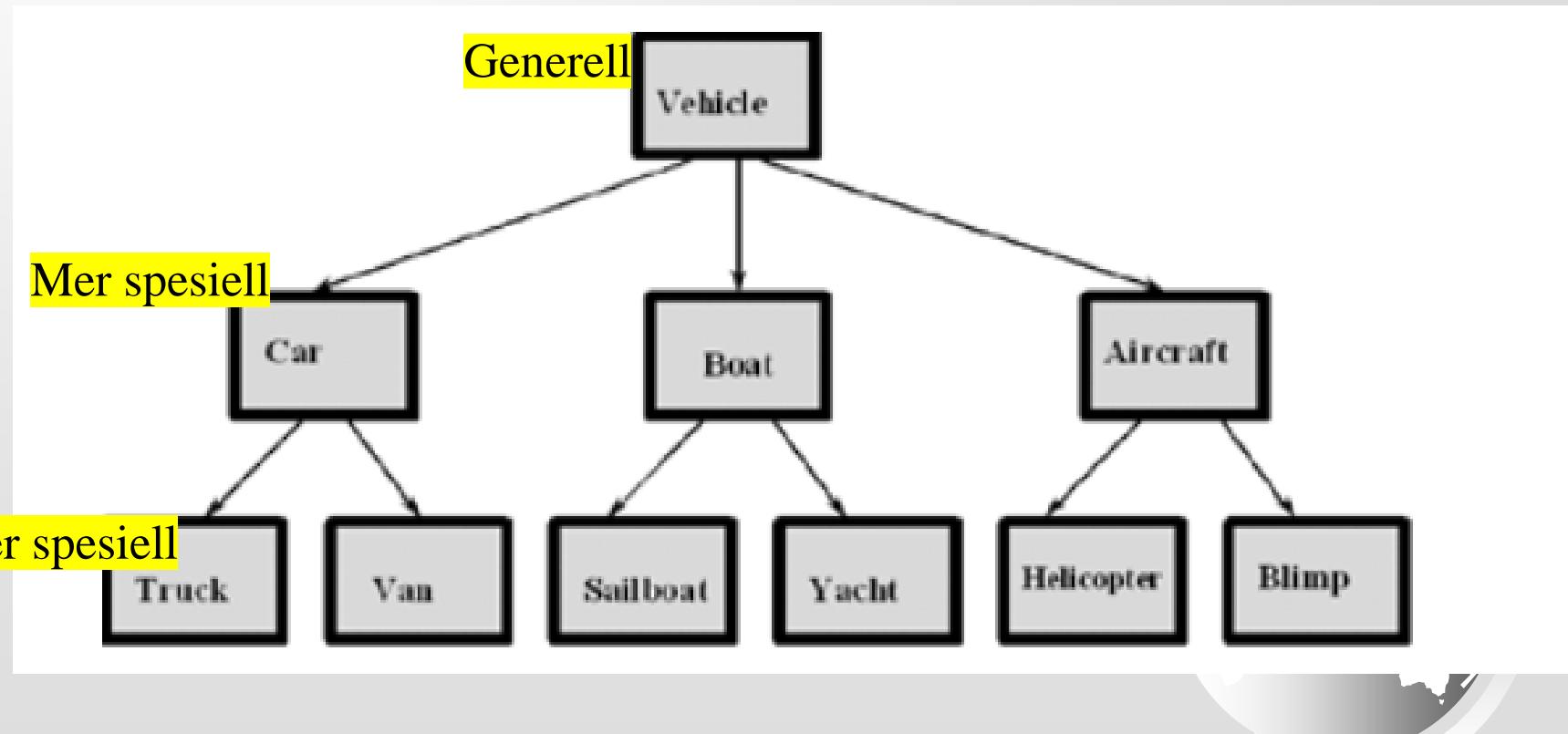


Arv / Inheritance

- Arv betyr at en klasse baseres på en annen klasse
 - Klassen som arves kalles base class / parent class
 - Klassen som arver kalles subclass / barneklasses
 - Subklassen har «mer» enn foreldreklassen
 - subklassen har både egne data og metoder, samt at den arver alt fra foreldreklassen



Subklassene blir mer og mer spesialisert jo lengre ned en går i hierarkiet



Superclasses and Subclasses

GeometricObject	
-color: str	The color of the object (default: white).
-filled: bool	Indicates whether the object is filled with a color (default: false).
GeometricObject(color: str, filled: bool)	Creates a GeometricObject with the specified color and filled values.
getColor(): str	Returns the color.
setColor(color: str): None	Sets a new color.
isFilled(): bool	Returns the filled property.
setFilled(filled: bool): None	Sets a new filled property.
__str__(): str	Returns a string representation of this object.

