Advanced Programming

Inheritance

Ramaseshan Ramachandran

INHERITANCE

Inheritance allows us to create new classes (subclasses) based on existing classes (superclasses), promoting code reuse and establishing relationships between objects.

WHAT IS INHERITANCE?

- Allows new classes to inherit properties and behaviours from existing classes
- Establishes a hierarchical relationship between classes
  - The class inheriting properties is called the subclass or derived class
- The class providing the properties is called the superclass or base class

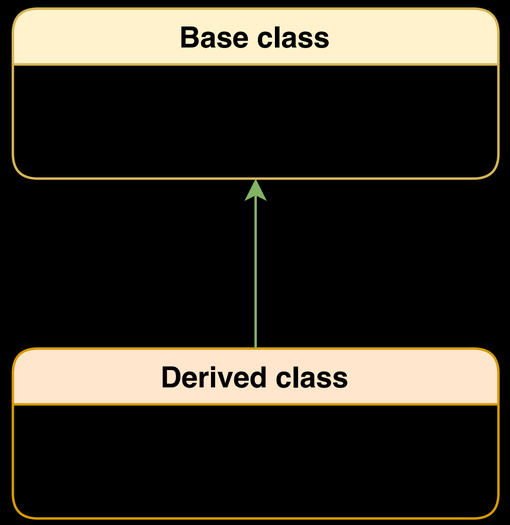
ADVANTAGES

- Allows us to reuse existing code, saving time and effort.
- Changes made to the superclass automatically propagate to subclasses, simplifying maintenance.
- Sub-class extends the functionality of the superclass by adding new methods or overriding existing ones
- Enables polymorphism, allowing objects of different subclasses to be treated uniformly.

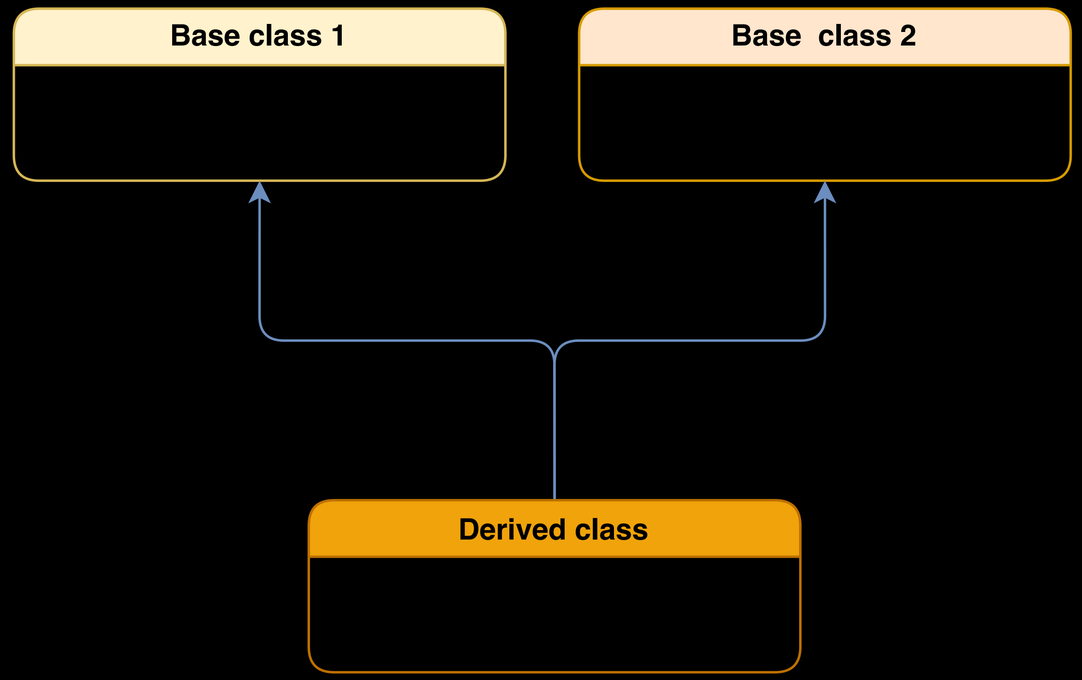
TYPES OF INHERITANCE

- Single Inheritance:
  - A subclass inherits from one superclass
- Multiple Inheritance
  - A subclass inherits from multiple superclasses
- Multilevel Inheritance
  - A subclass inherits from another subclass, which inherits from a base class, forming a chain.

