Python - OOP

```
module = Module(
    code="ELEE1147",
    name="Programming for Engineers",
    credits=15,
    module_leader="Seb Blair BEng(H) PGCAP MIET MIHEEM FHEA"
)
```

Download as a PDF

UNIVERSITY OF
GREENWICH

Object-Oriented Programming in Python

- **What is OOP?**
  - A programming paradigm based on "objects"
  - Combines **data** and **methods** into single entities
  - Focuses on **modularity** and **reusability**

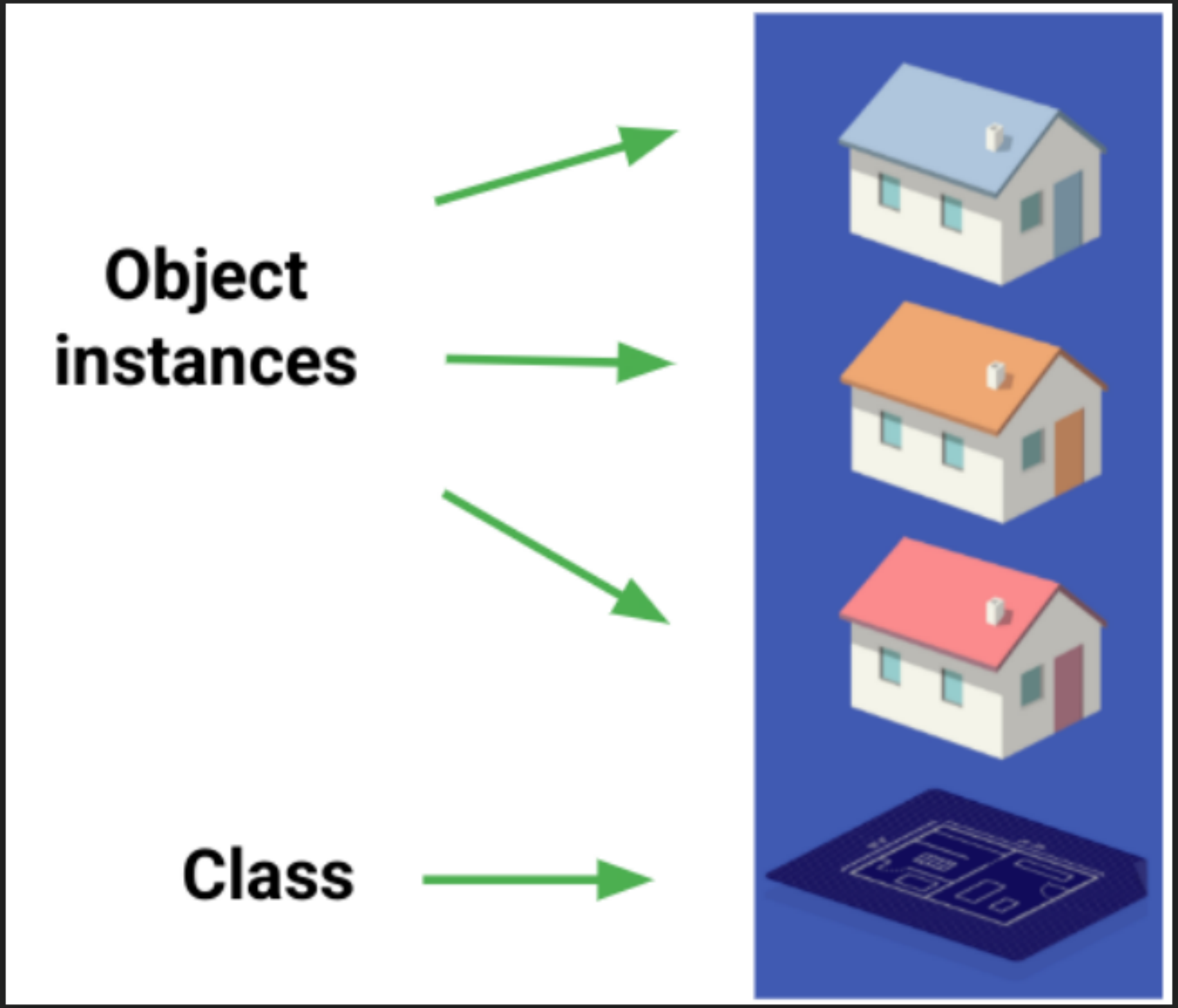
- **Key Features:**
  - Encapsulation
  - Inheritance
  - Polymorphism
  - Abstraction

Classes

Classes

- **Classes** are software programming models
  - abstractions of the real world or system entities.

