Object-oriented design

Computing & Information Sciences

W. H. Bell

Inheritance

```python
class MapPosition:
    def __init__(self):
        self.latitude = 0.
        self.longitude = 0.

class InclinedPosition(MapPosition):
    def __init__(self):
        self.elevation = 0.

m = MapPosition()   # Create a position
m.latitude = 13.0
m.longitude = -10.0
p = InclinedPosition() # Create an inclined position
p.latitude = 55.860916
p.longitude = -4.251433
p.elevation = 16
```

Public, Protected and Private

- Functions, data members - public, protected or private.
  - Public – Accessible from outside the class.
  - Protected - Accessible from a derived class, but not from outside the derived or base class.
  - Private – Not accessible from outside the class.

Public, Protected and Private

"_" (single underscore) => protected
"__" (double underscore) => private

```
class MyClass:
    def __init__(self):
        self.name = "MyClass"
        self._protected_name = "Only derived know"
        self.__private_name = "Only this class knows"

    def public_function(self):
        return "This a public function"

    def _protected_function(self):
        return "This is a protected function"

    def __private_function(self):
        return "This is a private function"
```

Accessor and mutator functions

```
class MyClass:
    def __init__(self):
        self.__name = "MyClass"

    def set_name(self, name):
        self.__name = name

    def get_name(self):
        return self.__name

m = MyClass()
m.set_name("New name")
print(m.get_name())
```

Accessor and mutator functions

- Used to access private or protected data members.
  - Accessor – get.
  - Mutator – set.
- Python programmers tend to avoid them.
  - Use public data members instead and directly access them.
  - There is a processing overhead.
  - Processing overhead is reduced slightly in compiled languages.

Operator overloading

- Define functions to allow object operations.
  - Conversion to strings.
  - Comparisons.
  - Mathematical operations.
- Implement within class to improve code structure.

Operator functions

- Characterised by the pattern `__`*`name`*`__`

`__repr__`	String representation.
`__str__`	Readable string representation.
`__eq__`	Comparison, equals.
`__ne__`	Comparison, not equal.
`__add__`	Add.
`__sub__`	Subtract.
`__mul__`	Multiply.

https://docs.python.org/3/library/operator.html

String representation

```python
class MyClass:
    def __init__(self, name):
        self.name = name

    def __repr__(self):
        return f"MyClass(name=\"{self.name}\")"

obj = MyClass("Some name")
print(obj)
obj2 = eval(str(obj))
```

Output

```
MyClass(name="Some name")
```

Comparisons

```python
class DataClass:
    def __init__(self, x):
        self.x = x

    def __eq__(self, other):
        return self.x == other.x

    def __ne__(self, other):
        return not self.__eq__(other)

d = DataClass(10)
p = DataClass(10)
print("d == p : " + str(d == p))
print("d != p : " + str(d != p))
```

