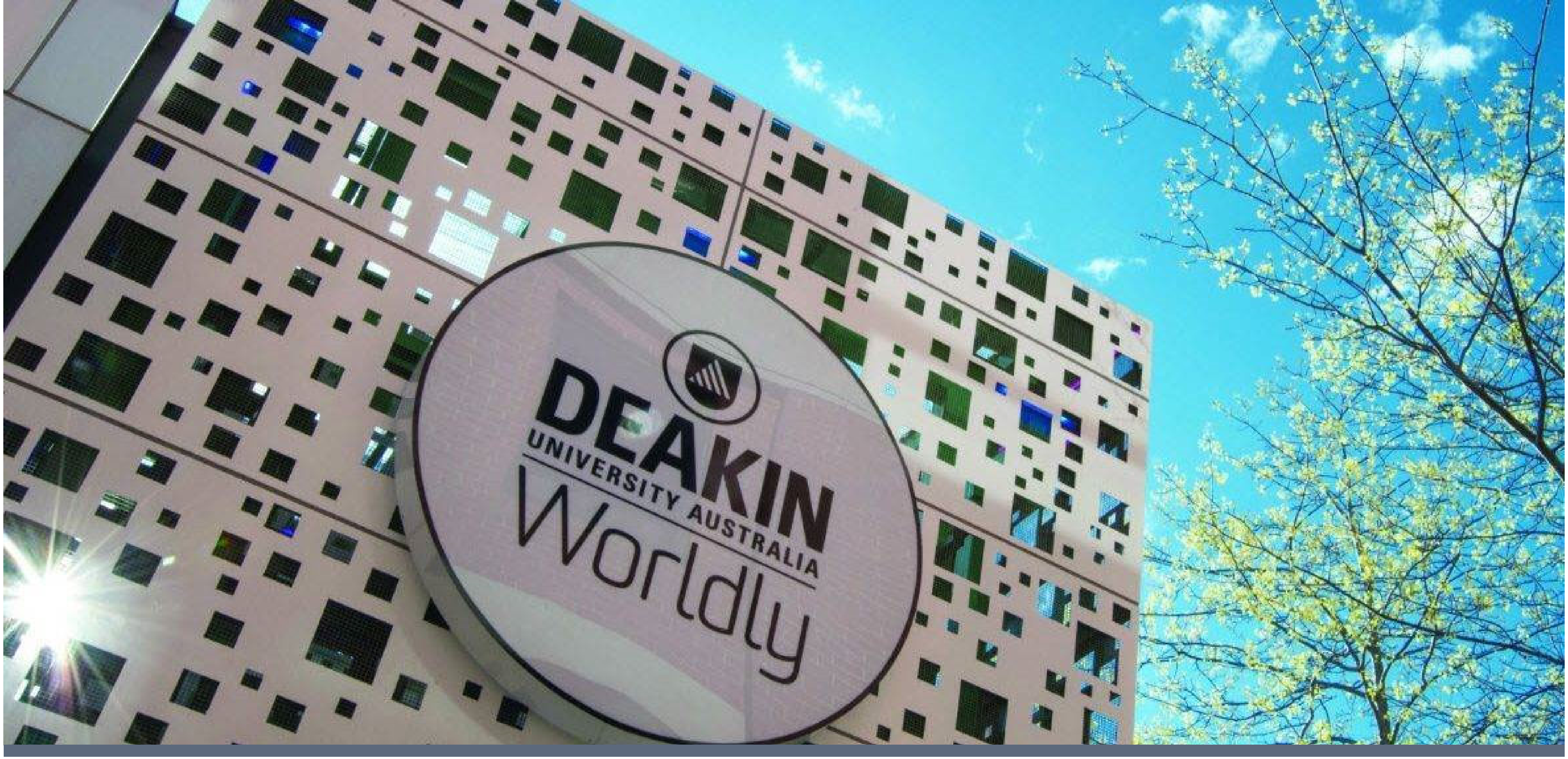

Lecture 06. Polymorphism. Abstract Classes and Interfaces

SIT232 Object-Oriented Development

Appetizer: Sergey and Andrew as objects

class **CourseDirector** extends **TeachingStaff**

- name: Andrew Cain
- position: Associate Head of School
- school: School of IT
- contact: andrew.cain@deakin.edu.au
- additional role: course director

- teach()
- set_exam()
- mark_exam()
- write_curriculum()

class **Lecturer** extends **TeachingStaff**

- name: Sergey Polyakovskiy
- position: Lecturer in Computer Science

  school : School of IT
- contact: sergey.polyakovskiy@deakin.edu.au

- teach()
- set_exam()
- mark_exam()
- write_unit_guide()

Course Director and Lecturer: Same but different

- Course directors and lecturers are similar in many ways (they represent teaching staff), but different in many others as well.
- These objects may exhibit absolutely different behaviour, e.g.
  - write_curriculum(…)
  - write_unit_guide(…)
- We want to refer to all staff members with general duties when we do not care about specific responsibilities.
- And only refer to the specific responsibilities (behaviour) when we need to.

