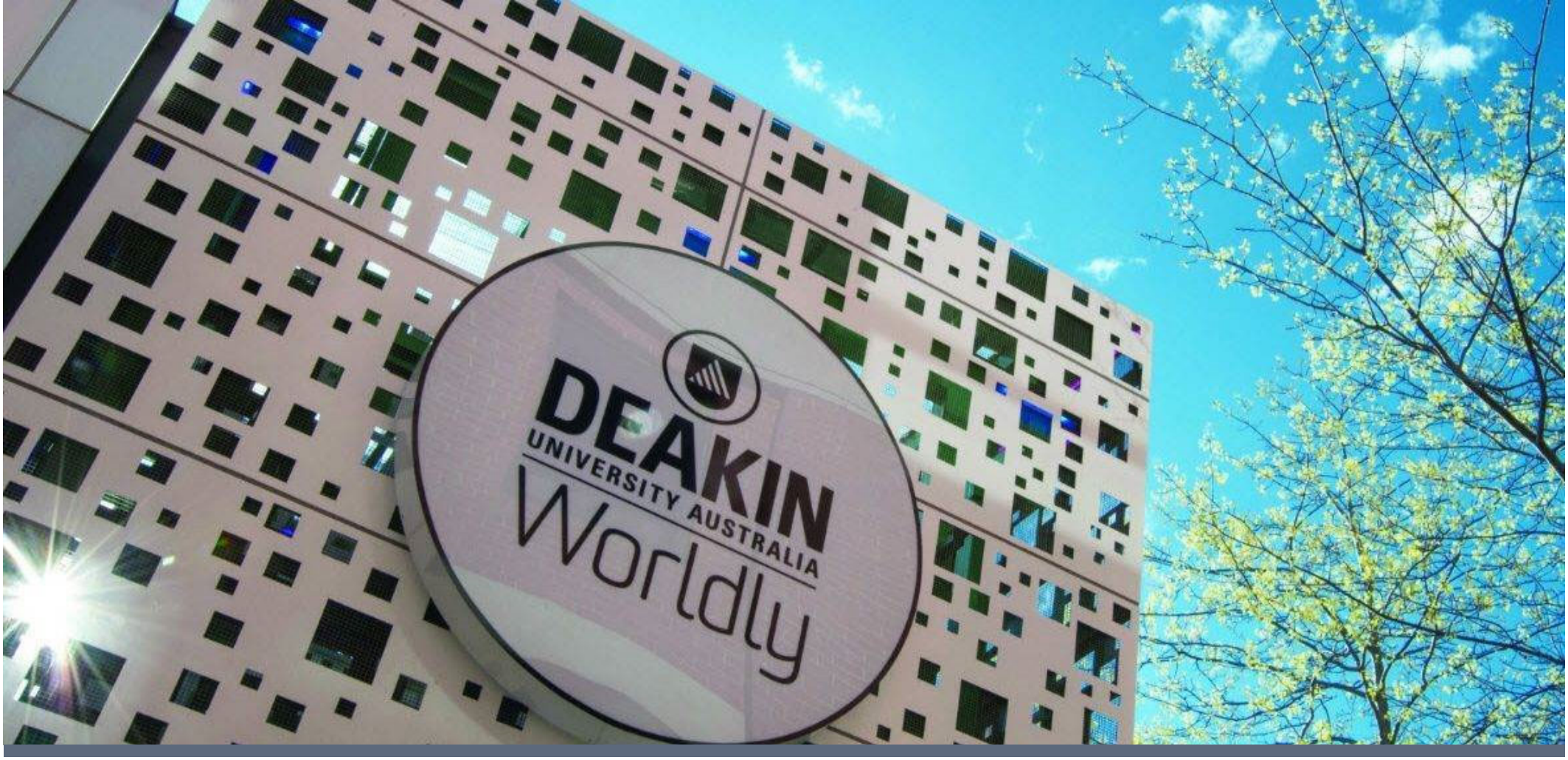


Lecture 1. Getting Started with Objects

SIT232 Object-Oriented Development

Appetizer

- Our study of programming to date was limited to procedural programming which deals with:
  - Variables
  - Decision structures
  - Looping structures
  - Methods/functions

- How do we translate this into developing complex applications?

