COMP130: Game Architecture

3: Advanced OOP Design

Learning outcomes

- Outcome 1
- Outcome 2
- Outcome 3

Relationships

Relationships

- OOP models three types of real-world relationships: **is a**, **has a** and **is a type of**

Relationships

- OOP models three types of real-world relationships: **is a**, **has a** and **is a type of**
  - Donald **is a** duck

Relationships

- OOP models three types of real-world relationships: **is a**, **has a** and **is a type of**
  - Donald **is a** duck
  - A duck **has a** bill

Relationships

- OOP models three types of real-world relationships: **is a**, **has a** and **is a type of**
  - Donald **is a** duck
  - A duck **has a** bill
  - A duck **is a type of** bird

Is-a $\rightarrow$ Instantiation

Is-a $\rightarrow$ Instantiation

- "X is a Y" means "the specific object X is an object of the type Y"

Is-a $\rightarrow$ Instantiation

- "X is a Y" means "the specific object X is an object of the type Y"
- Is-a is modelled by **classes** and **instances**:

Is-a $\rightarrow$ Instantiation

- "X is a Y" means "the specific object X is an object of the type Y"
- Is-a is modelled by **classes** and **instances**:
  - "Donald is a duck" $\rightarrow$ "`donald` is an **instance** of the **class** `Duck`"

Has-a $\rightarrow$ Composition

Has-a $\rightarrow$ Composition

- "X has a Y" means "an object of type X **possesses** an object of type Y"

Has-a $\rightarrow$ Composition

- "X has a Y" means "an object of type X **possesses** an object of type Y"
- OOP models this by having a **field** on X which holds an instance of Y

Has-a $\rightarrow$ Composition

- "X has a Y" means "an object of type X **possesses** an object of type Y"
- OOP models this by having a **field** on X which holds an instance of Y
  - "A duck has a bill" $\rightarrow$ "The class `Duck` has a **field** which contains an instance of the class `Bill`"

Is-a-type-of $\rightarrow$ Inheritance

Is-a-type-of $\rightarrow$ Inheritance

- "X is a type of Y" means "If an object is of type X, then it is **also** of type Y"

Is-a-type-of $\rightarrow$ Inheritance

- "X is a type of Y" means "If an object is of type X, then it is **also** of type Y"
  - "A duck is a type of bird" $\rightarrow$ "If something is a duck, then it is also a bird"

Is-a-type-of $\rightarrow$ Inheritance

- "X is a type of Y" means "If an object is of type X, then it is **also** of type Y"
  - "A duck is a type of bird" $\rightarrow$ "If something is a duck, then it is also a bird"
  - "Every duck is a bird"

Is-a-type-of $\rightarrow$ Inheritance

- "X is a type of Y" means "If an object is of type X, then it is **also** of type Y"
  - "A duck is a type of bird" $\rightarrow$ "If something is a duck, then it is also a bird"
  - "Every duck is a bird"
  - "If something is true for all birds, then it must be true for ducks"

Is-a-type-of $\rightarrow$ Inheritance

- "X is a type of Y" means "If an object is of type X, then it is **also** of type Y"
  - "A duck is a type of bird" $\rightarrow$ "If something is a duck, then it is also a bird"
  - "Every duck is a bird"
  - "If something is true for all birds, then it must be true for ducks"
- In OOP terms, this is called **inheritance**

Inheritance

Inheritance

- Recall: an **object** is a collection of **fields** (data) and **methods** (code)

Inheritance

- Recall: an **object** is a collection of **fields** (data) and **methods** (code)
- Recall: the **class** defines which fields and methods an object possesses

Inheritance

- Recall: an **object** is a collection of **fields** (data) and **methods** (code)
- Recall: the **class** defines which fields and methods an object possesses
- "X is a type of Y" $\rightarrow$ class `X` inherits from class `Y`

Inheritance

- Recall: an **object** is a collection of **fields** (data) and **methods** (code)
- Recall: the **class** defines which fields and methods an object possesses
- "X is a type of Y" $\rightarrow$ class `X` inherits from class `Y`
- Class X inherits all of the fields and methods from class Y, as well as any fields and methods of its own

When to inherit?

When to inherit?

- When modelling an **is-a-type-of** relationship from the real world

When to inherit?

- When modelling an **is-a-type-of** relationship from the real world
- When several classes can **share** some fields and/or methods

When to inherit?

- When modelling an **is-a-type-of** relationship from the real world
- When several classes can **share** some fields and/or methods
  - I.e. to minimise **code duplication**

When to inherit?

- When modelling an **is-a-type-of** relationship from the real world
- When several classes can **share** some fields and/or methods
  - I.e. to minimise **code duplication**
- When several classes should have methods with the **same names**, but which do **different things**

When to inherit?

- When modelling an **is-a-type-of** relationship from the real world
- When several classes can **share** some fields and/or methods
  - I.e. to minimise **code duplication**
- When several classes should have methods with the **same names**, but which do **different things**
  - This is called **polymorphism** — more on this later

Inheritance in Python

Inheritance in Python

Inheritance in Python

Inheritance in Python

Inheritance in Python

Chains of inheritance

Chains of inheritance

- "A mallard is a type of duck, which is a type of bird, which is a type of vertebrate, which is a type of animal..."

Chains of inheritance

- "A mallard is a type of duck, which is a type of bird, which is a type of vertebrate, which is a type of animal..."
- Is-a-type-of is **transitive**

Chains of inheritance

- "A mallard is a type of duck, which is a type of bird, which is a type of vertebrate, which is a type of animal..."
- Is-a-type-of is **transitive**
  - If A is-a-type-of B and B is-a-type-of C, then A is-a-type-of C

Chains of inheritance

- "A mallard is a type of duck, which is a type of bird, which is a type of vertebrate, which is a type of animal..."
- Is-a-type-of is **transitive**
  - If A is-a-type-of B and B is-a-type-of C, then A is-a-type-of C
- Likewise: class A inherits from class B, which inherits from class C, ...

