CoGrammar

# Polymorphism and The SOLID Principles

SKILLS FOR LIFE
SKILLS BOOTCAMPS

Department for Education

# Software Engineering Lecture Housekeeping

- The use of disrespectful language is prohibited in the questions, this is a supportive, learning environment for all - please engage accordingly. **(FBV: Mutual Respect.)**
- No question is daft or silly - **ask them!**
- There are **Q&A sessions** midway and at the end of the session, should you wish to ask any follow-up questions. Moderators are going to be answering questions as the session progresses as well.
- If you have any questions outside of this lecture, or that are not answered during this lecture, please do submit these for upcoming Open Classes. You can submit these questions here: **Open Class Questions**

# **Software Engineering Lecture Housekeeping** cont.

- For all **non-academic questions**, please submit a query: **www.hyperiondev.com/support**
- Report a **safeguarding** incident: **www.hyperiondev.com/safeguardreporting**
- We would love your **feedback** on lectures: **Feedback on Lectures**

# Prestigious Co-Certification Opportunities

**New Partnerships!**

- **University of Manchester & Imperial College London** join our circle along with The University of Nottingham Online.

**Exclusive Opportunity:**

- Co-certification spots awarded on a first-come basis.
- Meet the criteria early to gain eligibility for the co-certification.

**New Deadlines:**

- **11 March 2024**: 112 GLH & BYB tasks completion.
- **18 March 2024**: Record interview invitation or self-employment.
- **15 July 2024**: Submit verified job offer or new contract.

# Lecture Objectives

1. Define polymorphism and its role in OOP
2. Implement polymorphism into your own classes
3. Define the SOLID principles and the role they play with class creation in OOP

CoGrammar

# Recap on Previous Week

# What are special methods?

- We can use special methods to add and set specific behaviour for built-in python functions.
- We can add behaviour for string representation, mathematical operations, container-like objects and many more.

# Special methods

- We can add string representation using __str__() and __repr__()
- We can add functionality when using math operators with our objects. __add__(), __sub__(), __mul__(), etc.
- We can add container-like functionality to our classes with the special methods __len__(), __setitem__(), __getitem__(), etc.
- Lastly we can set the behaviour for comparators <, >, ==, <= etc. __lt__(), __gt__(), __le__(), __ge__(), etc

# __str__()

```
class Student:

    def __init__(self, fullname, student_number):
        self.fullname = fullname
        self.student_number = student_number

    def __str__(self):
        return f"Fullname:\t{self.fullname}\nStudent Num:\t{self.student_number}\n"

new_student = Student("Percy Jackson", "PJ323423")
print(new_student)
```

# Special Methods And Math

```
class MyNumber:

    def __init__(self, value):
        self.value = value

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

num1 = MyNumber(10)
num2 = MyNumber(5)
num3 = num1 + num2
print(num3.value) # Output: 15
```

# Container-Like Objects

```
class ContactList:

    def __init__(self):
        self.contact_list = []

    def add_contact(self, contact):
        self.contact_list.append(contact)

    def __getitem__(self, key):
        return self.contact_list[key]


contact_list = ContactList()
contact_list.add_contact("Test Contact")
print(contact_list[0]) # Output: Test Contact
```

# Comparators

```python
class Student:

    def __init__(self, fullname, student_number, average):
        self.fullname = fullname
        self.student_number = student_number
        self.average = average

    def __gt__(self, other):
        return self.average > other.average

student1 = Student("Peter Parker", "PP734624", 88)
student2 = Student("Tony Stark", "TS23425", 85)
print(student1 > student2) # Output: True
```

# Polymorphism

# What is Polymorphism?

- A concept in object-oriented programming (OOP) that allows objects of different classes to be treated as objects of a common base class.
- Enables flexibility and extensibility to your code.
- Allows functions and methods to interact with different types of objects without needing to know what type each object is.

# Polymorphism

- We can achieve polymorphism by using method overriding.
- By overriding a method in a subclass we can extend or completely change its behaviour from the base class.
- This allows us to use both the base and sub classes in the same function with each having different behaviour.

# Polymorphism

```
class Animal:

    def make_sound(self):
        print("Animal makes a sound")

class Lion(Animal):

    def make_sound(self):
        print("The Lion goes Raaaawwwrrr")

class Dog(Animal):

    def make_sound(self):
        print("The Dog goes 'Woof... woof... woof' at a very annoying tempo")
```

# Polymorphism

```
animals = [Dog(), Lion(), Lion(), Dog()]
for animal in animals:
    animal.make_sound()
```



```
The Dog goes 'Woof... woof... woof' at a very annoying tempo
The Lion goes Raaaawwwrrr
The Lion goes Raaaawwwrrr
The Dog goes 'Woof... woof... woof' at a very annoying tempo
```

# The SOLID Principles

# What are the SOLID principles?

- Five principles of object oriented class design.
- They are a set of rules and best practices to follow while designing a class structure.
- These principles will help us understand the need for certain design patterns.