Appetizer: The Melbourne Zoo Manager Problem

- The Melbourne Zoo is home to a large number of various animals, including but not limited to pandas, koalas, gorillas, lions, iguanas.
- Each animal has a different diet, sleeping pattern, requires different accommodations and can be seen by the public at different times.
- The Melbourne Zoo has one zoo manager. Every year, the zoo manager has to write a feeding, display and enclosure cleaning schedule that the zoo keepers use (they also have their own schedules btw.)
- He has hired you to write a program that does this!

The Zoo Manager Problem: Design and Coding

Complex programming problems are usually easier to solve when we can decompose them into smaller problems.

- Work on your design and identify sub-problems
  - How will solving the sub-problems lead to the solution?
  - Can the sub-problems be broken down into further sub-problems? Will this help?
- Decide on data structures to be used in sub-problems
- Turn the design into code
  - problem → program
  - sub-problems → functions
  - data structures → used in functions

The Zoo Manager Problem: Finding Sub Problems

The Melbourne Zoo has one zoo manager. Every year, the zoo manager has to write a feeding, … schedule that the zoo keepers use …

- Input:
  - all the animals in the zoo, dietary requirements, favourite foods, eating times; zoo keeper schedules
- Output:
  - for each animal or enclosure (which one? you decide!): feeding times and responsible keepers, food type and quantity
  - tabulated format
  - easily searchable

The Zoo Manager Problem: Finding Sub Problems

The Melbourne Zoo has one zoo manager. Every year, the zoo manager has to write a feeding, … schedule that the zoo keepers use …

- Data relevant to animals in the zoo:
  - species, sub-species and other classifications
  - name, id
  - dietary requirements; favorite food
  - eating plan: feeding times, quantities
  - display times
  - favorite toy
  - favorite zoo keeper
- Data relevant to zoo keepers:
  - name, id
  - working shift times

The Zoo Manager Problem: Challenge

- How would you represent the animals?
- Is it generally possible, for example, using arrays? Which of the following might be the best option?

1. Use a 2-dimensional array of strings. 2nd dimension contains the details of each animal.
2. Use an array of int or string or … for each kind of data you need to represent an animal. Matching positions in these arrays gives you a combination of characteristics describing the animal.
3. Do not use arrays, you cannot change their size but animals are born, die and move between zoos; the number of animals changes.

The Zoo Manager Problem: Objects to the rescue!

- The more programs are required to solve real-life problems with a large number of entities that interact in similar, but not quite the same ways, the more there is a need for a different paradigm.
- Objects are programming constructs that capture entities in the problem domain
- Object Oriented Programming is where we use objects to analyse, design and implement solutions to problems.

The Zoo Manager Problem: Objects' Examples

https://www.zoo.org.au/melbourne/animals/red-panda

- name: Bao Bao
- species: red panda
- class: mammalia
- favorite food: bamboo

- eat(...)
- treat(...)
- sleep(...)
- make_sound(...)

- name: Gao Gao
- species: red panda
- class : mammalia
- favorite food: japanese bamboo

- eat(...)
- treat(...)
- sleep(...)
- make_sound(...)

More of Objects' Examples

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

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

- name: Sergey Polyakovskiy
- position: Lecturer in Computer Science
  school : School of IT
- contact: sergey.polyakovskiy@deakin.edu.au

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

Objects: State and Behaviour

Each object has **state** and **behavior**

- **State:** collection of attributes and their values that define the object and, possibly, distinguish the object from other objects
- **Behavior:** what the object does: the functionality that it meets, how it interacts with other objects

Nouns are usually objects and state variables, **verbs** are behaviour

Objects are the fundamental building block, which can either be

- **Real**, such as a light switch, student, book, keyboard
- **Virtual**, such as an array, queue, textbox, avatar

Objects: State and Behaviour

Which of the following statements is true?

1. All nouns in the problem description must have their own objects.
2. All verbs in the problem description must have their own behaviours.
3. All nouns in the problem description must have their own objects or state information inside an object.
4. None of the above.

