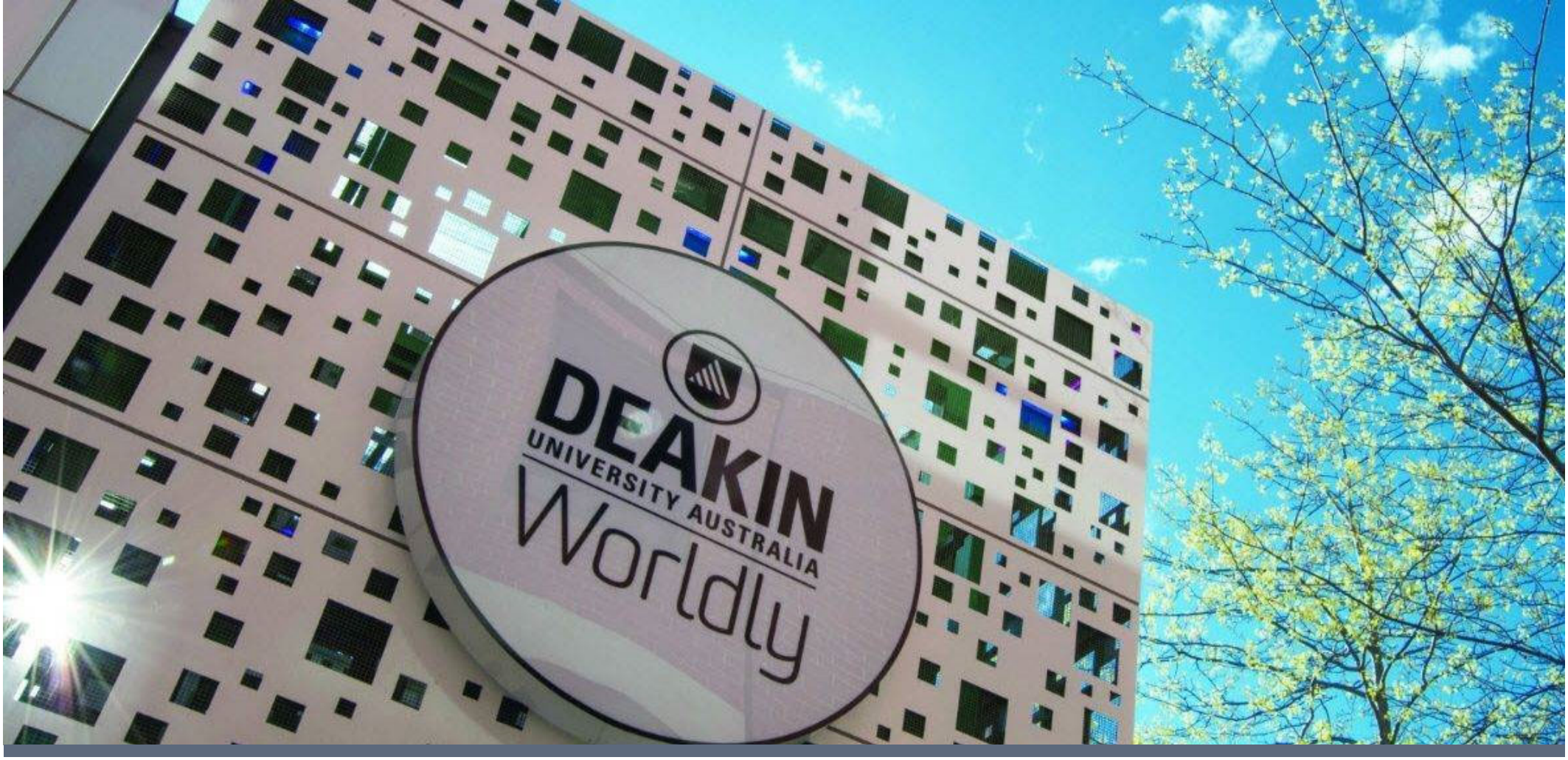

Lecture 10. Generics

SIT232 Object-Oriented Development

Generics: Appetizer

```
public class SampleClass {
   private double[] array;
   public int size { get; private set; }

   public DoubleStack(int capacity) {
       array = new double[capacity];
       this.size = 0;
   }

   public void Push(double val) {
       if (size < array.Length) array[size++] = val;
       else throw new InvalidOperationException("Out of capacity.");
   }

   public double Pop() {
       if (size > 0) return array[--size];
       else throw new InvalidOperationException("Is empty.");
   }
}
```

What this class is for?

