COMP110: Principles of Computing

6: Object oriented programming

Learning outcomes

- **Explain** the key concepts of OOP, including objects, classes, fields, methods, inheritance and polymorphism
- **Use** objects to model systems
- **Write** simple object oriented programs

FALMOUTH
UNIVERSITY

Classes and objects

A non-object-oriented program

Clone the `bsc-live-coding` repository to your local machine:

- Open an appropriate folder (e.g. on the `X:` drive) and right-click in empty space
- Select **Git Clone...**
- For the URL, enter `https://github.com/Falmouth-Games-Academy/bsc-live-coding.git`
- Click OK

Once it has finished downloading, open `bsc-live-coding\COMP110\06_OOP` in PyCharm

What's wrong with this program?

What's wrong with this program?

- Data for a single "thing" (a ball) is spread across multiple data structures

What's wrong with this program?

- Data for a single "thing" (a ball) is spread across multiple data structures
- It's messy — we have to type a lot of list indexing expressions

What's wrong with this program?

- Data for a single "thing" (a ball) is spread across multiple data structures
- It's messy — we have to type a lot of list indexing expressions
- It's inefficient — all that list indexing takes time

What's wrong with this program?

- Data for a single "thing" (a ball) is spread across multiple data structures
- It's messy — we have to type a lot of list indexing expressions
- It's inefficient — all that list indexing takes time
- It's error-prone — if we start inserting or removing elements, the lists can easily get out of step with each other

A better approach

A better approach

- We can use a **class** to collect the data for a single ball

A better approach

- We can use a **class** to collect the data for a single ball
- A class has **fields**, each of which is essentially a variable

A better approach

- We can use a **class** to collect the data for a single ball
- A class has **fields**, each of which is essentially a variable
- Syntax:

```
class Ball:
    def __init__(self):
        self.pos_x = ...
        self.pos_y = ...
```

A better approach

- We can use a **class** to collect the data for a single ball
- A class has **fields**, each of which is essentially a variable
- Syntax:

```
class Ball:
    def __init__(self):
        self.pos_x = ...
        self.pos_y = ...
```

- We use a class by creating **instances** of it

The constructor

```
class Ball:
    def __init__(self):
        self.pos_x = ...
        self.pos_y = ...
```

The constructor

```
class Ball:
    def __init__(self):
        self.pos_x = ...
        self.pos_y = ...
```

- The **constructor** or **initialiser** is called when an instance of the class is created

The constructor

```
class Ball:
    def __init__(self):
        self.pos_x = ...
        self.pos_y = ...
```

- The **constructor** or **initialiser** is called when an instance of the class is created
- Must be named `__init__`

The constructor

```
class Ball:
    def __init__(self):
        self.pos_x = ...
        self.pos_y = ...
```

- The **constructor** or **initialiser** is called when an instance of the class is created
- Must be named `__init__`
  - Note the double underscores: `_ _ i n i t _ _`

The constructor

```
class Ball:
    def __init__(self):
        self.pos_x = ...
        self.pos_y = ...
```

- The **constructor** or **initialiser** is called when an instance of the class is created
- Must be named `__init__`
  - Note the double underscores: `_ _ i n i t _ _`
- First parameter must be `self`, and is the instance being created

The constructor

```
class Ball:
    def __init__(self):
        self.pos_x = ...
        self.pos_y = ...
```

- The **constructor** or **initialiser** is called when an instance of the class is created
- Must be named `__init__`
  - Note the double underscores: `_ _ i n i t _ _`
- First parameter must be `self`, and is the instance being created
- Other parameters are optional

Fields

```
class Ball:
    def __init__(self):
        self.pos_x = ...
        self.pos_y = ...
```

Fields

```
class Ball:
    def __init__(self):
        self.pos_x = ...
        self.pos_y = ...
```

- In Python, **fields** are defined by **assigning** to them

Fields

```
class Ball:
    def __init__(self):
        self.pos_x = ...
        self.pos_y = ...
```

- In Python, **fields** are defined by **assigning** to them
  - Just like ordinary variables

Fields

```
class Ball:
    def __init__(self):
        self.pos_x = ...
        self.pos_y = ...
```

- In Python, **fields** are defined by **assigning** to them
  - Just like ordinary variables
- Usually define all fields in `__init__`

