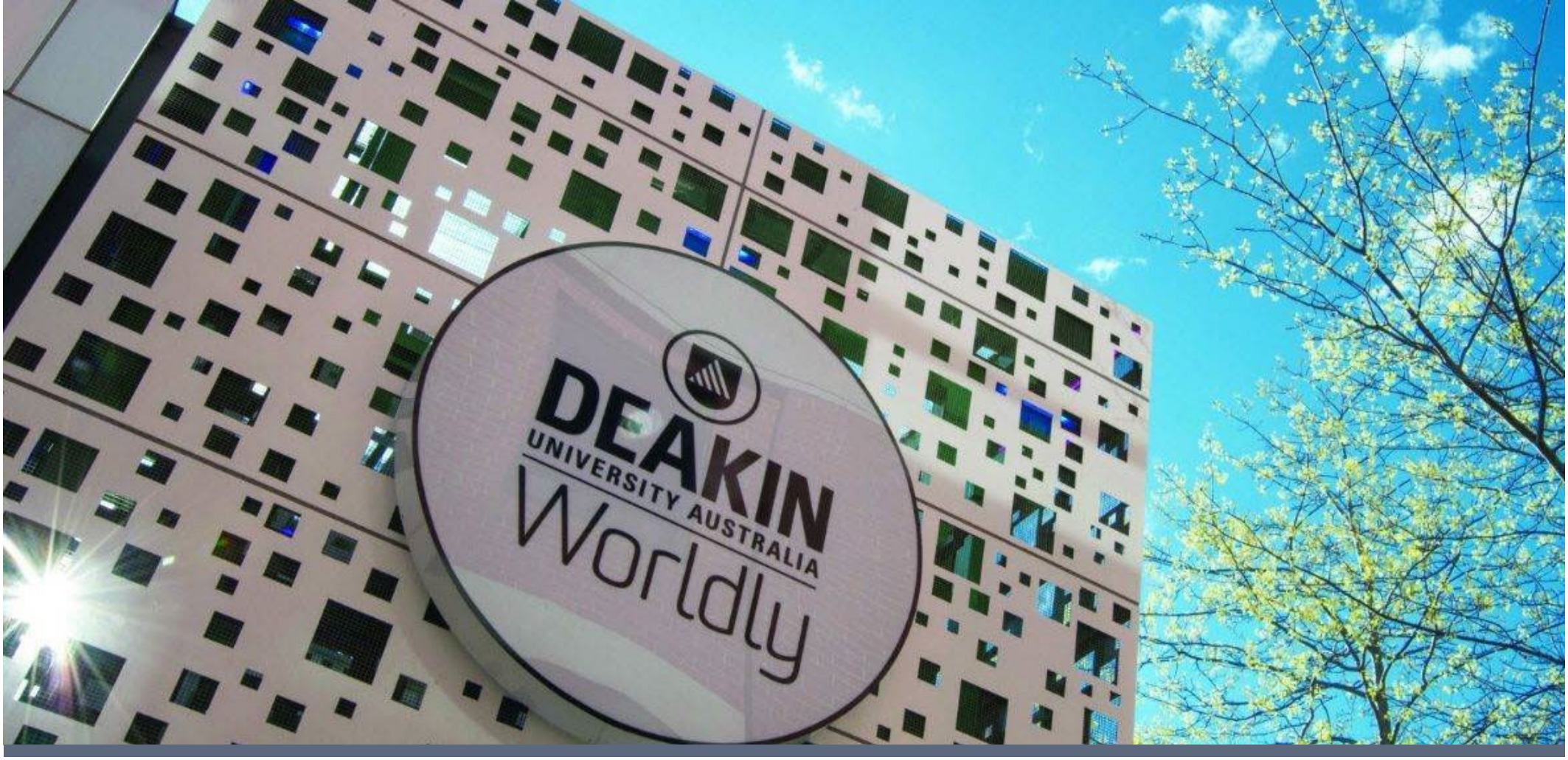

Lecture 02. Making Objects

SIT232 Object-Oriented Development

Class versus Object

- Real-world problems contain a number of entities (similar or different) that interact with each other in specific ways.
- We introduced the idea of objects to capture the entities in the problem domain.
  - Objects have **state**: attributes and values
  - Objects have **behaviour**: functionality
  - Objects interact with each other

Objects

- Objects **instantiate** entities from the problem domain.
- Objects **encapsulate** state: attributes and values.
- Across some objects (the two lecturers, the two pandas), the behaviour is almost the same.
- The behaviour may vary depending on state, but is very similar.
- A classification of these objects could group the lecturer and panda objects into separate **classes**.

Classes

- In Object Oriented Programming we create classes of objects by defining
  - the state that the objects should **encapsulate**, and
  - the behaviour that the objects should **exhibit**.
- An object is a run-time **instantiation** of a class.
- State information may be
  - a single copy shared by all objects of a class, or
  - created when the object is instantiated.

State Information: Unique versus Shared

Which of the following is the best option for the panda's state information?