Circle
-radius: float
Circle(radius: float, color: str, filled: bool)
getRadius(): float
setRadius(radius: double): None
getArea(): float
getPerimeter(): float
getDiameter(): float
printCircle(): None

Rectangle
-width: double
-height: double
Rectangle(width: float, height: float, color: string, filled: bool)
getWidth(): float
setWidth(width: float): None
getHeight(): float
setHeight(height: float): None
getArea(): float
getPerimeter(): float

GeometricObject

Circle

Rectangle

TestCircleRectangle

Overriding Methods

A subclass inherits methods from a superclass. 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*.

```
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.

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

Equivalent

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

There are more than a dozen methods defined in the object class. We discuss four methods __new__(), __init__(), __str__(), and __eq__(other) here.

```
Python 3.10 (tags/v3.10.8:aaaf517, Oct 11 2022, 16:50:30) [MSC v.1933 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> o = object()
>>> dir(o)
['__class__', '__delattr__', '__dir__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__', '__lt__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclasshook__']
>>>
>>>
>>>
```



The `__new__`, `__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.



`__new__()`

- The `__new__()` method in Python is a special method that is responsible for creating a new instance of a class.
- It is called before the `__init__()` method and is used to control the creation of a new instance.
- The `__new__()` method is typically used in scenarios where you need to customize the instance creation process, such as implementing singleton patterns, immutable objects, or metaclasses.



__new__() for å implementere *singleton pattern*

```
class Singleton:  
    _instance = None  
  
    def __new__(cls, *args, **kwargs): # cls er referansen til klassen  
        if cls._instance is None:  
            cls._instance = super().__new__(cls)  
        return cls._instance  
  
    def __init__(self, value):  
        self.value = value  
  
# Test the Singleton class  
s1 = Singleton(10)  
s2 = Singleton(20)  
  
print(s1.value) # Output: 10  
print(s2.value) # Output: 10  
print(s1 is s2) # Output: True (both references point to the same instance)
```



*args

```
def example_function(*args):  
    for arg in args:  
        print(arg)
```

```
example_function(1, 2, 3) # Output: 1 2 3
```



**kwargs

```
def example_function(**kwargs):
    for key, value in kwargs.items():
        print(f'{key} = {value}')

example_function(a=1, b=2, c=3)
    # Output: a = 1, b = 2, c = 3
```



cls

- **cls**
 - brukes for å referere til selve klassen innenfor *klassemetoder*.
Det er tilsvarende self, som refererer til *instansen av klassen*, men `cls` brukes i klassemetoder som er bundet til *klassen* i stedet for instansen.
- **Klassemetoder**
 - Klassemetoder er metoder som er bundet til klassen og ikke til instansen av klassen. De kan kalles på selve klassen, i stedet for på en instans av klassen. For å definere en klassemetode, bruker du `@classmethod` - dekoratøren, og den første parameteren til metoden skal være `cls`.



```
class MyClass:  
    class_variable = 0  
  
    def __init__(self, value: int) -> None:  
        self.instance_variable = value  
  
    @classmethod  
    def increment_class_variable(cls) -> None:  
        cls.class_variable += 1  
  
    @classmethod  
    def get_class_variable(cls) -> int:  
        return cls.class_variable  
  
# Test klassemetodene  
MyClass.increment_class_variable()  
print(MyClass.get_class_variable()) # Output: 1  
  
instance = MyClass(10)  
instance.increment_class_variable()  
print(instance.get_class_variable()) # Output: 2
```



The `__str__` Method

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.