Generics: Appetizer

```
public class Stack {
    private double[] array;
    public int size { get; private set; }

    public DoubleStack(int capacity) {
        array = new double[capacity];
        this.size = 0;
    }

    public void Push(double val) {
        if (size < array.Length) array[size++] = val;
        else throw new InvalidOperationException("Out of capacity.");
    }

    public double Pop() {
        if (size > 0) return array[--size];
        else throw new InvalidOperationException("Is empty.");
    }
}
```

How to write functionality that applies equally to different types?

Use of Object as a Global Superclass

- The **System.Object** class is the parent class of all the classes in C# by default.
- The System.Object class is beneficial if you want to refer any object whose type you don't know.
- A collection of objects could hold an int, a string, an Employee, a Student, and a Cat object at the same time.
- It is easy to add things, but tedious to retrieve/use them because what we retrieved is a System.Object, and not a Cat, int, or a string.
- Type checking and casting along with related coding techniques are necessary to use the items.

Use of Object as a Global Superclass

```
public class ObjectStack {
    private object[] array;
    public int size { get; private set; }

    public ObjectStack(int capacity) {
        array = new object[capacity];
        this.size = 0;
    }

    public void Push(object val) {
        if (size < array.Length) array[size++] = val;
        else throw new InvalidOperationException("Out of capacity.");
    }

    public object Pop() {
        if (size > 0) return array[--size];
        else throw new InvalidOperationException("Is empty.");
    }
}
```

This seems to be a stack implementation independent of the type of objects stored inside it. However…

Use of Object as a Global Superclass

- It is enough to get a program which compiles without a warning, but which is **not type safe**.
- We know that the use of casts in languages with strict type checking is particularly wrong in terms of design.
- Our solution is error prone and may give the programmer an illusory feeling that his program is type safe (since the language and the compiler impose type checking).
- We need a **type safe**, **generic** implementation of the data structure.

Generic Classes

- A **generic class** is not an actual class but a blueprint or template from which many concrete classes can be generated.
- It has a complete class definition except that it uses one or more placeholders (also called parameterized types) representing unspecified types.
- When it is time to use the generic class template to create an actual class, real types are substituted for the placeholders.

Generic Classes: Syntax

- The specification of generic type requires special syntax with angle brackets for designating parameters of generic type(s).

```
access_modifier class class_name < T, K, ...>
{
        ...
}
```

generic datatypes (placeholders)

- One may create many actual classes from a generic class by substituting different actual data types for the placeholders.

Example:

List<Student>, List<string>, List<Bitmap> are actual classes for the List<T> is a generic collection class with one “placeholder” type T.

Generic Classes: Application

```
public class GenericStack<ANY_TYPE> {
  private ANY_TYPE[] array;
  public int size { get; private set; }

  public GenericStack(int capacity) {
      array = new ANY_TYPE[capacity];
      this.size = 0;
  }

  public void Push(ANY_TYPE val) {
      if (size < array.Length) array[size++] = val;
      else throw new InvalidOperationException("Out of capacity.");
  }

  public ANY_TYPE Pop() {
      if (size > 0) return array[--size];
      else throw new InvalidOperationException("Is empty.");
        }
   }
```

Now the compiler is able to type check every use of the defined generic type and to find all violations of the type system.

Generic Classes: Using Constraints

- We can restrict type T by introducing constraints on it.
- The syntax of constraints declaration involves the where keyword, used as

```
access_modifier class class_name < T >
where T: base_class_name, interface_name
{
        …
}
```

- In this case, we restrict T to be derived from the specified base class and implement the specified interface.
- Objects of the resulting concrete classes may only hold object references of the designated types (and their subtypes, if any).
- Read more about constraints here.

Generic Methods: Syntax

A generic method is a method that is declared with type parameters.

access_modifier return_type method_mame<T>([parameters])

Example:

void SampleMethod<T>( ), or

T SampleMethod<T>( ), or

T SampleMethod<T>(T var1, T var2, int var3)

are all examples of generic methods with one “placeholder” type T.