1. Name, species, type and favourite food are shared.
2. Species and type are shared, each object separately records name and favourite food.
3. Each object separately records name, species, type and favourite food.

Encapsulation and Information Hiding

- Syntax to define a class:

```
[access_modifier] class class_name
{
        [access_modifier] class_member
        ...
}
```

- What is the role of the class member access_modifier?

Encapsulation and Information hiding

- **Encapsulation** is a separation of object/class data and methods that operate on that data.
- Usually the other objects that use an object do not have access to the object's data directly, but only through the methods.
- An object's data is hidden through a mechanism called information hiding.
- **Information hiding** is an ability to prevent certain aspects of a class from being accessible to other classes.

Information Hiding: Access Modifiers

In C#, information hiding can be achieved through class member access modifiers

- **public** - a public member is accessible from anywhere outside the class. If a variable is public, you can set/read its value directly.
- **private** (by default) - a private member is not accessible (not even for reads) from outside the class; only the class can access private members.
- **protected** - a protected member is accessible to methods of the class and sub-classes of the class (examined with Inheritance).

Information Hiding: Why we need it

Imagine we have a class called **BankAccount** that contains an attribute called **balance** representing all the money that the customer has; this attribute is public, and we have a function like:

```
class BankAccount
{
    public double balance;

    public void SetBalance(double newBalance)
    {
            if (newBalance >= 0) balance = newBalance;
    }
}
```

- In your groups, identify a scenario in which this can backfire
- Then, identify a fix

Information Hiding: Accessor and Mutator Methods

- **Accessor** and **mutator** methods
  - provide a public interface to private attributes
  - preserve encapsulation
  - define the abstraction/interface
- Accessor method
  - prefixed with ‘get’
  - used for reading data
- Mutator method
  - prefixed with ‘set’
  - used for storing/modifying data

Information Hiding: Accessor and Mutator Methods

Syntax for accessor and mutator methods:

```
// CLASS MEMBER (VARIABLE)
private string _name;

// ACCESSOR METHOD
public string GetName()
{
    return _name;
}

// MUTATOR METHOD
public void SetName (string value)
{
    _name = value;
}
```

Information Hiding: Property

- Properties are offered by many modern object-oriented languages
  - Provide a more intuitive interface, e.g.,

```
sales.Count = sales.Count + 1;
```

  - instead of

```
sales.SetCount(sales.GetCount() + 1);
```

- Have optional get and set blocks:
  - A read/write property – defines both get and set blocks;
  - A read-only property – defines only the get block; and
  - A write-only property – defines only the set block.

Information Hiding: Property

Syntax for property:

```
// CLASS MEMBER (VARIABLE)
private string _name;

public string Name
{
    // ACCESSOR PART
    get { return _name; }

    // MUTATOR PART
    set { _name = value; }
}
```

- omit ‘get’ part to get a write-only property
- omit ‘set’ part to get a read-only property

Information Hiding: Property

- The C# programming language also supports auto-implemented properties, e.g.,

```
public string Name { get; set; }
```

- The compiler automatically creates a hidden variable to store the data for the property
- Have optional get and set blocks:
  - A read/write property – indicate both ‘get;’ and ‘set;’
  - A read-only property – set ‘get;’ and make ‘private set;’
  - A write-only property – complete ‘set;’ and make ‘private get;’

Encapsulation and Information hiding

- Encapsulation and information hiding talk about how we design a single class, but what about a design with more than one class?
- These two fundamental principles imply:
  - Loose coupling
  - Cohesion

When you have to find and replace something like in a pile of climbing gear

It is much easier when the software behaves like this, isn't it?

Loose Coupling

- Coupling: refers to the degree of knowledge that one class needs to have of another in order for it to function.
- A system that is not loosely coupled will have classes in which methods depend on the implementation details of other classes.
- The less knowledge, the better:
  - classes could be replaced easily
  - buggy classes would not drastically influence other classes – if something crashes, other classes in the system should still be able to function

Loose Coupling

```
public class CartEntry {
      public float price { get; set; }
      public int quantity { get; set; }
}

public class CartContents {
      public CartEntry[] items { get; set; }
}

public class Order {
      private CartContents cart;
      private float salesTax;

      public float OrderTotal() {
          float cartTotal = 0;
          for (int i=0; i<cart.items.Length; i++) {
              cartTotal += cart.items[i].price * cart.items[i].quantity;
          }
          cartTotal += cartTotal * salesTax;
          return cartTotal;
      }
}
```

Does this implementation adhere to loose coupling?

Should we introduce discounts, how does this impact the program?

Cohesion

- Loose coupling is closely related to cohesion
- A cohesive class or module is one in which the functionalities of the class have much in common
- We want designs with high cohesion and loose coupling. This allows us to better reuse and maintain our code

Cohesion: Example

Where should **CountCustomersInBank** and **PrintStatement** be?

```
class BankAccount
{
        private double balance;
        public void SetBalance(double newBalance)
        {
                balance = newBalance;
        }
        public void CountCustomersInBank(){...}
        public void PrintStatement(){...}
}
```