```
def __str__(self):
    return "color: " + self.__color + \
           " and filled: " + str(self.__filled)
```



The `__eq__` Method

The `__eq__` method returns `True` if two objects are the same.

By default, `x.__eq__(y)` (i.e., `x == y`) returns `False`, but `x.__eq__(x)` is `True`.

You can override this method to return `True` if two objects have the same contents.



Polymorphism

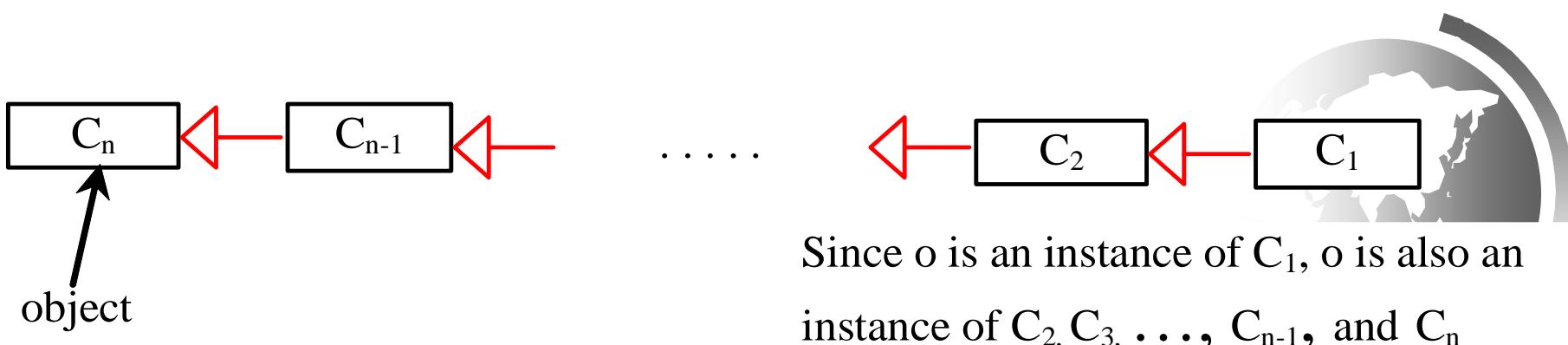
The three pillars of object-oriented programming are *encapsulation*, *inheritance*, and *polymorphism*.

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.



Dynamic Binding

Dynamic binding works as follows: Suppose an object o is an instance of classes C_1, C_2, \dots, C_{n-1} , and C_n , where C_1 is a subclass of C_2 , C_2 is a subclass of C_3, \dots , and C_{n-1} is a subclass of C_n . That is, C_n is the most general class, and C_1 is the most specific class. In Python, C_n is the object class. If o invokes a method p , the interpreter searches the implementation for the method p in C_1, C_2, \dots, C_{n-1} and C_n , in this order, until it is found. Once an implementation is found, the search stops and the first-found implementation is invoked.



The isinstance Function

The `isinstance` function can be used to determine if an object is an instance of a class.

IsinstanceDemo



Relasjoner mellom klasser

- Assosiasjon
 - Den mest generelle relasjon
 - Et objekt har en referanse til et annet objekt
- Aggregat
 - Type «har»: En bedrift har filialer (filialer kan opprettes og legges ned)
 - Uavhengig levetid
- Composition
 - Sterkere enn aggregat, er «uløselig knyttet» til / «kan ikke eksistere uten»
 - De aggregerte og det aggregerende objekt lever like lenge
 - Et hus har rom
- Vi snakker uansett om at et objekt har en *referanse* til ett eller flere objekter av en annen type (hvis flere, gjerne en liste).



Association

Association represents a general binary relationship that describes an activity between two classes.