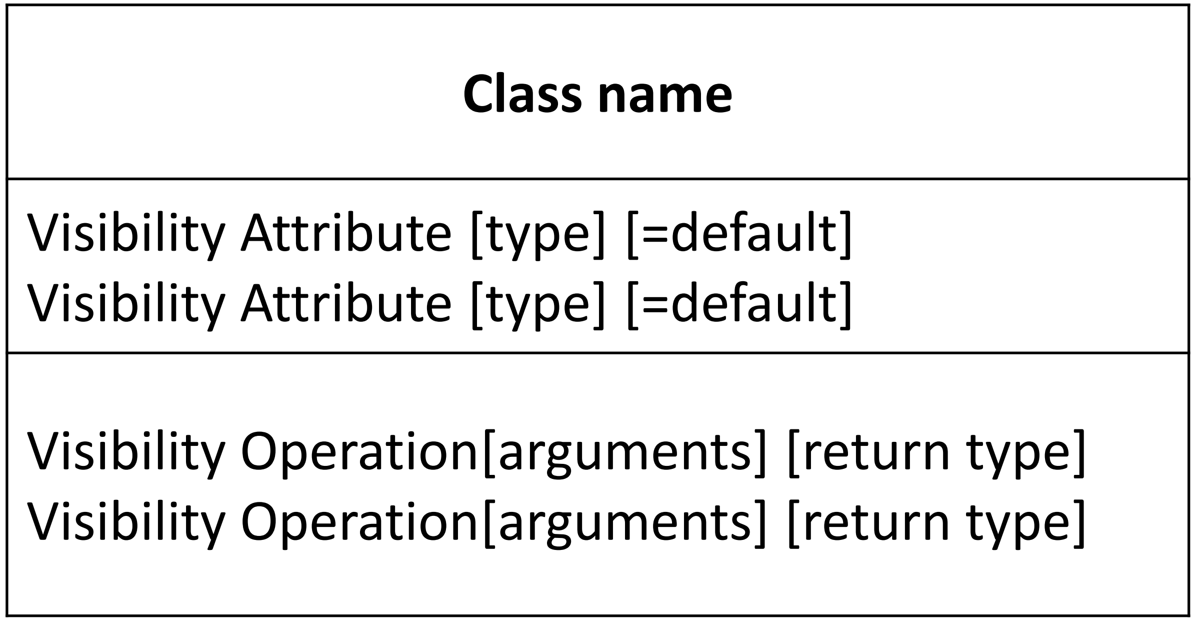
Output

```
d == p : True
d != p : False
```

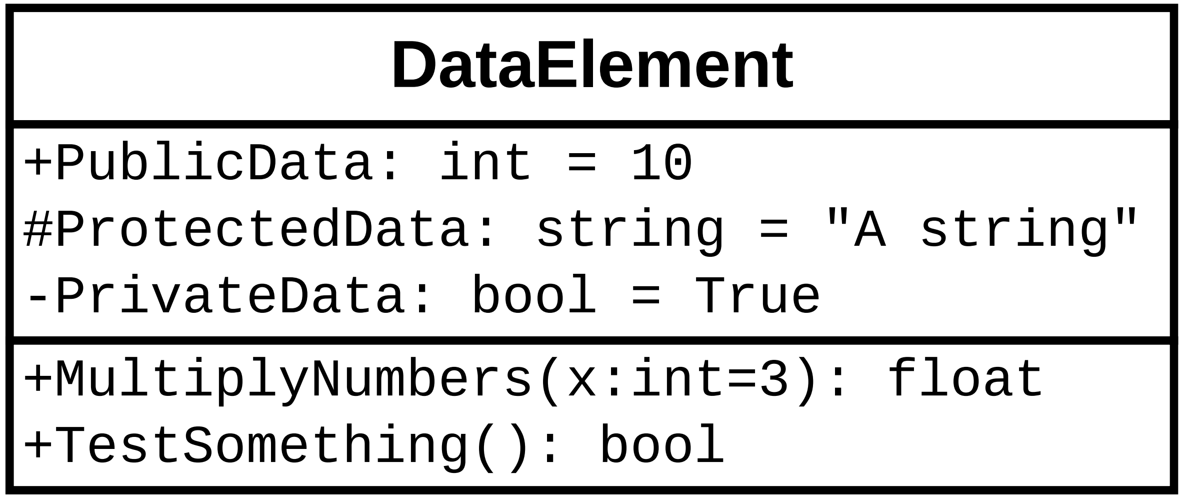
Object-oriented issues

- Limited understanding at start of development.
  - Difficult to encapsulate all data and functionality.
  - Incorrect encapsulation may result in large changes.
  - Incorrect use of inheritance may be costly to rewrite.
- State split between objects.
  - Obscure data flow or copy data around needlessly.

