SIT232 - OBJECT ORIENTED DEVELOPMENT

Session 5. Inheritance

Outline

- Session 05. Inheritance
  - Objectives
  - Reusability
  - Inheritance
  - Substitutability and Delegation
  - Bad Smells in Code

SESSION 5. INHERITANCE

Objectives

- At the end of this session you should:
  - Understand the **concept of reusability**, how it is *manifested in OO programs, and how to design for reuse*;
  - Understand **inheritance, how it is applied in object-oriented** programs, and what it should and shouldn't be used;
  - Be able to ***identify potential areas of poor design in your programs***.

Reusability

- **Reusability** – the ability to ***exploit/reuse previously developed code*** in the construction of new solutions
  - Seen in OO for **inheritance and delegation**
  - Not just restricted to these elements however
    - *Individual classes may be reused*, e.g., GUI components
    - *Entire hierarchies of classes can be reused*, e.g., database access mechanisms, printing mechanisms, etc.
    - *Elements of design can be reused*, e.g., interface/form design

Reusability

- **Advantages** of reusability include:
  - *A reduction in development time* – reused code is already complete and does not need to be developed from scratch;
  - *A reduction in testing time* – reused code has usually been tested thoroughly and can be relied upon in a new project;
  - *A reduction in time/effort to maintain existing code* – bug fixes, etc., to code can quickly and easily be carried to all projects in which it was reused; and
  - *Improved quality of code* – code that has been developed to be reusable will usually have been more carefully designed, coded, and tested.

Reusability

- **Disadvantages** of reusability include:
  - *Reusable code takes longer to develop than purpose-built code* (it is often said that it takes three times longer to develop reusable code);
  - *Reusable code is sometimes rarely*, if ever, reused, thus wasting the extra effort to develop it in the first place;
  - *Reusable code can take a long time to learn and/or adapt* for (or plug-in to) a new for project;
  - *Reused code can be restrictive*, i.e., if the code does not support a particular feature you may not be able to extend/easily extend that code to support that feature; and
  - *Bugs in reused code will be present in all projects* using that code.

Reusability

- Rules for improved reusability:
  - *Keep methods coherent* – methods should *perform either a single function or a group of closely related functions*
  - *Keep methods small* – smaller more *general functions are much easier to reuse* than larger more application specific functions
  - *Keep methods consistent* – the same *basic functions should have the same names, parameter li*sts, etc.
  - *Separate policy and implementation* – keep decision making (application specific) *separate from the mechanisms/logic to implement those decisions* (implementation)

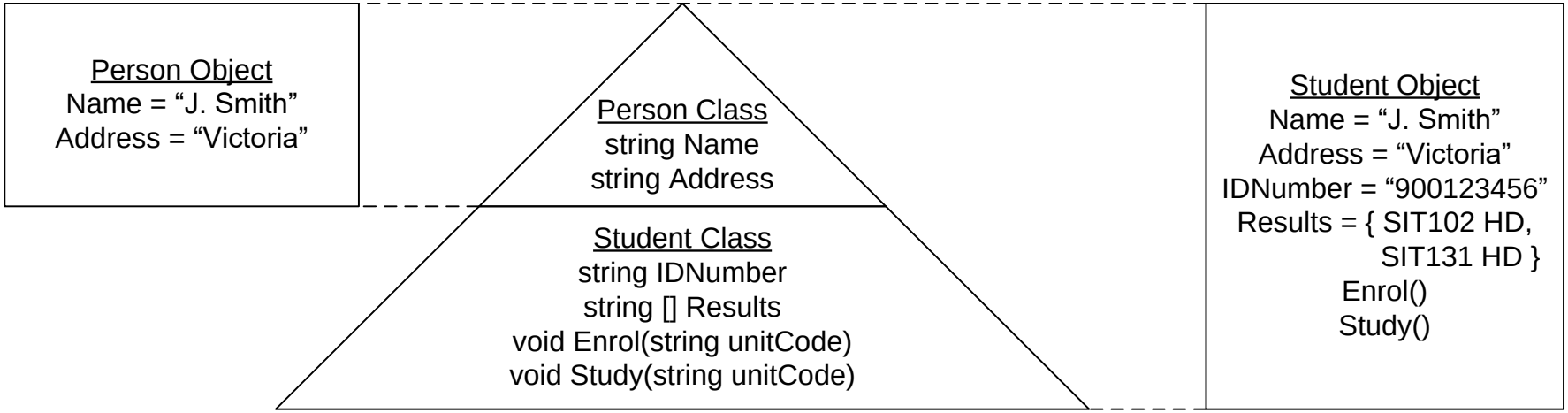
Reusability

- **Rules for improved reusability**:
  - *Provide uniform coverage* – provide *methods to handle all possible input possibilities, not just the expected/common ones*
  - *Broaden the method as much as possible* – accept more data types, *make fewer assumptions, provide meaningful results* for empty/extreme/invalid inputs, etc.
  - *Avoid global information* – *minimise the use of data read from outside* of a method
  - *Avoid methods with state* – try to *eliminate methods that change behaviour depending on their execution history*, usually by dividing the method