# Where do the SOLID principles come from?

- Introduced by Robert J. Martin in a paper he wrote in 2000
- Also known as “Uncle Bob”

- Although he introduced the principles the SOLID acronym was introduced later by Michael Feathers.

# Why do we use the SOLID principles?

- Helps to reduce dependencies.
- Engineers can change one area without impacting others.
- Makes designs easier to understand, maintain and extend.

- **S - Single Responsibility Principle**
  A class should have one, and only one, reason to change.
- **O - Opened Closed Principle**
  You should be able to extend a classes behavior, without modifying it.
- **L - Liskov Substitution Principle**
  Derived classes must be substitutable for their base classes.
- **I - Interface Segregation Principle**
  A class should not be forced to inherit a function in will not use.
- **D - Dependency Inversion Principle**
  Depend on abstractions, not on concretions.

# S - Single Responsibility Principle

A class should have one, and only one, reason to change.

- **A class should only have a single purpose**
- **If a class has to many responsibilities it increases the possibility of bugs as changing one responsibility might affect the others.**

# S - Single Responsibility Principle

## A class should have one, and only one, reason to change.

```python
class Book:
    def __init__(self, title, author, price):
        self.title = title
        self.author = author
        self.price = price

    def change_title(self, new_title):
        # Change title of book
        self.title = new_title

    def change_author(self, new_author):
        # Change author of book
        self.author = new_author

    def process_invoice(self, file, inv_num):
        # Create invoice for sale of book
        file.write(f"Invoice number: {inv_num}"
                   f"Sale of book: {self.title} by {self.author}\n"
                   f"Price: {self.price}")
```

# S - Single Responsibility Principle

## A class should have one, and only one, reason to change.

```python
class Book:
    def __init__(self, title, author, price):
        self.title = title
        self.author = author
        self.price = price

    def change_title(self, new_title):
        # Change title of book
        self.title = new_title

    def change_author(self, new_author):
        # Change author of book
        self.author = new_author
```

```python
class Invoice:
    def __init__(self, inv_num, book):
        self.inv_num = inv_num
        self.book = book

    def process_invoice(self, file):
        # Create invoice for sale of book
        file.write(f"Invoice number: {self.inv_num}"
                   f"Sale of book: {self.book.title} by {self.book.author}\n"
                   f"Price: {self.book.price}")
```

# O - Opened Closed Principle

**You should be able to extend a classes behavior, without modifying it.**

- Open for extension, meaning that the class's behavior can be extended.
- Closed for modification, meaning that the source code is set and cannot be changed.

# O - Opened Closed Principle

**You should be able to extend a classes behavior, without modifying it.**

```python
class Shape:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def calculate_area(self):
        return self.width * self.height
```



```python
class Shape:
    def __init__(self, shape_type, length_list):
        self.shape_type = shape_type
        if shape_type == "Rectangle":
            self.width = length_list[0]
            self.height = length_list[1]
        elif shape_type == "Circle":
            self.radius = length_list[0]

    def calculate_area(self):
        if self.shape_type == "Rectangle":
            return self.width * self.height
        elif self.shape_type == "Circle":
            return pi * self.radius**2
```

# O - Opened Closed Principle

## You should be able to extend a classes behavior, without modifying it.

```python
class Shape:
    def __init__(self, shape_type):
        self.shape_type = shape_type

    def calculate_area(self):
        pass
```

```python
class Rectangle(Shape):
    def __init__(self, width, height):
        super().__init__("Rectangle")
        self.width = width
        self.height = height

    def calculate_area(self):
        return self.width * self.height
```

```python
class Circle(Shape):
    def __init__(self, radius):
        super().__init__("Circle")
        self.radius = radius

    def calculate_area(self):
        return pi * self.radius**2
```

## L - Liskov Substitution Principle

**Derived classes must be substitutable for their base classes.**

- The base class should be able to be replaced with a derived class without the code breaking.
- This is an extension to the open-closed principle as it is also ensuring the derived classes extend the base class without changing behavior

# L - Liskov Substitution Principle

Derived classes must be substitutable for their base classes.

```python
def determine_total_area(shapes):
    total_area = 0
    for shape in shapes:
        total_area += shape.calculate_area
```

```python
class Rectangle(Shape):
    def __init__(self, width, height):
        super().__init__("Rectangle")
        self.width = width
        self.height = height

    def calculate_area(self):
        return self.width * self.height
```

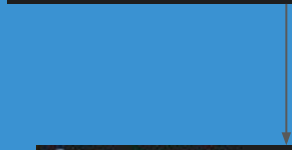
```python
class Circle(Shape):
    def __init__(self, radius):
        super().__init__("Circle")
        self.radius = radius

    def calculate_area(self):
        return pi * self.radius**2
```

```python
class Square(Shape):
    def __init__(self, length):
        self.length = length

    def calculate_area(self):
        return str(self.length**2)
```

# L - Liskov Substitution Principle

**Derived classes must be substitutable for their base classes.**

```python
class Square:
    def __init__(self, length):
        self.length = length

    def calculate_area(self):
        return str(self.length**2)
```



```python
class Square:
    def __init__(self, length):
        self.length = length

    def calculate_area(self):
        return self.length**2
```

# I - Interface Segregation Principle

**A class should not be forced to inherit a function in will not use.**

- A derived class should only inherit functions it will be using and should not be forced to inherit any extra functions.
- This help us avoid the temptation of having one big, general-purpose class.