UML class diagrams



```
Class name
----------------------------------------------
Visibility Attribute [type] [=default]
Visibility Attribute [type] [=default]
----------------------------------------------
Visibility Operation[arguments] [return type]
Visibility Operation[arguments] [return type]
```



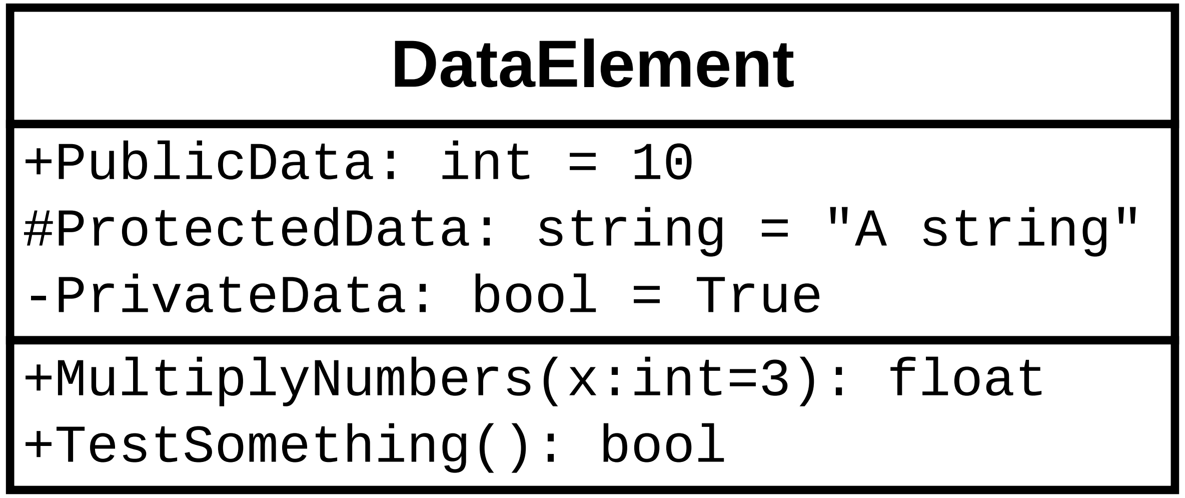
```
DataElement
----------------------------------------------
+PublicData: int = 10
#ProtectedData: string = "A string"
-PrivateData: bool = True
----------------------------------------------
+MultiplyNumbers(x:int=3): float
+TestSomething(): bool
```

Visibility:

- + => Public
- # => Protected
- - => Private

UML and Python implementation

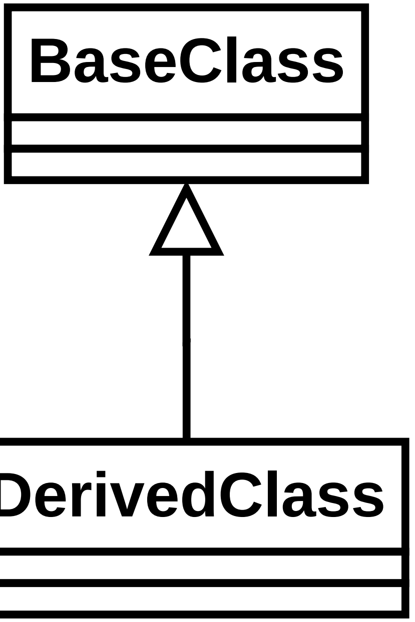
```python
class DataElement:
    def __init__(self):
        self.public_data = 10
        self._protected_data = "A string"
        self.__private_data = True

    def multiply_numbers(self, x=3):
        return 0.

    def test_something(self):
        return False
```