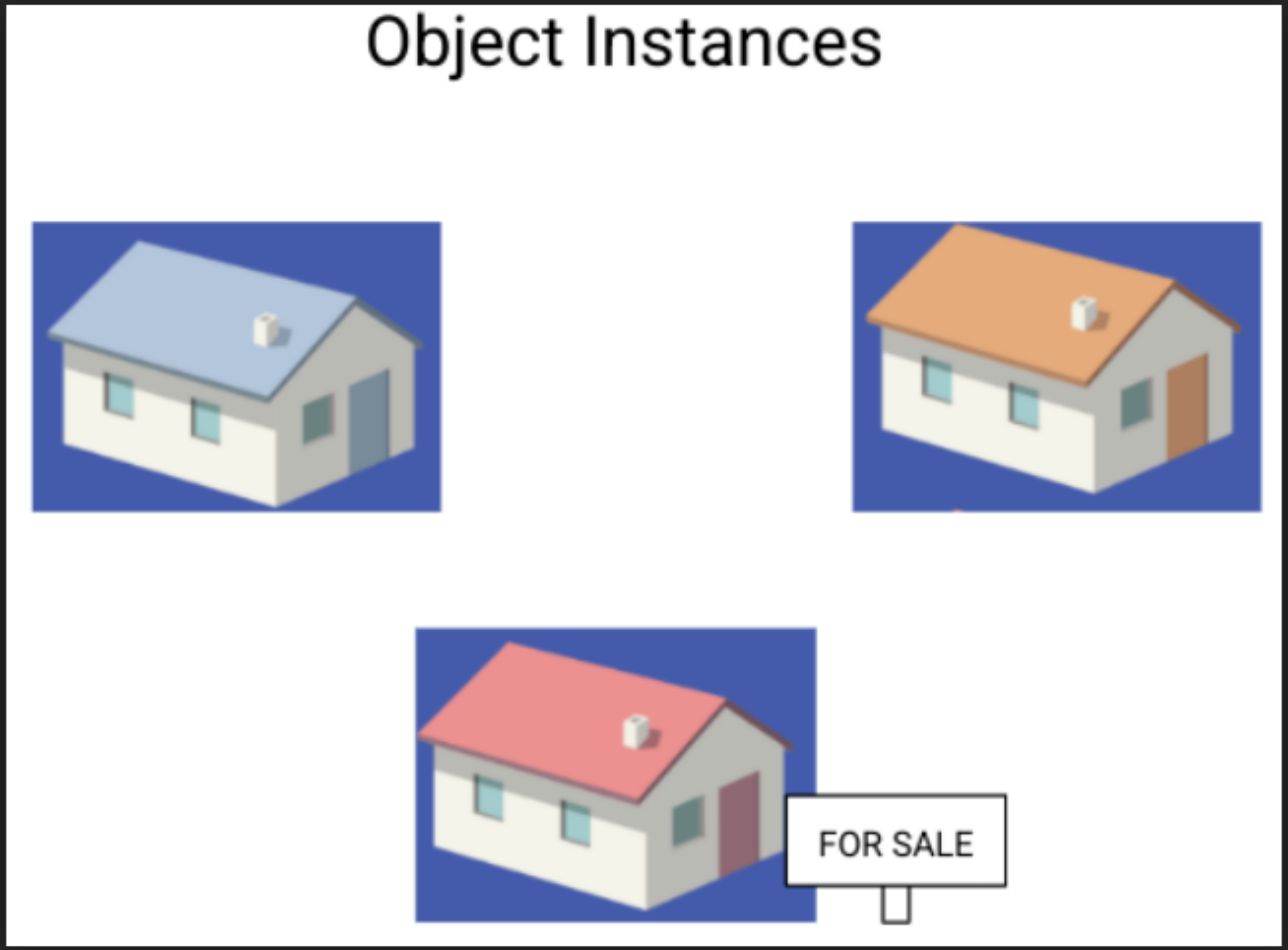
- **Classes** define methods that operate on their object instances



Classes vs Objects (2)