```
class Student:  
    def addCourse(self,  
                 course):  
        # add course to a list
```

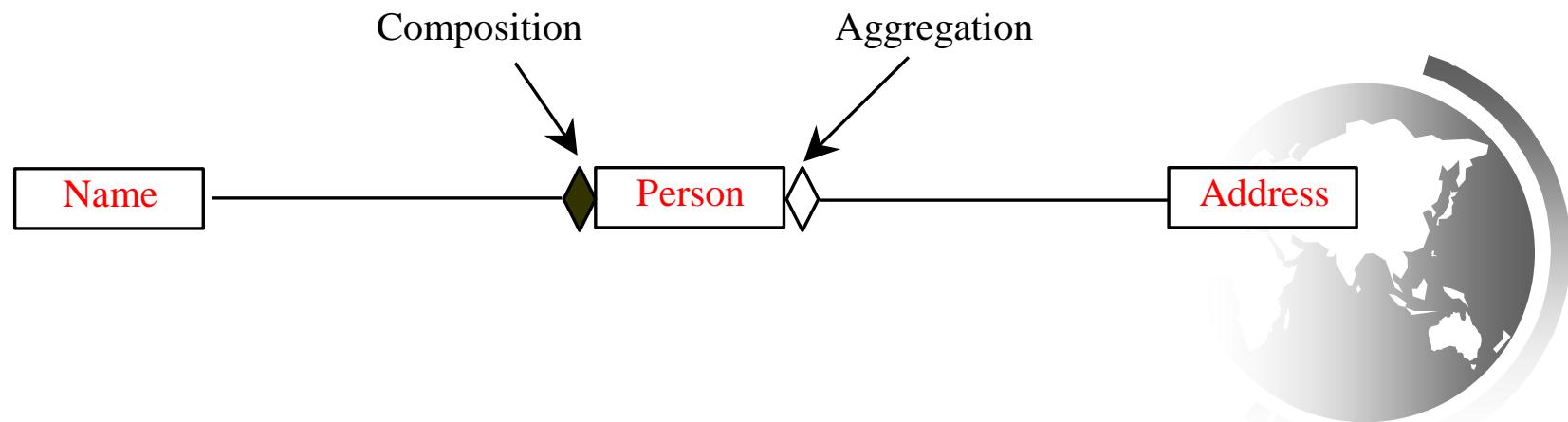
```
class Course:  
  
    def addStudent(self,  
                  student):  
        # add student to a list  
  
    def setFaculty(self, faculty):  
        # Code omitted
```

```
class Faculty:  
    def addCourse(self,  
                 course):  
        # add course to a list
```

An association is usually represented as a data field in the class.

Aggregation and Composition

Aggregation is a special form of association, which represents an ownership relationship between two classes. Aggregation models the has-a relationship. If an object is exclusively owned by an aggregated object, the relationship between the object and its aggregated object is referred to as *composition*.



Stack Animation

<https://liveexample.pearsoncmg.com/dsanimation/StackeBook.html>

The screenshot shows a web browser window titled "Stack Animation by Y. Daniel Liang". The URL in the address bar is "www.cs.armstrong.edu/liang/animation/web/Stack.html". The main content area displays the title "Stack Animation by Y. Daniel Liang". Below the title, there is a text instruction: "Enter a value and click the Push button to push the value into the stack. Click the Pop button to remove the top element from the stack." A red-bordered box contains a diagram of a stack. The stack has a pointer labeled "Top" pointing to the top element, which is the number 5. Below the stack, there are four more elements: 3, 3, and 4, each in its own box. Below the stack diagram is a form with the text "Enter a value:" followed by a text input field containing the number 5, a blue "Push" button, and a grey "Pop" button. To the right of the browser window, there is a partial view of a globe.

Stack Animation by Y. Daniel Liang

Enter a value and click the Push button to push the value into the stack. Click the Pop button to remove the top element from the stack.

Top →

5
3
3
4

Enter a value:

© Copyright 2018 by Pearson Education, Inc. All Rights Reserved.

The Stack Class (1 of 2)

You can define a class to model stacks. You can use a list to store the elements in a stack. There are two ways to design the stack and queue classes:

Using inheritance: You can define a stack class by extending `list`.

Using composition: You can create a list as a data field in the stack class.



Both designs are fine, but using composition is better because it enables you to define a completely new stack class without inheriting the unnecessary and inappropriate methods from the list class.



The Stack Class

Stack
-elements: list
+Stack()
+isEmpty(): bool
+peek(): object
+push(value: object): None
+pop(): object
+getSize(): int

A list to store elements in the stack.

Constructs an empty stack.

Returns True if the stack is empty.

Returns the element at the top of the stack without removing it from the stack.

Stores an element into the top of the stack.

Removes the element at the top of the stack and returns it.

Returns the number of elements in the stack.



Stack

TestStack