Fields

```
class Ball:
    def __init__(self):
        self.pos_x = ...
        self.pos_y = ...
```

- In Python, **fields** are defined by **assigning** to them
  - Just like ordinary variables
- Usually define all fields in `__init__`
  - It's possible to define new fields after, but for maintainability it is better to collect them all in the same place

Instantiating

Instantiating

- To create an instance of a class, call the class as if it was a function:

```
ball = Ball()
```

Instantiating

- To create an instance of a class, call the class as if it was a function:

```
ball = Ball()
```

- If `__init__` takes parameters other than `self`, specify them here

Instantiating

- To create an instance of a class, call the class as if it was a function:

```
ball = Ball()
```

- If `__init__` takes parameters other than `self`, specify them here
- To access fields, use **dot notation**:

```
ball.pos_x += 10
print ball.pos_x
```

Live coding

Methods

Methods

- As well as **fields**, classes can also contain **methods**

Methods

- As well as **fields**, classes can also contain **methods**
- A method is simply a **function** which operates on a particular instance of the class

Methods

- As well as **fields**, classes can also contain **methods**
- A method is simply a **function** which operates on a particular instance of the class
- Syntax is similar to regular Python functions:

```
class Ball:
    def update(self):
        print "Do something"
```

Methods

- As well as **fields**, classes can also contain **methods**
- A method is simply a **function** which operates on a particular instance of the class
- Syntax is similar to regular Python functions:

```
class Ball:
    def update(self):
        print "Do something"
```

- Methods take `self` as their first parameter, can optionally take other parameters

Methods

- As well as **fields**, classes can also contain **methods**
- A method is simply a **function** which operates on a particular instance of the class
- Syntax is similar to regular Python functions:

```
class Ball:
    def update(self):
        print "Do something"
```

- Methods take `self` as their first parameter, can optionally take other parameters
- `self` can be used inside the method to access fields

Methods

- As well as **fields**, classes can also contain **methods**
- A method is simply a **function** which operates on a particular instance of the class
- Syntax is similar to regular Python functions:

```
class Ball:
    def update(self):
        print "Do something"
```

- Methods take `self` as their first parameter, can optionally take other parameters
- `self` can be used inside the method to access fields
- Use dot notation to call methods

Live coding

Recap of terminology

Recap of terminology

- An **object** is a collection of **fields** which store data, and **methods** which act on that data

Recap of terminology

- An **object** is a collection of **fields** which store data, and **methods** which act on that data
  - Fields can also be called **attributes** or **member variables**

Recap of terminology

- An **object** is a collection of **fields** which store data, and **methods** which act on that data
  - Fields can also be called **attributes** or **member variables**
  - Methods can also be called **member functions**

Recap of terminology

- An **object** is a collection of **fields** which store data, and **methods** which act on that data
  - Fields can also be called **attributes** or **member variables**
  - Methods can also be called **member functions**
- The fields and methods available on an object are determined by its **class**, i.e. its **type**

Recap of terminology

- An **object** is a collection of **fields** which store data, and **methods** which act on that data
  - Fields can also be called **attributes** or **member variables**
  - Methods can also be called **member functions**
- The fields and methods available on an object are determined by its **class**, i.e. its **type**
- A class is a "blueprint" for an object; an **instance** is the object itself

Encapsulation

Good OOP design allows for:

Encapsulation

Good OOP design allows for:

- Related code and data definitions to be collected in a single place

Encapsulation

Good OOP design allows for:

- Related code and data definitions to be collected in a single place
- Development of modular reusable components

Encapsulation

Good OOP design allows for:

- Related code and data definitions to be collected in a single place
- Development of modular reusable components
- Decoupling of object behaviour from implementation details

FALMOUTH
UNIVERSITY

Inheritance and polymorphism

Different shapes

Different shapes

Handling of multiple shapes is currently not ideal

Different shapes

Handling of multiple shapes is currently not ideal

- `draw` method has a big `if-elif` block

Different shapes

Handling of multiple shapes is currently not ideal

- `draw` method has a big `if-elif` block
- Imagine if we also had to do e.g. collision detection – we'd need another `if-elif` block

Different shapes

Handling of multiple shapes is currently not ideal

- `draw` method has a big `if-elif` block
- Imagine if we also had to do e.g. collision detection – we'd need another `if-elif` block
- If we add a new shape, we have to remember to update all of these blocks

A better way?