House Class

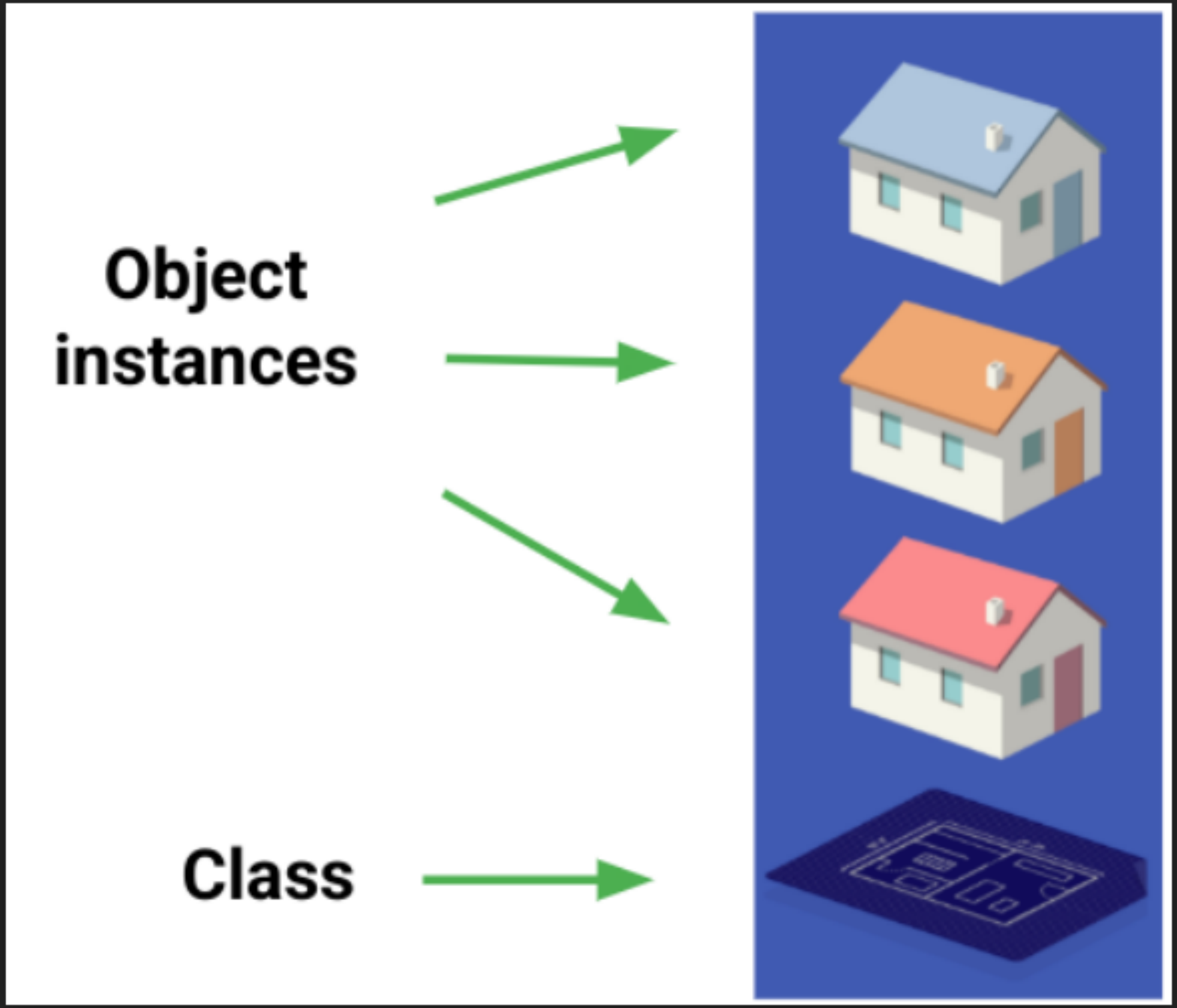
- Data
    - House color (`String`)
    - Number of windows (`Number`)
    - Is for sale (`Boolean`)
- Behavior
    - `updateColor()`
    - `putOnSale()`



