

Class hierarcies

- inheritance
- method overriding
- super
- multiple inheritance

Calling methods of a class

- If an object *obj* of class *C* has a method *method*, then usually you call *obj.method()*
- It is possible to call the method in the class directly using *C.method*, where the object is the first argument

C.method(obj)

X.py

```
class X:
    def set_x(self, x):
        self.x = x

    def get_x(self):
        return self.x

obj = X()

obj.set_x(42)

print("obj.get_x() =", obj.get_x())
print("obj.x =", obj.x)
print("X.get_x(obj) =", X.get_x(obj))
```

Python shell

```
> obj.get_x() = 42
> obj.x = 42
> X.get_x(obj) = 42
```

Classes and Objects

Observation: **students** and **employees** are **persons** with additional attributes

```
class Person
set_name(name)
get_name()
set_address(address)
get_address()
```

instance



Person object

```
name = 'Mickey Mouse'
address = 'Mouse Street 42, Duckburg'
```

```
class Student
set_name(name)
get_name()
set_address(address)
get_address()
```

instance



Student object

```
name = 'Donald Duck'
address = 'Duck Steet 13, Duckburg'
id = '1094'
grades = {'programming' : 'A' }
```

```
set_id(student_id)
get_id()
set_grade(course, grade)
get_grades()
```

Employee object

```
name = 'Goofy'
address = 'Clumsy Road 7, Duckburg'
employer = 'Yarvard University'
```

Classes and Objects

```
class Person
set_name(name)
get_name()
set_address(address)
get_address()
```

```
class Student
set_name(name)
get_name()
set_address(address)
get_address()
set_id(student_id)
get_id()
set_grade(course, grade)
get_grades()
```

person attributes

Goal – avoid redefining the 4 methods below from person class again in student class

person.py

```
class Person:
    def set_name(self, name):
        self.name = name

    def get_name(self):
        return self.name

    def set_address(self, address):
        self.address = address

    def get_address(self):
        return self.address
```

Classes inheritance

```
class Person
set_name(name)
get_name()
set_address(address)
get_address()
```

```
class Student
set_name(name)
get_name()
set_address(address)
get_address()
set_id(student_id)
get_id()
set_grade(course, grade)
get_grades()
```

person attributes

class Student **inherits** from class Person
class Person is the **base class** of Student

person.py

```
class Student(Person):
    def set_id(self, student_id):
        self.id = student_id

    def get_id(self):
        return self.id

    def set_grade(self, course, grade):
        self.grades[course] = grade

    def get_grades(self):
        return self.grades
```

Classes constructors

```
class Person
```

```
set_name(name)  
get_name()  
  
set_address(address)  
get_address()
```

```
class Student
```

```
set_name(name)           person  
get_name()               attributes  
  
set_address(address)  
get_address()  
  
set_id(student_id)  
get_id()  
  
set_grade(course, grade)  
get_grades()
```

```
person.py
```

```
class Person:  
    def __init__(self):  
        self.name = None  
        self.address = None  
  
    ...  
  
class Student(Person):  
    def __init__(self):  
        self.id = None  
        self.grades = {}  
        Person.__init__(self)  
  
    ...
```

} constructor for
Person class

} constructor for
Student class

Notes

- 1) If `Student.__init__` is not defined, then `Person.__init__` will be called
- 2) `Student.__init__` must call `Person.__init__` to initialize the name and address attributes

super()

```
class Person
```

```
    set_name(name)
    get_name()

    set_address(address)
    get_address()
```

```
class Student
```

```
    set_name(name)
    get_name()
    set_address(address)
    get_address()
    set_id(student_id)
    get_id()
    set_grade(course, grade)
    get_grades()
```

person attributes

```
person.py
```

```
class Person:
    def __init__(self):
        self.name = None
        self.address = None
    ...

class Student(Person):
    def __init__(self):
        self.id = None
        self.grades = {}
        Person.__init__(self)
        super().__init__()
    ...
```

} alternative constructor

Notes

- 1) Function `super()` searches for attributes in base class
- 2) `super` is often a keyword in other OO languages, like Java and C++
- 3) Note `super().__init__()` does not need `self` as argument

