

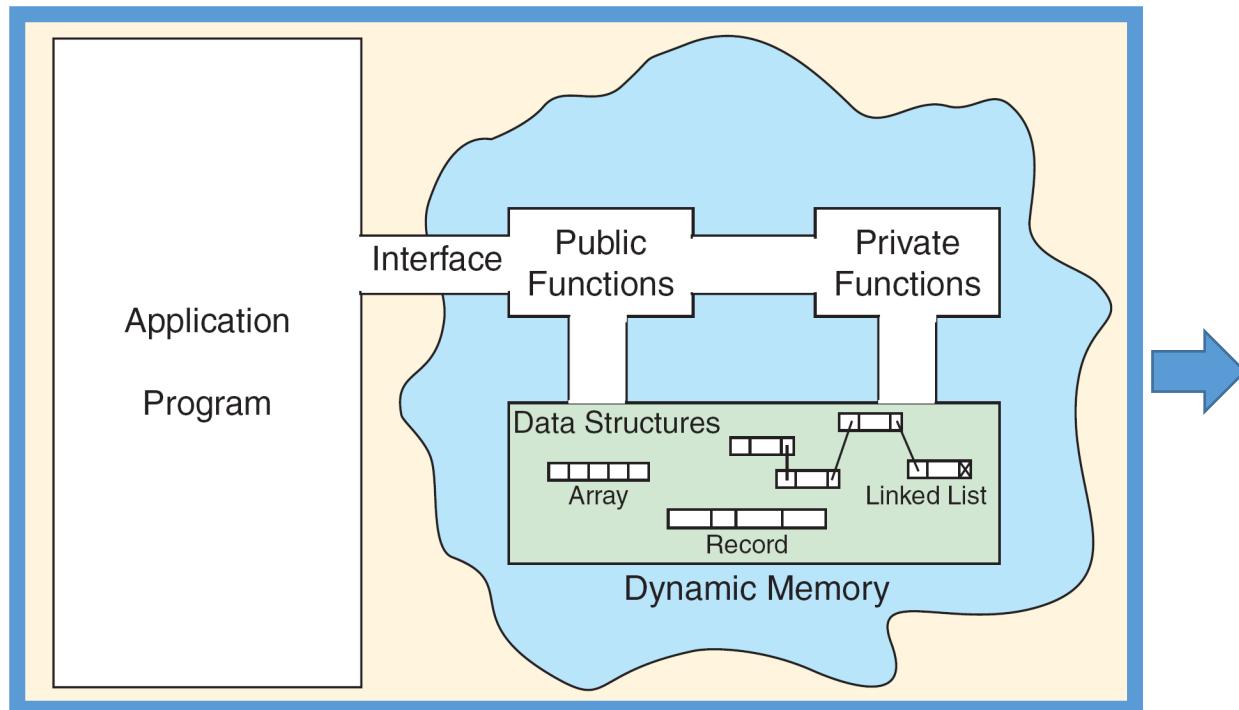
# Classes and Object Oriented Programming examples

# Outline

- Abstract Data Type: ADT
- Object Oriented Programming
- Encapsulation
  - Class, Object(instance)
- Inheritance
  - Method Overriding
  - Operator Overloading
- Polymorphism

# ADT and class

- 객체지향언어는 ADT 개념을 class 명령어를 사용하여 구현
- Python에서 class는 데이터와 함수를 묶어서 선언할 수 있도록 만들어진 사용자 정의 자료형(User defined data type)이라 할 수 있음



```
#Python - Class

class Car :
    color = ""
    speed = 0

    def upSpeed(self, value) :
        self.speed += value

    def downSpeed(self, value) :
        self.speed -= value

mycar1=Car() #객체 생성
```

# Public and Private

- Python은 객체지향 캡슐화 개념을 도입하여 class 명령어를 사용하나 기본적으로 데이터는 공개(public)함
- Python에서 private으로 데이터와 메소드를 선언하는 방법은 클래스 안에 데이터이름과 메소드 이름 앞에 (\_)를 붙여서 선언 (ex) \_\_data, \_\_method()

# Public and Private example

```
class Student:  
    def __init__(self, name, age):  
        self.__name = name  
        self.__age = age  
  
    def __show_name(self):  
        print(self.__name)  
  
    def __show_age(self):  
        print(self.__age)  
  
student = Student('Alice', 20)  
  
print(student.__name) # error  
print(student.__age) # error  
student.__show_name() # error  
student.__show_age() # error
```

AttributeError: 'Student' object has no attribute '\_\_name'

# Public and Private example

```
class Student:  
    def __init__(self, name, age):  
        self.__name = name  
        self.__age = age  
  
    def show_name(self):  
        print(self.__name)  
  
    def show_age(self):  
        print(self.__age)  
  
student = Student('Alice', 20)  
  
#print(student.__name)  
#print(student.__age)  
student.__show_name() #error  
student.__show_age() #error
```

AttributeError: 'Student' object has no attribute '\_\_show\_name'