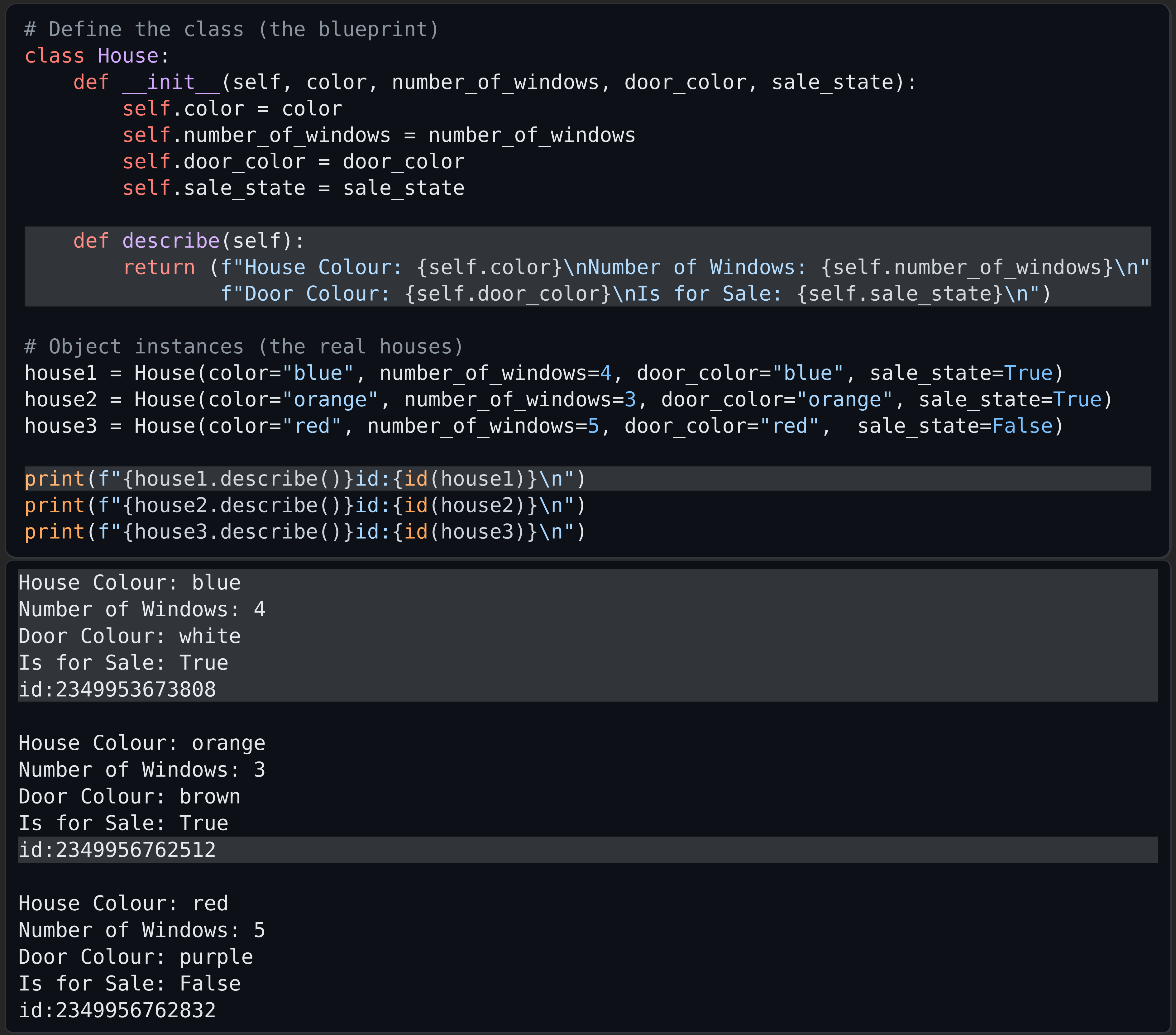
Python - OOP > Classes

```python
# Define the class (the blueprint)
class House:
    def __init__(self, color, number_of_windows, door_color, sale_state):
        self.color = color
        self.number_of_windows = number_of_windows
        self.door_color = door_color
        self.sale_state = sale_state

    def describe(self):
        return (f"House Colour: {self.color}\nNumber of Windows: {self.number_of_windows}\n"
                f"Door Colour: {self.door_color}\nIs for Sale: {self.sale_state}\n")

# Object instances (the real houses)
house1 = House(color="blue", number_of_windows=4, door_color="blue", sale_state=True)
house2 = House(color="orange", number_of_windows=3, door_color="orange", sale_state=True)
house3 = House(color="red", number_of_windows=5, door_color="red",  sale_state=False)

print(f"{house1.describe()}id:{id(house1)}\n")
print(f"{house2.describe()}id:{id(house2)}\n")
print(f"{house3.describe()}id:{id(house3)}\n")
```

```
House Colour: blue
Number of Windows: 4
Door Colour: white
Is for Sale: True
id:2349953673808

House Colour: orange
Number of Windows: 3
Door Colour: brown
Is for Sale: True
id:2349956762512

House Colour: red
Number of Windows: 5
Door Colour: purple
Is for Sale: False
id:2349956762832
```