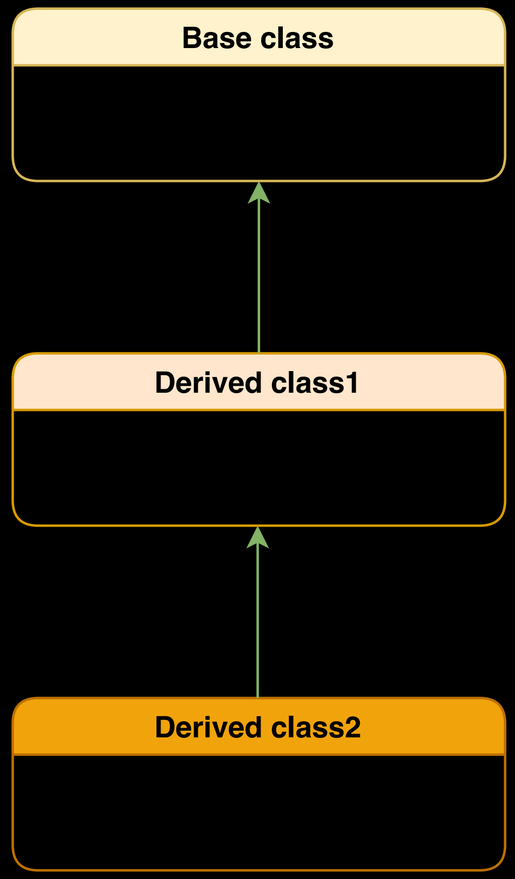
INHERITANCE DIAGRAM



Single Inheritance



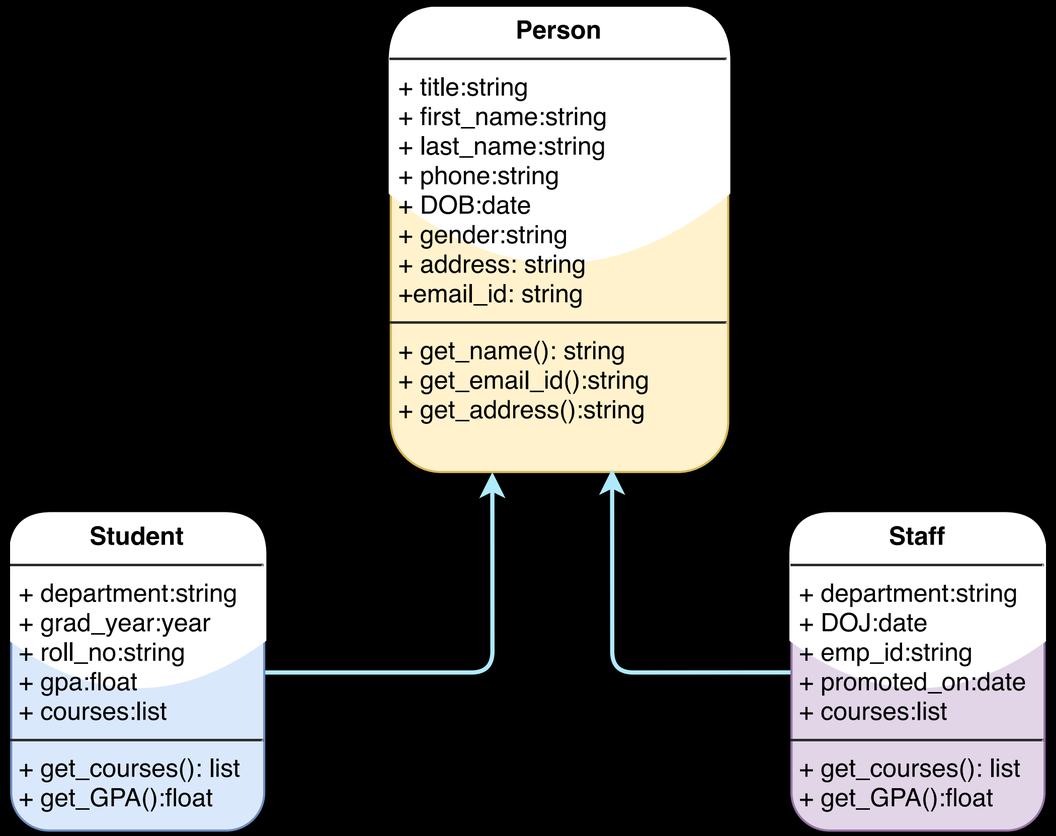
Multiple Inheritance



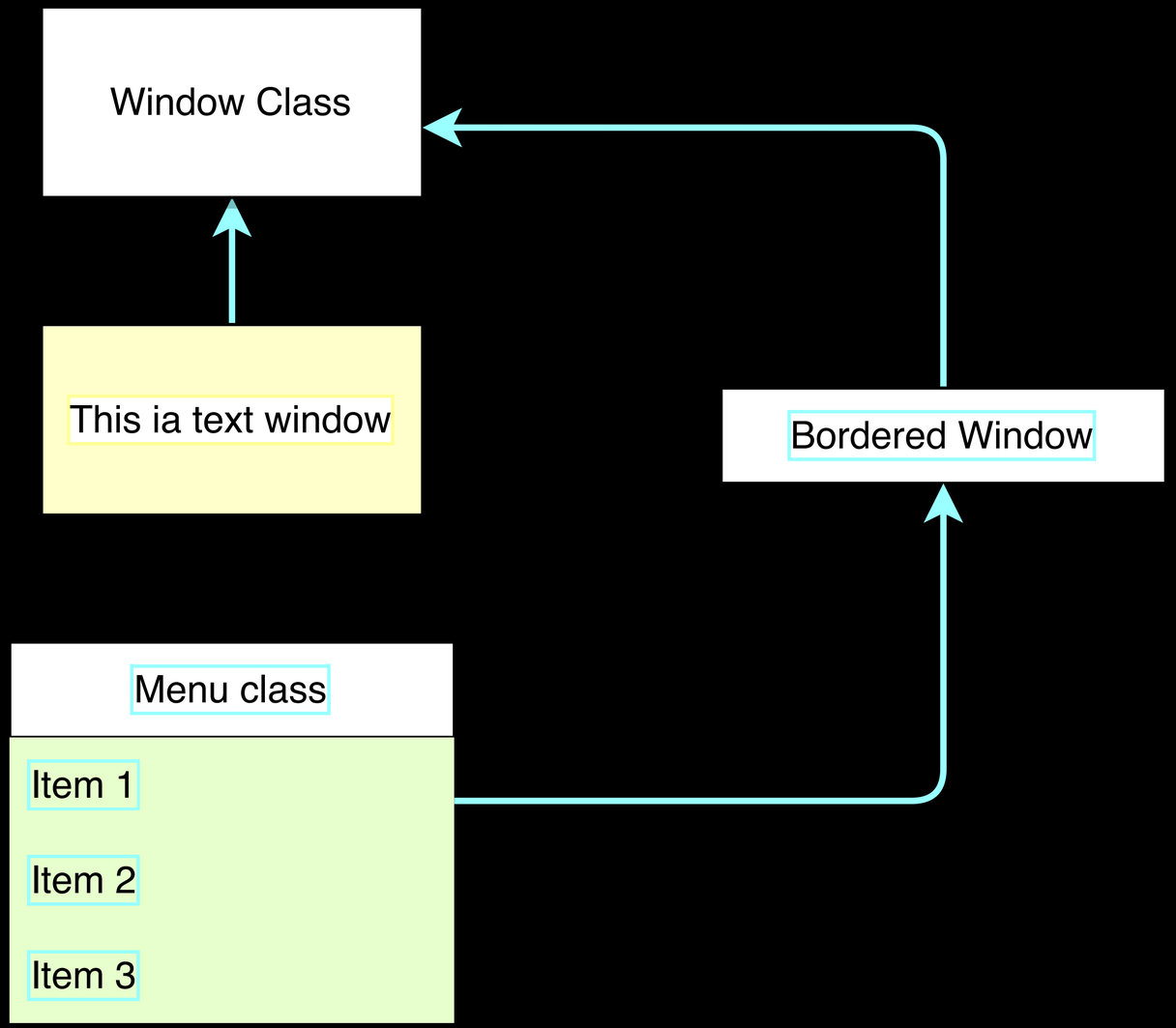
Multilevel Inheritance

Universal Modeling Language (UML) representation

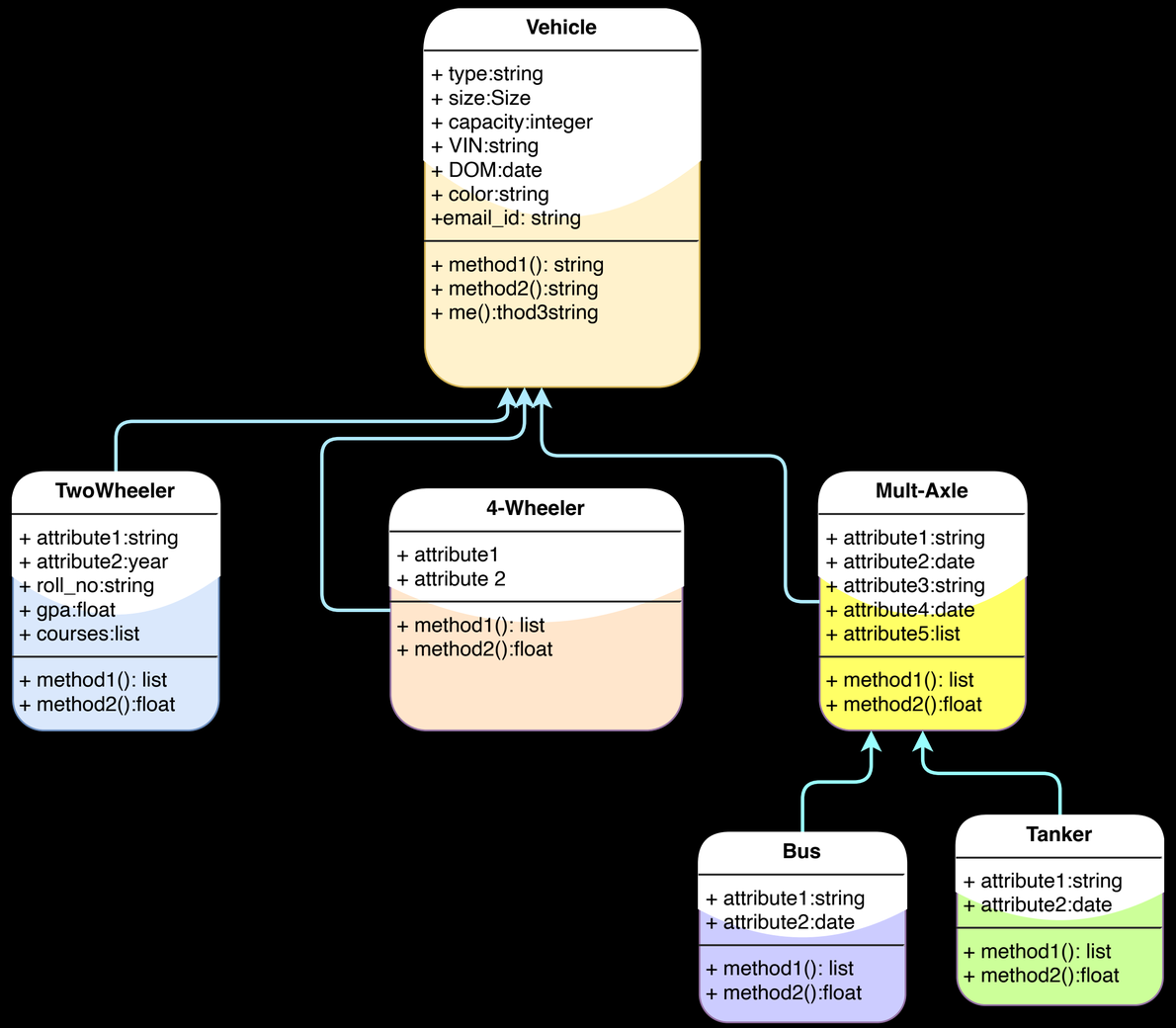
INHERITANCE - EXAMPLE



WINDOW INHERITANCE



INHERITANCE - EXAMPLE



PRIVATE MEMBERS

- No enforced access modifiers like private
- Offers conventions to establish a controlled access mechanism
  - Single Leading Underscore (_name)
  - Discourages direct access from outside the class
  - Does not strictly enforce privacy
  - Members can still be accessed from outside the class - Python does not prevent this
- Dunder (double underscore) conventions

```
class Vehicle:
    def __init__(self):
        self._VIN = "MCXX1234B"

# Outside the class
obj = Vehicle()
print(obj._VIN) #Access is allowed
```

```
class Vehicle:
    def __init__(self):
        self.__VIN = "MCXX1234B"

# Outside the class
obj = Vehicle()
print(obj.__VIN) # Not Allowed

AttributeError: 'Vehicle' object has no
attribute ‘__VIN'
```