# Public and Private example

```
class Student:  
    def __init__(self, name, age):  
        self.__name = name  
        self.__age = age  
  
    def show_name(self):  
        print(self.__name)  
  
    def show_age(self):  
        print(self.__age)  
  
student = Student('Alice', 20)  
student.show_name()  
student.show_age()
```

Alice  
20

# Class example(Stack)

- 스택(Stack)은 데이터를 저장하는 일종의 자료 구조로, 가장 마지막에 추가된 데이터가 가장 먼저 꺼내지는 후입선출(LIFO: Last-In-First-Out) 구조
- 스택은 push(데이터 추가)과 pop(데이터 삭제) 두 가지 기본 동작을 가짐

- Stack 클래스를 정의하고, push, pop, peek, is\_empty 메소드를 구현
- push : 스택에 데이터를 추가, 파라미터로 전달된 item을 items 리스트에 추가
- pop : 스택에서 데이터를 꺼내는 메소드, 스택이 비어있을 경우 None 값을 반환, 스택이 비어있지 않을 경우 가장 마지막에 추가된 데이터를 꺼내서 반환.
- peek : 스택의 맨 위에 있는 데이터를 반환하는 메소드, 스택이 비어있을 경우 None 값을 반환
- is\_empty : 스택이 비어있는지 확인하는 메소드, 스택이 비어있으면 True 값을, 그렇지 않으면 False 값을 반환

# Class example(Stack)

```
class Stack:  
    def __init__(self):  
        self.items = []  
  
    def push(self, item):  
        self.items.append(item)  
  
    def pop(self):  
        if self.is_empty():  
            return None  
        return self.items.pop()  
  
    def peek(self):  
        if self.is_empty():  
            return None  
        return self.items[-1]  
  
    def is_empty(self):  
        return len(self.items) == 0
```

peek 메서드는 스택의 맨 위에 있는 데이터를 반환  
이를 위해 리스트의 [-1] 인덱스를 사용  
스택이 비어있을 경우에는 None 값을 반환

# Class example(Stack)

```
class Stack:  
    def __init__(self):  
        self.items = []  
  
    def push(self, item):  
        self.items.append(item)  
  
    def pop(self):  
        if self.is_empty():  
            return None  
        return self.items.pop()  
  
    def peek(self):  
        if self.is_empty():  
            return None  
        return self.items[-1]  
  
    def is_empty(self):  
        return len(self.items) == 0
```

```
stack = Stack()  
stack.push(1)  
stack.push(2)  
stack.push(3)  
  
print(stack.peek())  
print(stack.pop())  
print(stack.pop())  
print(stack.pop())  
print(stack.pop())
```

```
3  
3  
2  
1  
None
```

# Class example(Queue)

- 큐(Queue)는 데이터를 저장하는 일종의 자료 구조로, 가장 처음에 추가된 데이터가 가장 먼저 꺼내지는 선입선출(FIFO: First-In-First-Out) 구조
- 큐는 enqueue(데이터 추가)과 dequeue(데이터 삭제) 두 가지 기본 동작을 가짐

1. Queue 클래스를 정의하고, enqueue, dequeue, peek, is\_empty 메소드를 구현
2. enqueue : 큐에 데이터를 추가하는 메소드, 파라미터로 전달된 item을 items 리스트의 끝에서 앞으로 추가
3. dequeue : 큐에서 데이터를 꺼내는 메소드, 큐가 비어있을 경우 None 값을 반환하고, 큐가 비어있지 않을 경우 가장 먼저 추가된 데이터를 꺼내서 반환
4. peek : 큐의 맨 앞에 있는 데이터를 반환하는 메소드, 큐가 비어있을 경우 None 값을 반환
5. is\_empty : 큐가 비어있는지 확인하는 메소드, 큐가 비어있으면 True 값을, 그렇지 않으면 False 값을 반환

# Class example(Queue)

```
class Queue:  
    def __init__(self):  
        self.items = []  
  
    def enqueue(self, item):  
        self.items.append(item)  
  
    def dequeue(self):  
        if self.is_empty():  
            return None  
        return self.items.pop(0)  
  
    def peek(self):  
        if self.is_empty():  
            return None  
        return self.items[0]  
  
    def is_empty(self):  
        return len(self.items) == 0
```

```
queue = Queue()  
queue.enqueue(1)  
queue.enqueue(2)  
queue.enqueue(3)  
  
print(queue.peek())  
print(queue.dequeue())  
print(queue.dequeue())  
print(queue.dequeue())  
print(queue.dequeue())
```

```
1  
1  
2  
3  
None
```