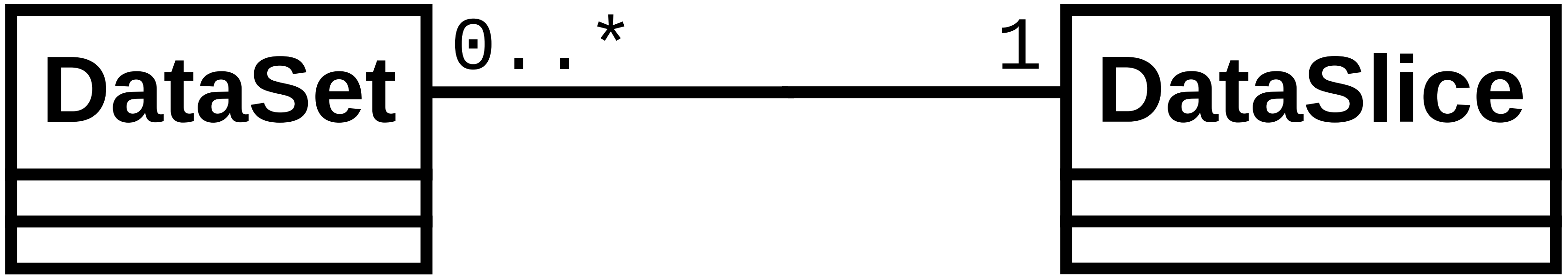
```
DataElement
+PublicData: int = 10
#ProtectedData: string = "A string"
-PrivateData: bool = True
+MultiplyNumbers(x:int=3): float
+TestSomething(): bool
```

UML: class relationships

- Classes can inherit from others.
- Define inherited attributes or operations in the base class.



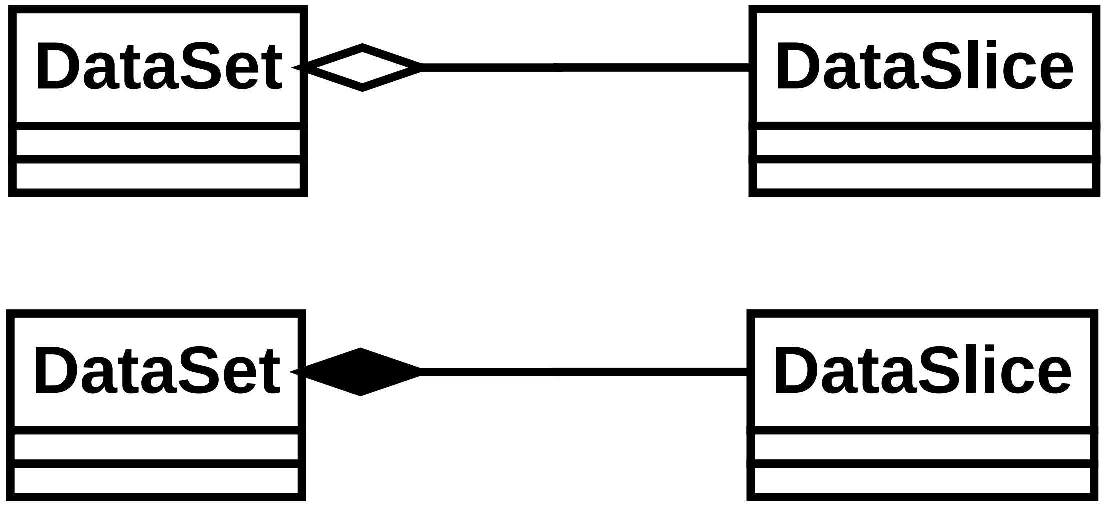
UML: class association



- Define multiplicities in association:
  - 0..* - Zero or more.
  - 1 - Exactly one.
  - 1..* - One or more.
  - 0..1 - Zero or one.

UML: class composition

- Used to express that classes are part of another class.



Aggregation: may contain 0 or more.

Composition: may contain 1 or more.

UML Tools

- Create UML diagrams with:
    - https://app.diagrams.net/ (Online).
    - Dia Diagram Editor (Windows, Linux, Mac).
    - UML Designer.
    - Microsoft Visio Pro.
- Can autogenerate UML from code.
    - Pyreverse, Doxygen.

University of Strathclyde Glasgow