Inheritance

- Recall class relationships:
  - **Association**
  - **Aggregation**
  - **Composition**
  - **Inheritance**
- Unlike the others, *inheritance does not result in an object relationship*
  - Results in a new class containing the members of both the base class and the derived class

Inheritance



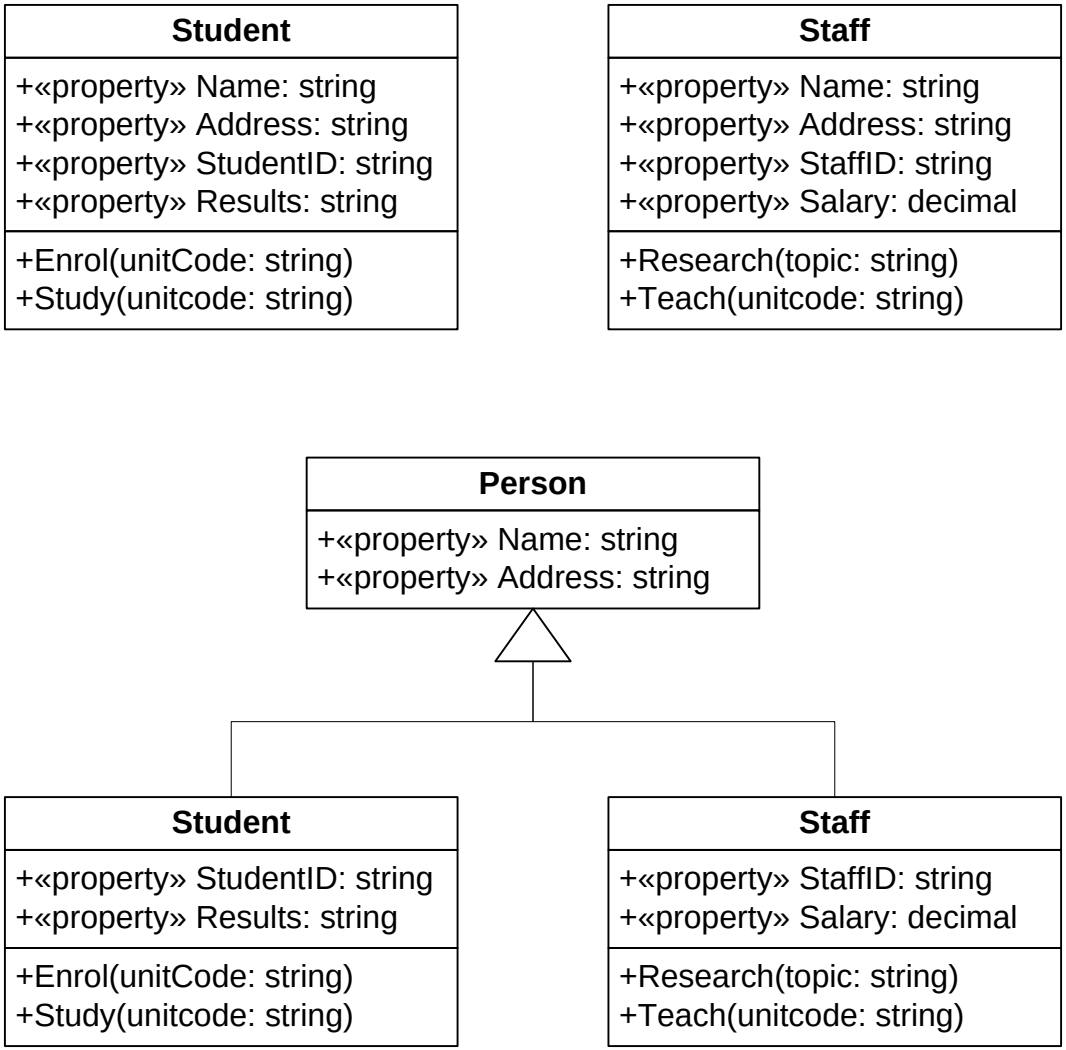
- Two phrases:
  - A Student **is a** Person
  - A Student **is a** kind of Person