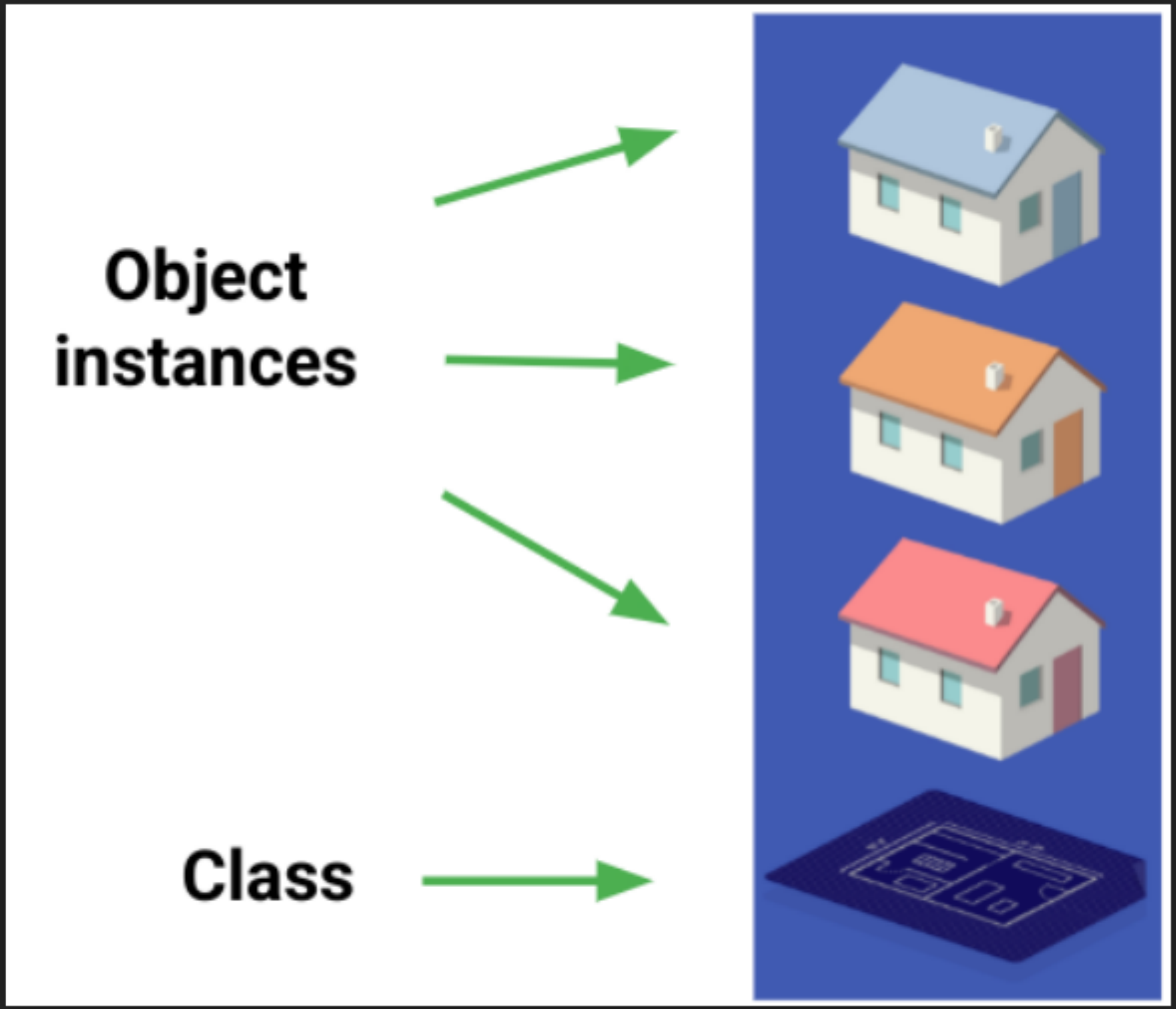


Python - OOP > Classes



```python
# Define the class (the blueprint)
class House:
    def __init__(self, color, number_of_windows, door_color, sale_state):
        self.color = color
        self.number_of_windows = number_of_windows
        self.door_color = door_color
        self.sale_state = sale_state

    def describe(self):
        return (f"House Colour: {self.color}\nNumber of Windows: {self.number_of_windows}\n"
                f"Door Colour: {self.door_color}\nIs for Sale: {self.sale_state}\n")

# Object instances (the real houses)
house1 = House(color="blue", number_of_windows=4, door_color="blue", sale_state=True)
house2 = House(color="orange", number_of_windows=3, door_color="orange", sale_state=True)
house3 = House(color="red", number_of_windows=5, door_color="red",  sale_state=False)

print(f"{house1.describe()}id:{id(house1)}\n")
print(f"{house2.describe()}id:{id(house2)}\n")
print(f"{house3.describe()}id:{id(house3)}\n")
```

```
House Colour: blue
Number of Windows: 4
Door Colour: white
Is for Sale: True
id:2349953673808

House Colour: orange
Number of Windows: 3
Door Colour: brown
Is for Sale: True
id:2349956762512

House Colour: red
Number of Windows: 5
Door Colour: purple
Is for Sale: False
id:2349956762832
```



Encapsulation

- **Definition**:
  - Bundling data and methods into a single unit (class) and restricting direct access to some components.

- Example:
  - Use **getter and setter methods** to access private attributes.

```python
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # Private attribute

    # Setter
    def deposit(self, amount):
        self.__balance += amount

    # Getter
    def get_balance(self):
        return self.__balance
```

Inheritance

- **Definition**: A mechanism to derive a class from another class.

- Python supports single and multiple inheritance.

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

    def eat(self):
        print("I can eat!")

    def sleep(self):
        print("I can sleep")

class Dog(Animal):
    def __init__(self, name, breed):
        # Call the parent class's __init__
        super().__init__(name)
        self.breed = breed

    def bark(self):
        print("Woof!")

dog = Dog("Alfski", "Norwegian Elkhound")
print(dog.name)  # Alfski
print(dog.breed)  # Norwegian Elkhound
print(dog.bark)  # Woof!
```

Inheritance

- keyword: `super()` gives you access to the parent class you inherited

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

    def eat(self):
        print("I can eat!")

    def sleep(self):
        print("I can sleep")

class Dog(Animal):
    def __init__(self, name, breed):
