# Object-Oriented Programming II

CSCI 1030U - Intro to Computer Science
@IntroCS

Randy J. Fortier
@randy_fortier

OntarioTech UNIVERSITY

# Outline

- Inheritance
- Method resolution
- Multiple inheritance

OntarioTech UNIVERSITY

# Inheritance

# Inheritance

- A class (B) can inherit members from another class (A)
- We call this relationship by many names:
  - B *is a* A  (e.g. Car *is a* Vehicle)
  - B is a specialization of A
  - A is a generalization of B
  - A is the parent class of B
  - B is a child of A
  - A is a superclass of B
  - B is a sub-class of A

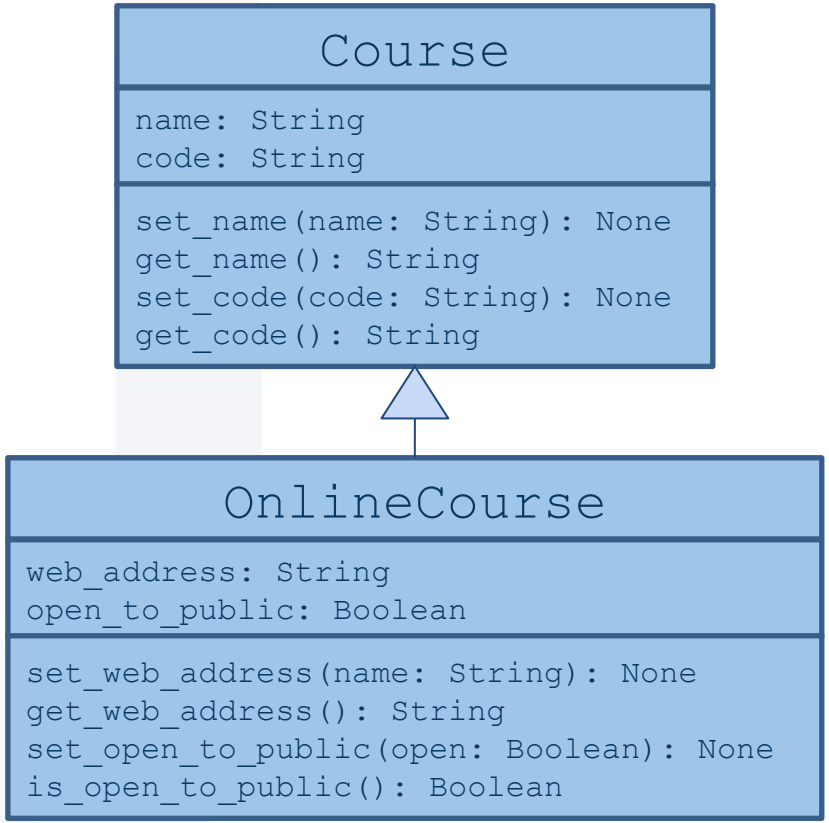
# Inheritance

- Example:
  - Let's say we have a class called `Course`
  - We want to create a class, `OnlineCourse`
    - `OnlineCourse` has its own unique members it needs
      - Web address, open/closed to the public
    - However, `OnlineCourse` also needs everything that is in `Course`
    - Thus, we make `OnlineCourse` a subclass of `Course`, and it inherits all of `Course`'s members

# Inheritance

```
class OnlineCourse(Course):
    def __init__(self, code, name):
        super().__init__(self, code, name)
        self.web_address = ""
        self.open_to_public = False

    def get_web_address(self):
        return self.web_address

    def set_web_address(self, web_address):
        self.web_address = web_address

    def is_open_to_public(self):
        return self.open_to_public

    def set_open_to_public(self, open_to_public):
        self.open_to_public = open_to_public
```

# Inheritance in UML



# Instances with Inheritance

- You can treat an object like any more general class variable (but not more specific)
  - Remember that `OnlineCourse` *is a* `Course`

```
online_course = OnlineCourse()
online_course.set_code("ONLI 1000U")
online_course.set_name("Some Online Course")
student.set_course_grade(online_course, 91.0)
```

OntarioTech UNIVERSITY

# Inheritance

```
01  class Animal:
02      def __init__(self, name):
03          self.name = name
04      def eat(self):
05          print(f'{self.name} eats.')

06  class Lion(Animal):
07      def __init__(self, name):
08          self.name = name
09      def roar(self):
10          print(f'{self.name} roars!')

11  nala = Lion('Nala')
12  nala.roar()
13  nala.eat()
```

# Polymorphism

- The idea of treating an instance of a sub-class the same as an instance of the parent class is called *polymorphism*
  - This is convenient for the programmer, since we can extend and create new classes without having to write code to use those classes

```
printers = [
  InkjetPrinter('Inkblob', 'Inker 2000', '201.74.138.44'),
  LaserPrinter('Laserdude', 'Laserbot 340', '192.77.31.109')
]
for printer in printers:
    printer.print(document)
```

OntarioTech UNIVERSITY

# Method Resolution

- Python uses *dynamic binding* to determine which method is supposed to be called
  - It looks at the class that defined the object first
  - If that class doesn't define a method that matches, it then looks at the parent class
  - This continues up the hierarchy of classes until a match is found

OntarioTech UNIVERSITY

# Method Resolution

- Languages like C++ also support *static binding* to determine which method is supposed to be called
  - This means that it determines which class defines the method during compilation
  - This is more efficient, but not always possible
  - C++ can also support dynamic binding (*virtual* functions)

OntarioTech UNIVERSITY

# Coding Exercise 06b.1

- Write a set of three classes that represent:
  - A dog (`Dog`)
  - A cat (`Cat`)
  - A generic pet (`Pet`)

```
pets = [
  Dog('Rufus', 'Husky', 8.0, 'female'),
  Cat('Boots', 'Long hair', 3.2, 'male')
]
for pet in pets:
  pet.speak()

# Rufus: Woof!
# Boots: Meow!
```

OntarioTech UNIVERSITY

# Coding Exercise 06b.2

- Write a class, `Shoe`, that represents a shoe for sale at an online shoe store, and a parent class, `Product`
- Instance variables:
  - `Product: price, description`
  - `Shoe: brand, size, colour`
- Methods for `Shoe`:
  - Constructor
  - A string converter (`__str__`), which returns a string representation

OntarioTech UNIVERSITY

# Multiple Inheritance

OntarioTech UNIVERSITY

# Multiple Inheritance

- Classes can inherit from multiple parent classes

e.g.

```
class Printer:
    pass

class Scanner:
    pass

class AllInOne(Printer, Scanner):
    def __init__(self):
        Printer.__init__(self)
        Scanner.__init__(self)
```

OntarioTech UNIVERSITY

# Hacker's Corner: Mix-ins

- A mix-in class is a class that you can add to another class (using inheritance or another mechanism) to add functionality that many classes might need

```
class Howler:
    def speak(self):
        print('Aaaooooo!')

class Dog(Pet, Howler):
    pass

class Wolf(WildAnimal, Howler):
    pass
```

OntarioTech UNIVERSITY

# Wrap-up

- Inheritance
- Method resolution
- Multiple inheritance

OntarioTech UNIVERSITY

# Coming Up

- Finite state automata
- Finite state automata types
  - Deterministic FSAs (DFAs)
  - Push-down Automata (PDAs)
  - Non-deterministic FSAs (NFAs)
- Parsers (lexical analysis)
- Conway's Game of Life

OntarioTech UNIVERSITY