A better way?

- We could define multiple classes, e.g. `SquareBall`, `CircleBall`, `TriangleBall`

A better way?

- We could define multiple classes, e.g. `SquareBall`, `CircleBall`, `TriangleBall`
- But they would have common code (e.g. `update` method) that we would need to copy and paste...

Inheritance

Inheritance

- Classes can **inherit** from other classes

```
class SquareBall(Ball):
    def __init__(self):
        ...
```

Inheritance

- Classes can **inherit** from other classes

```
class SquareBall(Ball):
    def __init__(self):
        ...
```

- Here `Ball` is the **base class**, `SquareBall` is the **derived class** or **subclass**

Inheritance

- Classes can **inherit** from other classes

```
class SquareBall(Ball):
    def __init__(self):
        ...
```

- Here `Ball` is the **base class**, `SquareBall` is the **derived class** or **subclass**
- The subclass automatically has all fields and methods from the base class

Inheritance

- Classes can **inherit** from other classes

```
class SquareBall(Ball):
    def __init__(self):
        ...
```

- Here `Ball` is the **base class**, `SquareBall` is the **derived class** or **subclass**
- The subclass automatically has all fields and methods from the base class
- The subclass can add new fields and methods

Inheritance

- Classes can **inherit** from other classes

```
class SquareBall(Ball):
    def __init__(self):
        ...
```

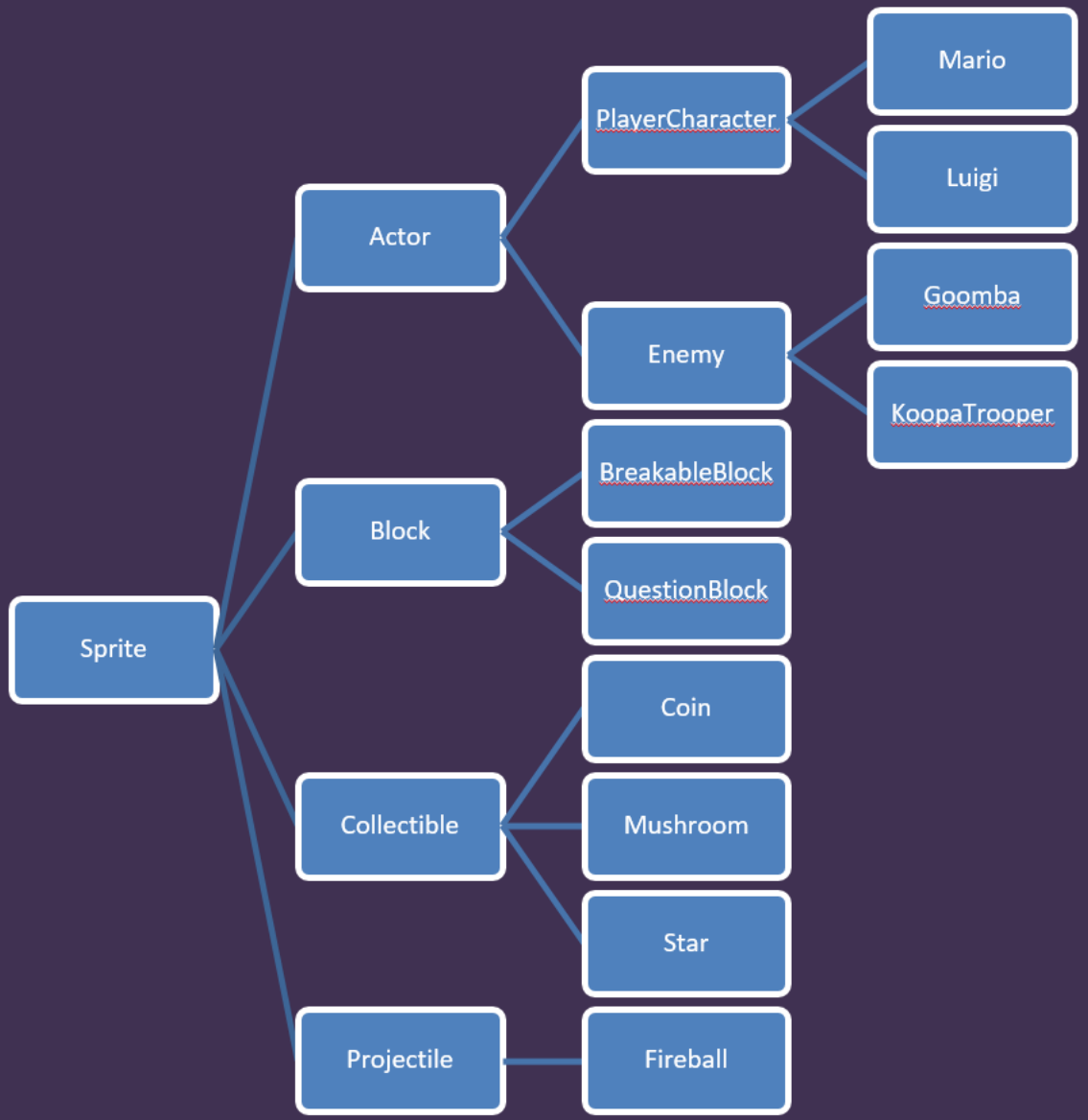
- Here `Ball` is the **base class**, `SquareBall` is the **derived class** or **subclass**
- The subclass automatically has all fields and methods from the base class
- The subclass can add new fields and methods
- The subclass can also **override** methods from the base class