        # Call the parent class's __init__
        super().__init__(name)
        self.breed = breed

    def bark(self):
        print("Woof!")

dog = Dog("Alfski", "Norwegian Elkhound")
print(dog.name)  # Alfski
print(dog.breed)  # Norwegian Elkhound
print(dog.bark)  # Woof!
```

Inheritance

- Expand with additional functions for this class

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

    def eat(self):
        print("I can eat!")

    def sleep(self):
        print("I can sleep")

class Dog(Animal):
    def __init__(self, name, breed):
        # Call the parent class's __init__
        super().__init__(name)
        self.breed = breed

    def bark(self):
        print("Woof!")

dog = Dog("Alfski", "Norwegian Elkhound")
print(dog.name)  # Alfski
print(dog.breed)  # Norwegian Elkhound
print(dog.bark)  # Woof!
```

Polymorphism

- **Definition**: The ability of objects to take many forms.

```python
class Bird:
    def fly(self):
        print("Bird flies")

class Penguin(Bird):
    def fly(self):
        print("Penguins cannot fly")

def test_fly(bird):
    bird.fly()

test_fly(Bird())    # Bird flies
test_fly(Penguin())  # Penguins cannot fly
```

Abstraction

- **Definition**:
    - Hiding implementation details while showing essential features.
- Achieved using abstract base classes.

```python
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def area(self):
        pass ## keyword for placeholder

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14 * self.radius ** 2

circle = Circle(5)
print(circle.area())  # 78.5
```

Special Methods in Python

- **Magic/Dunder Methods**: Special methods with double underscores.
- Examples:
  - `__init__`: Initialize an object.
  - `__str__`: String representation of an object.
  - `__add__`: Define addition behavior for objects.

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

    def __add__(self, other):
        return Vector(self.x + other.x, self.y + other.y)

    def __str__(self):
        return f"Vector({self.x}, {self.y})"

v1 = Vector(2, 3)
v2 = Vector(5, 6)
print(v1 + v2)  # Vector(7, 9)
```

Why Use OOP?

- **Benefits:**
  - Code reusability through inheritance.
  - Modularity for easier debugging and maintenance.
  - Encapsulation enhances data security.
  - Polymorphism makes systems more flexible.
- **Real-World Use Cases:**
  - GUI applications
  - Games
  - Web frameworks

Summary

- OOP provides a structured and modular way of programming.
- Key concepts:
    - Classes and Objects
    - Encapsulation
    - Inheritance
    - Polymorphism
    - Abstraction
    - `super()` for parent class method calls
- Use OOP for scalable and maintainable code.