PRIVATE MEMBERS

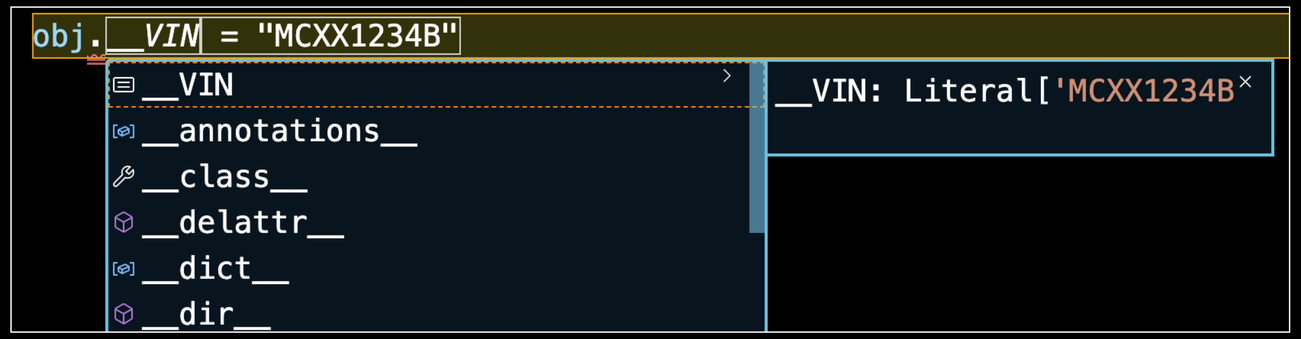
Dunder (double underscore) conventions

- A stronger convention for private members, especially when dealing with attributes that might conflict with names used by subclasses or for internal implementation details.
- Added layer of discouragement due to the naming style
- Employs name **mangling** to automatically rename these members
  - For example, _ClassName__private_var becomes _object._ClassName__private_var. While technically accessible, it's generally considered bad practice to rely on this for direct access.

```
class Vehicle:
    def __init__(self):
        self.__VIN = "MCXX1234B"

# Outside the class
obj = Vehicle()
print(obj.__VIN) # Not Allowed

AttributeError: 'Vehicle' object has no
attribute '__VIN'
```



HOW PRIVATE IS PRIVATE?

```
class Student:
  __school_name = 'CMI' # private class attribute

  def __init__(self, name, age):
    self.__name = name
    self.__age = age

  def __print(self):
    print(self.__school_name)

std = Student("Ram", 25)
print(std._Student__name)

std._Student__name = 'Raj'
print(std._Student__name)
std._Student__print()
```

PRIVATE IN C++

```cpp
#include <iostream>

class Person {
public:
 std::string name;
private:
  // Private variable
  int age;
public:
 void set_age(int new_age) {
    if (new_age >= 0) {
      age = new_age;
    } else {
      std::cout << "Error: Age cannot be negative."
                << std::endl;
    }
  }

 int get_age() {
    return age;
  }
};
```

```cpp
int main() {
  Person person;
  person.name = "Ram";
  //person.age = 25;
  person.set_age(25);
  std::cout << person.name << " is "
            << person.get_age()
            << " years old." << std::endl;

  return 0;
}
```

PRIVATE IN RUST

```
struct Person {
   pub name: String,
   age: i32,
  }

  impl Person {
   fn set_age(&mut self, new_age: i32) {
      if new_age >= 0 {
        self.age = new_age;
      } else {
        println!("Error:
             Age cannot be negative.");
      }
    }
     //(ontrolled access
    fn get_age(&self) -> i32 {
      self.age
    }
  }
```

```
fn main() {
    let mut person = Person
                    { name: String::from("Ram"),
                      age: 25 };
    println!("{} is {} years old.",
              person.name, person.get_age());

   // person.age = 30;
    person.set_age(30);
    println!("{} is {} years old.",
              person.name, person.get_age());
  }
```