Chains of inheritance

- "A mallard is a type of duck, which is a type of bird, which is a type of vertebrate, which is a type of animal..."
- Is-a-type-of is **transitive**
  - If A is-a-type-of B and B is-a-type-of C, then A is-a-type-of C
- Likewise: class A inherits from class B, which inherits from class C, ...
  - "Inherits from" is also transitive

A possible inheritance hierarchy

OOP: Polymorphism

Polymorphism

Polymorphism

- From Greek: “many-shape-ism”

Polymorphism

- From Greek: “many-shape-ism”
- Different classes can have the **same public interface**

Polymorphism

- From Greek: "many-shape-ism"
- Different classes can have the **same public interface**
- Thus we can write code that **uses** this interface, but doesn't need to worry about the **implementation** behind it

Method overriding

Method overriding

- A class can **override** methods defined in the class from which it inherits

Method overriding

- A class can **override** methods defined in the class from which it inherits
- The overridden method can call the method from the base class, but it doesn't have to

Without polymorphism

Without polymorphism

- We have a list of shapes, and want to draw them all

Without polymorphism

- We have a list of shapes, and want to draw them all
- This approach is messy and difficult to maintain

Polymorphism to the rescue!

Polymorphism to the rescue!

- All subclasses of `Shape` implement `draw`

Polymorphism to the rescue!

- All subclasses of `Shape` implement `draw`
- We can call `shape->draw()` without worrying which type of shape it is

Abstract classes and methods

Abstract classes and methods

- Some classes should never be instantiated directly, as they only exist to be inherited from

Abstract classes and methods

- Some classes should never be instantiated directly, as they only exist to be inherited from
  - `Shape` is an example

Abstract classes and methods

- Some classes should never be instantiated directly, as they only exist to be inherited from
  - `Shape` is an example
- Such classes are called **abstract**

Abstract classes and methods

- Some classes should never be instantiated directly, as they only exist to be inherited from
  - `Shape` is an example
- Such classes are called **abstract**
- Abstract methods are methods of an abstract class which are left unimplemented, so **must** be implemented in subclasses

Abstract classes and methods

- Some classes should never be instantiated directly, as they only exist to be inherited from
  - `Shape` is an example
- Such classes are called **abstract**
- Abstract methods are methods of an abstract class which are left unimplemented, so **must** be implemented in subclasses
  - `draw` is an example

OOP: Access control

Access control

Access control

- For **encapsulation**, it is a good idea to restrict access to certain attributes and methods from outside the class

Access control

- For **encapsulation**, it is a good idea to restrict access to certain attributes and methods from outside the class
- **Private** members are only accessible from the class's **own** methods

Access control

- For **encapsulation**, it is a good idea to restrict access to certain attributes and methods from outside the class
- **Private** members are only accessible from the class's **own** methods
- **Protected** members are accessible from the class's own methods, **and** methods defined in **subclasses**

Access control

- For **encapsulation**, it is a good idea to restrict access to certain attributes and methods from outside the class
- **Private** members are only accessible from the class's **own** methods
- **Protected** members are accessible from the class's own methods, **and** methods defined in **subclasses**
- **Public** members are accessible from **outside** the class

Access control

Access control

- The **public interface** of an object is how it interacts with other objects and the rest of the program

Access control

- The **public interface** of an object is how it interacts with other objects and the rest of the program
- The **protected interface** of an object is what allows subclasses to change the way the base class behaves

Access control

- The **public interface** of an object is how it interacts with other objects and the rest of the program
- The **protected interface** of an object is what allows subclasses to change the way the base class behaves
- The **private members** of an object are implementation details, hidden from the outside world

Pedantic detail

Pedantic detail

- Access control is merely a **convention** in Python

Pedantic detail

- Access control is merely a **convention** in Python
  - The interpreter **won't stop** you from accessing protected members from outside the class or its subclasses, but PyCharm will **warn** you

Pedantic detail

- Access control is merely a **convention** in Python
  - The interpreter **won't stop** you from accessing protected members from outside the class or its subclasses, but PyCharm will **warn** you
  - Private members are "name mangled", but you can still access them if you know how

Pedantic detail

- Access control is merely a **convention** in Python
  - The interpreter **won't stop** you from accessing protected members from outside the class or its subclasses, but PyCharm will **warn** you
  - Private members are "name mangled", but you can still access them if you know how
- Almost all other languages enforce access control with compile-time errors

Summary

Summary

- OOP models three main types of real-world relationships

Summary

- OOP models three main types of real-world relationships
  - Is-a $\rightarrow$ instantiation

Summary

- OOP models three main types of real-world relationships
  - Is-a $\rightarrow$ instantiation
  - Has-a $\rightarrow$ composition

Summary

- OOP models three main types of real-world relationships
  - Is-a $\rightarrow$ instantiation
  - Has-a $\rightarrow$ composition
  - Is-a-type-of $\rightarrow$ inheritance

Summary

- OOP models three main types of real-world relationships
  - Is-a $\rightarrow$ instantiation
  - Has-a $\rightarrow$ composition
  - Is-a-type-of $\rightarrow$ inheritance
- Inheritance allows polymorphism: different classes with the same public interface

Summary

- OOP models three main types of real-world relationships
  - Is-a $\rightarrow$ instantiation
  - Has-a $\rightarrow$ composition
  - Is-a-type-of $\rightarrow$ inheritance
- Inheritance allows polymorphism: different classes with the same public interface
- Access control is an important tool in designing reusable, encapsulated objects