Live coding

Inheritance hierarchy

Inheritance hierarchy

Inheritance hierarchy



Polymorphism

```
ball.draw(screen)
```

Polymorphism

```
ball.draw(screen)
```

- Here `ball` might be an instance of `SquareBall`, `CircleBall` or `TriangleBall`

Polymorphism

```
ball.draw(screen)
```

- Here `ball` might be an instance of `SquareBall`, `CircleBall` or `TriangleBall`
- Whichever it is, Python will execute the appropriate `draw` method

Polymorphism

```
ball.draw(screen)
```

- Here `ball` might be an instance of `SquareBall`, `CircleBall` or `TriangleBall`
- Whichever it is, Python will execute the appropriate `draw` method
- The author of this code doesn't need to worry about squares, circles or triangles

Polymorphism

```
ball.draw(screen)
```

- Here `ball` might be an instance of `SquareBall`, `CircleBall` or `TriangleBall`
- Whichever it is, Python will execute the appropriate `draw` method
- The author of this code doesn't need to worry about squares, circles or triangles
- If we added a new shape class, everything works automatically

Polymorphism

```
ball.draw(screen)
```

- Here `ball` might be an instance of `SquareBall`, `CircleBall` or `TriangleBall`
- Whichever it is, Python will execute the appropriate `draw` method
- The author of this code doesn't need to worry about squares, circles or triangles
- If we added a new shape class, everything works automatically
- This is **polymorphism** — the same code can use objects of many different classes

Polymorphism

```
ball.draw(screen)
```

- Here `ball` might be an instance of `SquareBall`, `CircleBall` or `TriangleBall`
- Whichever it is, Python will execute the appropriate `draw` method
- The author of this code doesn't need to worry about squares, circles or triangles
- If we added a new shape class, everything works automatically
- This is **polymorphism** — the same code can use objects of many different classes
  - From Greek: "many-shape-ism"

Abstract classes and methods

Abstract classes and methods

- It no longer makes sense to instantiate `Ball` directly — it exists only as a base class for other classes

Abstract classes and methods

- It no longer makes sense to instantiate `Ball` directly — it exists only as a base class for other classes
- `Ball` is an **abstract class**

Abstract classes and methods

- It no longer makes sense to instantiate `Ball` directly — it exists only as a base class for other classes
- `Ball` is an **abstract class**
- Similarly, `Ball.draw()` should never be called directly — all subclasses should override it

Abstract classes and methods

- It no longer makes sense to instantiate `Ball` directly — it exists only as a base class for other classes
- `Ball` is an **abstract class**
- Similarly, `Ball.draw()` should never be called directly — all subclasses should override it
- `Ball.draw()` is an **abstract method**

FALMOUTH
UNIVERSITY

Worksheet C

Worksheet C

- Upload your work to GitHub **now**!
- If you still have an open pull request for worksheet B, it will be automatically updated — you don't need to create a new one
- (in fact, GitHub won't let you create a new one...)