Objects: Objects Interaction

Objects interact with each other in various ways:

- **Aggregation**: objects can contain other objects (one or many)
  - a classroom contains many students and one lecturer
  - a cyclist has more than one bike
  - the Melbourne Zoo has several red pandas
- **Communication**: objects can tell other objects to do things, that is, perform their behaviour
  - a zoo keeper feeds a lion
  - a lion eats meat

Defining Classes

Object-oriented applications consist of a collection of cooperating objects

- An object is an instance of a class, the class acts as a **template** for the object/s
- A class definition consists of
  - Variables
  - Properties
  - Methods
- These represent the attributes and operations of an object's interface

Defining Classes: Syntax

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

access_modifier:

- public: accessible to all code
- private: accessible only within the same class
- and other types which we will consider later in the unit

Creating objects: Variables

There is often a need to store some data

- To reserve some memory, we must provide two pieces of information:
  - The **data type**, i.e., what sort of data is to be stored
  - A **name** to call this storage location
- This storage is known as a variable

```
data_type variable_name = variable_value;
```

Defining Classes: Class Members

- Creating instance variables (attributes):

  [access_modifier] data_type attribute_name [= value];

  These variables are part of an object (instance) rather than part of class (template)

- Creating methods to run operations :

```
[access_modifier] return_type method_name( [parameters])
{
        …
        method_body (sequence of operations and calls)
        …
}
```

Defining Classes: Constructor

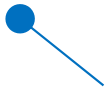
- Constructor methods are invoked immediately and automatically when an object is created
- Used for initialising an object

```
[ access_modifier ] class class_name
{
        [ access_modifier ] class_name( [parameters] )
        {
                ...
                method_body (sequence of operations and calls)
                ...
        }
        ...
}
```

Creating Objects

```
class_name variable_name;

variable_name = new class_name([parameters]);
```



this way we are calling the constructor and its parameter list must match one in the definition

or alternatively

```
class_name variable_name = new class_name([parameters]);
```

For example:

```
string line = new string('*', 50);
```

Using Objects

- Calling a method from another method in the same class:

```
method_name([parameter[, ...]]);
```

- Calling a method of a class from a method of another class using an object reference:

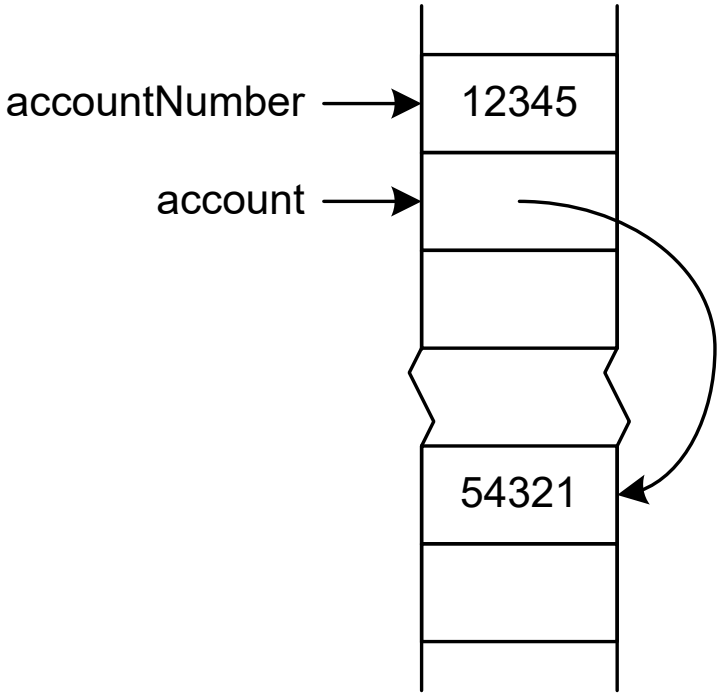
```
object_name.method_name([parameter[, ...]]);
```

e.g. database.AddStudent(55144789, "Helen“, 1985);

Value Types versus Reference Types

- (Simple) value types store data directly

```
int accountNumber = 12345;
```



- Reference types store a memory reference to the object/data

```
Account account = new Account(54321);
```

Static Members: Unique versus Shared

- In OOP 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 (thus be unique)
- Static (shared) members are associated with a class rather than objects and are declared using the 'static' keyword, e.g.

  static string species = "red panda";
- Address a static member of a class by the class name, e.g.

  Console.WriteLine("I am happy!"):

Static Members: Examples

- We can have a Panda class, that is instantiated by two objects (for Bao and Gao respectively)
  - Class and Order can be static (shared)
  - Name and Favourite Food must be unique to each object
- We can have a Lecturer class, that is instantiated by two objects (for Andrew and Sergey respectively)
  - School can be static (shared)
  - Name must be unique to each object

Summary

Procedural programming does not suffice to solve problems with a large number of interacting entities with varied states and behaviour

Objects are programming constructs that abstract entities from the problem domain

- **they have state:** attributes and their values
- **they have behaviour:** they can do things