# Inheritance example(method overriding )

1. Student 클래스 : 부모 클래스로 정의하고, Undergraduate 클래스와 Graduate 클래스가 Student 클래스를 상속 받아 생성
2. Student 클래스 : 이름(name)과 성적(grade)을 인스턴스 변수로 가지며, display 메소드를 구현, display 메소드는 인스턴스 변수의 값을 출력
3. Undergraduate 클래스 : Student 클래스를 상속받아 생성, 전공(major)이 추가된 클래스
  - `init` 메소드 : `super()` 함수를 이용하여 상위 클래스의 생성자를 호출하고, 전공(major)을 추가로 초기화
  - `display` 메소드 : 상위 클래스의 `display` 메소드를 호출하고, 전공(major) 정보를 출력
6. Graduate 클래스 : Student 클래스를 상속받아 생성, 지도 교수(advisor)가 추가된 클래스
  - `init` 메소드에서 `super()` 함수를 이용하여 상위 클래스의 생성자를 호출하고, 지도 교수(advisor)를 추가로 초기화
  - `display` 메소드 : 마찬가지로 상위 클래스의 `display` 메소드를 호출하고, 지도 교수(advisor) 정보를 출력

# Inheritance example(method overriding )

```
class Student:  
    def __init__(self, name, grade):  
        self.name = name  
        self.grade = grade  
    def display(self):  
        print(f"이름: {self.name}, 성적: {self.grade}")  
  
class Undergraduate(Student):  
    def __init__(self, name, grade, major):  
        super().__init__(name, grade)  
        self.major = major  
  
    def display(self):  
        super().display()  
        print(f"전공: {self.major}")  
  
class Graduate(Student):  
    def __init__(self, name, grade, advisor):  
        super().__init__(name, grade)  
        self.advisor = advisor  
  
    def display(self):  
        super().display()  
        print(f"지도 교수: {self.advisor}")
```

# Inheritance example(method overriding )

```
student = Student("김학생", 90)
student.display()

undergrad = Undergraduate("이학생", 85, "컴퓨터공학")
undergrad.display()

grad = Graduate("박학생", 95, "김교수")
grad.display()
```

이름: 김학생, 성적: 90  
이름: 이학생, 성적: 85  
전공: 컴퓨터공학  
이름: 박학생, 성적: 95  
지도 교수: 김교수

# Operator overloading example

```
class MyNumber:
    def __init__(self, value):
        self.value = value

    def __repr__():
        return str(self.value)

    def __add__(self, other):
        if isinstance(other, MyNumber):
            return MyNumber(self.value + other.value)
        else:
            return MyNumber(self.value + other)

    def __radd__(self, other):
        return MyNumber(self.value + other)

num1 = MyNumber(10)
num2 = MyNumber(20)
result1 = num1 + num2
result2 = 5 + num1
print(result1)  # 30
print(result2)  # 15
```

# Operator overloading example

- `__radd__` : + 연산자의 왼쪽에 있는 피연산자가 `MyNumber` 객체가 아닐 때도 처리, `__radd__` 메서드에서는 `self`와 `other` 두 개의 인자 받음. `self`는 현재 객체이며, `other`는 더할 대상 객체, `__radd__` 메서드에서는 `other`와 `self.value`를 더한 결과를 새로운 `MyNumber` 객체로 반환
- `num1 + num2` 연산을 수행하면, `num1.__add__(num2)`가 호출되어 `MyNumber(10 + 20)`이 출력
- `5 + num1` 연산을 수행하면, `num1.__radd__(5)`가 호출되어 `MyNumber(5 + 10)`이 출력
- `__add__` 메서드와 `__radd__` 메서드를 함께 사용하여, + 연산자의 양쪽 피연산자가 모두 `MyNumber` 객체이거나, 왼쪽 피연산자가 정수인 경우에도 처리

# Operator overloading example2

```
class MyNumber:
    def __init__(self, value):
        self.value = value

    def __repr__():
        return str(self.value) # value 값을 문자열로 변환하여 반환

    def __add__(self, other):
        if isinstance(other, MyNumber):
            return MyNumber(self.value + other.value)
        else:
            return MyNumber(self.value + other)

    def __radd__(self, other):
        return MyNumber(self.value + other)

    def __iadd__(self, other): # '+=' 연산자 재정의
        if isinstance(other, MyNumber):
            self.value += other.value
        else:
            self.value += other
        return self
```

# Operator overloading example2

```
num1 = MyNumber(10)
num2 = MyNumber(20)
num1 += num2
print(num1) # 30
print(num1 + 100) # 130
print(num2 + 100) # 120
print(100 + num1) # 130
print(100 + num2 + 100) # 220
num1 += 5
print(num1) # 35
num2 += 10
print(num2) # 30
```

```
30
130
120
130
220
35
30
```

# Polymorphism example

```
class Shape:
    def area(self):
        pass
    def draw(self):
        pass

class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height
    def area(self):
        return self.width * self.height
    def draw(self):
        print(f"Drawing a {self.width}x{self.height} rectangle.")

class Triangle(Shape):
    def __init__(self, base, height):
        self.base = base
        self.height = height
    def area(self):
        return 0.5 * self.base * self.height
    def draw(self):
        print(f"Drawing a triangle with base {self.base} and height {self.height}.")
```

# Polymorphism example

```
shapes = [Rectangle(2, 3), Triangle(4, 5)]  
  
for shape in shapes:  
    print(shape.area())  
    shape.draw()
```

```
6  
Drawing a 2x3 rectangle.  
10.0  
Drawing a triangle with base 4 and height 5.
```