Method search order

```
class Person
set_name(name)
get_name()
set_address(address)
get_address()
```

 **parent class**

```
class Student(Person)
set_id(student_id)
get_id()
set_grade(course, grade)
get_grades()
```

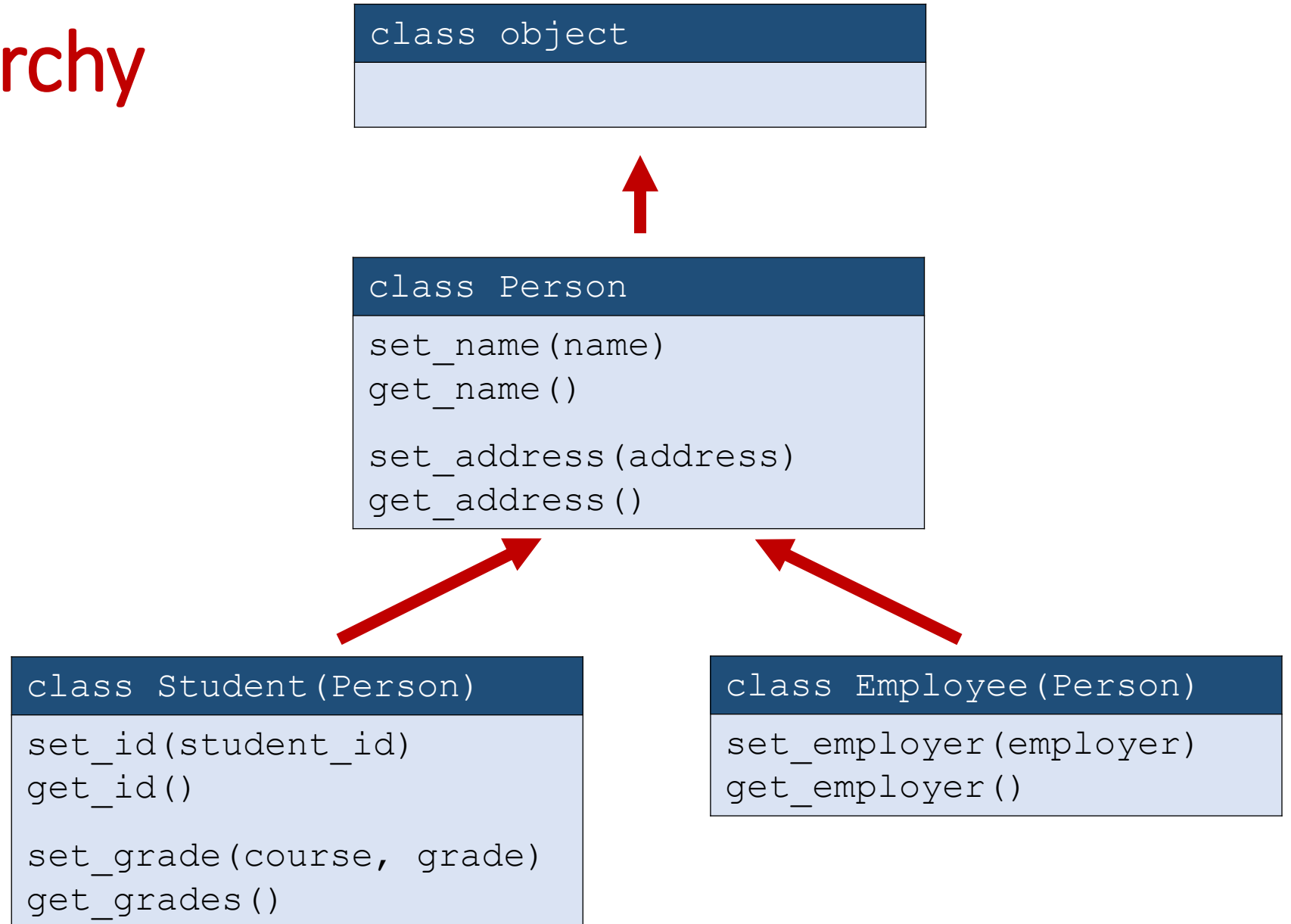
instance of



Student object

```
name = 'Donald Duck'
address = 'Duck Steet 13, Duckburg'
id = '1094'
grades = {'programming' : 'A' }
```