# I - Interface Segregation Principle

**A class should not be forced to inherit a function in will not use.**

```python
class Printer:
    def normal_print(self, document):
        print(f"Printing {document}")

    def fax(self, document):
        print(f"Faxing {document}")

    def scan(self, document):
        print(f"Scanning {document}")
```



```python
class OldPrinter(Printer):
    def normal_print(self, document):
        print(f"Printing {document}")

    def fax(self, document):
        print(f"Not implemented")

    def scan(self, document):
        print(f"Not implemented")
```

# I - Interface Segregation Principle

**A class should not be forced to inherit a function in will not use.**

```python
class Printer:
    def normal_print(self, document):
        pass
```

```python
class Faxer:
    def fax(self, document):
        pass
```

```python
class Scanner:
    def scan(self, document):
        pass
```

```python
class OldPrinter(Printer):
    def normal_print(self, document):
        print(f"Printing {document}")
```

```python
class ModernPrinter(Printer, Faxer, Scanner):
    def normal_print(self, document):
        print(f"Printing {document}")

    def fax(self, document):
        print(f"Faxing {document}")

    def scan(self, document):
        print(f"Scanning {document}")
```

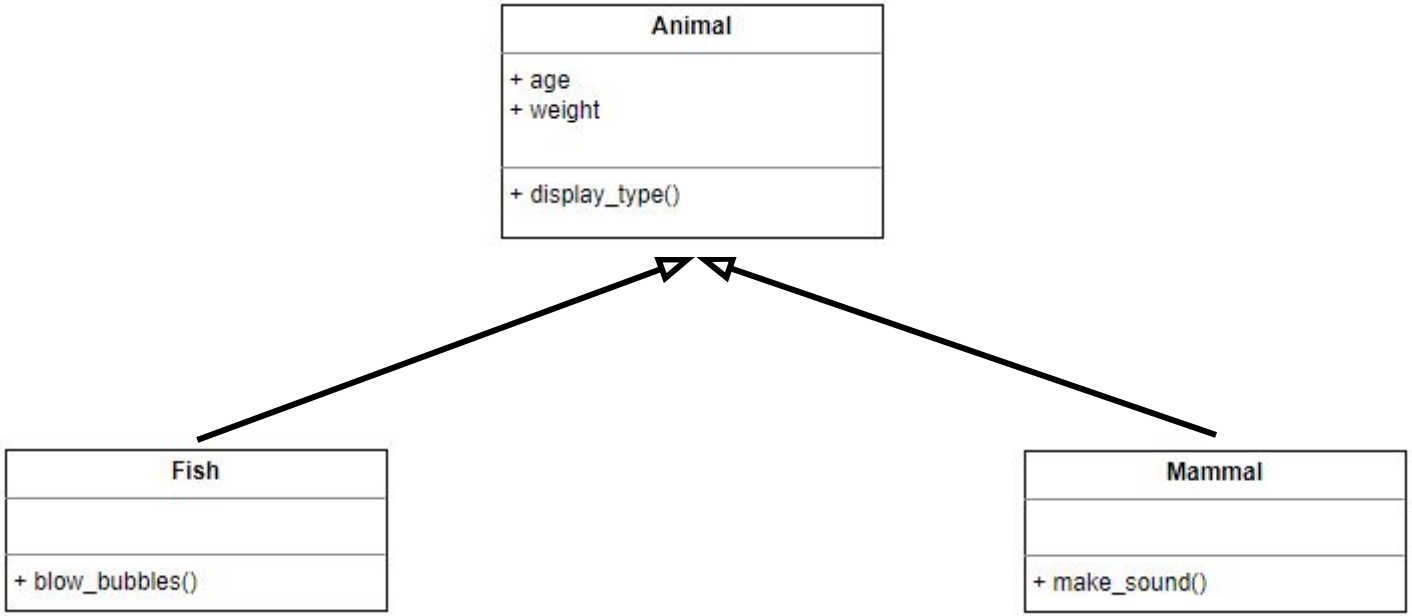
# D - Dependency Inversion Principle

**Depend on abstractions, not on concretions.**

- When creating classes we try to rely on abstraction to stay focused on the classes and what they do rather than how they do it.
- We first determine what the class will do then we worry about the implementation later.

# D - Dependency Inversion Principle

## Depend on abstractions, not on concretions.



CoGrammar

Questions

# CoGrammar

## Thank you for joining