OOP: The four major concepts

For a programming language to be considered as an object-oriented programming language, the four major concepts are expected:

- Abstraction
- Encapsulation
- Inheritance
- **Polymorphism**

Polymorphism: The idea

Polymorphism allows two or more objects of different types to respond to the same request

```
class SavingsAccount : Account
{
    ...
    public override string GetStatement()
    {
             ...
    }
    ...
}

class CreditCard : Account
{
    ...
    public override string GetStatement()
    {
             ...
    }
    ...
}

class TermDeposit : Account
{
    ...
    public override string GetStatement()
    {
             ...
    }
    ...
}
```

Polymorphism: The Idea

Polymorphism provides the ability to operate on and manipulate different objects in a uniform way

```
Account[] accounts = new Account[3];
accounts[0] = new SavingsAccount(...);
accounts[1] = new CreditCard(...);
accounts[2] = new TermDeposit(...);
...
Console.WriteLine(accounts[0].GetStatement());
Console.WriteLine(accounts[1].GetStatement());
Console.WriteLine(accounts[2].GetStatement());
```

Polymorphism: Syntax

// Base class:

```
[access_modifier] class base_class_name
{
    [access_modifier] virtual return_type method_name([parameter[, ...]])
    {
        method_body
    }
    ...
}
```

Derived class:

```
[access_modifier] class derived_class_name : base_class_name
{
    [access_modifier] override return_type method_name([parameter[, ...]])
    {
        method_body
    }
    ...
}
```

Polymorphism: How it works

- Binding – how the computer knows/learns the memory address of a particular method
  - **Static binding** – the compiler determines the types of objects and thus the address
  - **Dynamic binding** – the type of object is determined at run-time (required for polymorphism)
- **virtual** tells the compiler that it should get the implementation of the method from the object instance when the program is running.
- Note that if we make something virtual in the base class, it is automatically virtual when declared in the children.

Polymorphism: Examples of use

- Polymorphism allows a function or data structure to work correctly if passed an object of type T, but also work correctly when passed an object of type S that is a subclass of T.
- We can write a function that takes a parameter of class TeachingStaff, but that will accept (and work correctly), for CourseDirector and Lecturer.
- When we carry out operations on the subclasses, we may achieve different behaviour.

Polymorphism: Upcasting and Downcasting

Upcasting is the act of casting a reference of a subclass to one of its superclass type.

For example, let

`CallPhone(Person per)`

be a method, then

`CallPhone(st)` for `Student st`

must also work because Student class object is upcasted to Person class object automatically.

Polymorphism: Upcasting and Downcasting

Downcasting is the act of casting a reference of a base class to one of its derived classes.

It is possible to test the data type of a class (check for downcasting) using the **‘is’** operator, e.g.,

```
foreach(Account acc in accounts)
    if (acc is CreditCard)
        Console.WriteLine("Credit card found!");
```

Polymorphism: Upcasting and Downcasting

It is possible to convert between reference types using either the **‘as’** operator or **casting**, e.g.

```
foreach(Account acc in accounts) {
    if(acc is CreditCard) {
            CreditCard cc = acc as CreditCard;
            cc.Payment(500);
    }
}

foreach(Account acc in accounts) {
    if(acc is CreditCard)
    {
            CreditCard cc = (CreditCard) acc;
            cc.Payment(500);
    }
}
```

Abstract Classes and Methods

- An **abstract class** is a way to generalize concepts from which more specific classes can be derived.
- They cannot be instantiated, but we can use them as base classes to implement some general concepts.
- Derived classes must implement all abstract methods, otherwise they remain abstract.
- Along with abstract classes, we can declare **abstract methods** to represent the cases when no sensible implementation can be given for a method.

Abstract Classes and Methods: Syntax

```
// Abstract class
[access_modifier] abstract class base_class_name
{
    [access_modifier] class_member
    ...
}
```

```
[access_modifier] class base_class_name
{
    // Abstract class:
    [access_modifier] abstract return_type method_name( [parameter[, ...]] );
    ...
}
```