Class hierarchy



Method overriding

overloading.py

```
class A:
    def say(self):
        print("A says hello")

class B(A): # B is a subclass of A
    def say(self):
        print("B says hello")
        super().say()
```

Python shell

```
> B().say()
| B says hello
| A says hello
```

In Java one can use the keyword "finally" to prevent any subclass to override a method


Question – What does `b.f()` print ?

Python shell

```
> class A():
    def f(self):
        print("Af")
        self.g()
    def g(self):
        print("Ag")
> class B(A):
    def g(self):
        print("Bg")
> b = B()
> b.f()
| ?
```

a) AttributeError

b) Af Ag

 c) Af Bg

d) Don't know

Name mangling and inheritance

Python shell

```
> class A():
    def f(self):
        print("Af")
        self.__g()
    def __g(self):
        print("Ag")
> class B(A):
    def __g(self):
        print("Bg")
> b = B()
> b.f()
| Af
| Ag
```

- The call to `A.__g` in `A.f` forces a call to `__g` to stay within `A`
- Recall that due to name mangling, `__g` is accessible as `A._A__g`

Multiple inheritance

- A class can inherit attributes from multiple classes (in example two)
- When calling a method defined in several ancestor classes, Python executes only one of these (in the example `say_hello`).
- Which one is determined by the so called "C3 Method Resolution Order" (originating from the Dylan language).

multiple_inheritance.py

```
class Alice:
    def say_hello(self):
        print("Alice says hello")
    def say_good_night(self):
        print("Alice says good night")
class Bob:
    def say_hello(self):
        print("Bob says hello")
    def say_good_morning(self):
        print("Bob says good morning")
class X(Alice, Bob): # Multiple inheritance
    def say(self):
        self.say_good_morning()
        self.say_hello()
        self.say_good_night()
```

Python shell

```
> X().say()
| Bob says good morning
| Alice says hello
| Alice says good night
```

Method resolution order

- Use `help(class)` to determine the resolution order for the class

Python shell

```
> help(X)
| Help on class X in module __main__:
| class X(Alice, Bob)
|     | Method resolution order:
|     |     X
|     |     Alice
|     |     Bob
|     |     builtins.object
|     | Methods defined here:
|     | say(self)
|     | -----
|     | Methods inherited from Alice:
|     | say_good_night(self)
|     | say_hello(self)
|     | -----
|     | ...
|     | -----
|     | Methods inherited from Bob:
|     | say_good_morning(self)
```

Question – Who says hello ? Bob says good morning

`inheritance.py`

```
class Alice:
    def say_hello(self):
        print("Alice says hello")
class Bob:
    def say_hello(self):
        print("Bob says hello")
    def say_good_morning(self):
        self.say_hello()
        print("Bob says good morning")
class X(Alice, Bob): # Multiple inheritance
    pass

X().say_good_morning()
```



- a) Alice
- b) Bob
- c) Dont' know

Comparing objects and classes

- `id(obj)` returns a unique identifier for an object (in CPython the memory address)
- `obj1 is obj2` tests if `id(obj1) == id(obj2)`
- `type(obj)` and `obj.__class__` return the class of an object
- `isinstance(object, class)` checks if an object is of a particular class, or a derived subclass
- `issubclass(class1, class2)` checks if class1 is a subclass of class2

Comparison of OO in Python, Java and C++

- private, public, – in Python everything in an object is public
- class inheritance
 - Python and C++ support multiple inheritance
 - Java only allows single inheritance, but Java "interfaces" allow for something like multiple inheritance
- Python and C++ allows overloading standard operators (+, *, ...).
In Java it is not possible.
- Overloading methods
 - Python extremely dynamic (hard to say anything about the behaviour of a program in general)
 - Java and C++'s type systems allow several methods with same name in a class, where they are distinguished by the type of the arguments