Inheritance

- **Generalisation** –the *base class* is a *more general version of the derived class*

- **Specialisation** – the *derived class* is a *more specific version the base class*, or makes it more specific, by introducing new features (members)

Inheritance



Inheritance

- Syntax for inheritance in C#:

```
[access_modifier ]class derived_class_name : base_class_name
{
    [access_modifier ]class_member
    ...
}
```

Inheritance

```
/*************************************************************
** File: Account.cs
** Author/s: Justin Rough
** Description:
**     The Account base class used to demonstrate inheritance.
*************************************************************/
using System;

namespace BankAccounts
{
    class Account
    {
        protected Account(decimal balance)
        {
            _Balance = balance;
        }

        protected decimal _Balance;
        public decimal Balance
        {
            get { return _Balance; }
        }

        public override string ToString()
        {
            return string.Format("Balance: {0:c}", _Balance);
        }
    }
}
```

File: Account.cs

Inheritance

```
/*************************************************************
** File: SavingsAccount.cs
** Author/s: Justin Rough
** Description:
**     The SavingsAccount class that inherits from the Account
** class for demonstrating inheritance.
*************************************************************/
using System;

namespace BankAccounts
{
    class SavingsAccount : Account
    {
        private const decimal DEFAULT_BALANCE = 0.00M;

        public SavingsAccount() : this(DEFAULT_BALANCE)
        {
        }

        public SavingsAccount(decimal balance) : base(balance)
        {
        }

        public void Deposit(decimal amount)
        {
            _Balance += amount;
        }

        public bool Withdraw(decimal amount)
        {
            bool result = true;

            if (_Balance >= amount)
                _Balance -= amount;
            else
                result = false;

            return result;
        }
    }
}
```

Inheritance

```
/************************************************************
** File: CreditCard.cs
** Author/s: Justin Rough
** Description:
**      The CreditCard class that inherits from the Account
** class for demonstrating inheritance.
************************************************************/
using System;

namespace BankAccounts
{
    class CreditCard : Account
    {
        public CreditCard(decimal balance, decimal limit) : base(balance)
        {
            _Limit = limit;
        }

        private decimal _Limit;
        public decimal Limit
        {
            get { return _Limit; }
            set { _Limit = value; }
        }

        public void Payment(decimal amount)
        {
            _Balance += amount;
        }

        public bool Purchase(decimal amount)
        {
            bool result = true;

            if(-_Balance + amount <= _Limit)
                _Balance -= amount;
            else
                result = false;

            return result;
        }
```

File: CreditCard.cs (Part i)

Inheritance

Method hiding

```
        public new decimal Balance
        {
            get { return -_Balance; }
        }

        public override string ToString()
        {
            string result;

            if (_Balance < 0)
                result = string.Format("Balance: {0:c} owed", -_Balance);
            else if(_Balance == 0)
                result = base.ToString();
            else
                result = base.ToString() + " in credit";

            result += ", limit " + _Limit.ToString("c");

            return result;
        }
    }
}
```

File: CreditCard.cs (Part ii)

Inheritance

- Note that **`_Balance`** *in Account was protected*
  - Against: *Instance variables should always be private*
    - Allow their representation to be *changed without affecting derived classes*
  - For: Nobody will ever see it!
    - Often only the *original developer will ever see protected elements*
    - Why suffer the cost of the developer's salary to implement a never-to-be-used interface?
- **Generally**:
  - *Instance variables should be private where possible*
  - *Can you afford the cost / what is the ROI?*

Substitutability and Delegation

- **When to use inheritance?**
  - Consider the Liskov Substitution Principle
    - **An object of the base type should be substitutable by an object of the derived type**
    - *Is a Circle an Ellipse?*
      - Circle has constant radius
      - Ellipse has variable radius
      - Cannot be substituted, e.g.,
        - `anEllipse.SetSize(5, 10);` ✓
        - `aCircle.SetSize(5, 10);` ✘

Substitutability and Delegation

- There are **three solutions to this problem**:
  - *Eliminate any relationship* from between these classes;
  - *Introduce an additional class as a common base*; and
  - *Apply delegation*.

```
class Circle
{
        ...
        private Ellipse _Ellipse = new Ellipse();
        ...
        public double Area()
        {
                return _Ellipse.Area()
        }
        ...
}
```

Delegation Example

Bad Smells in Code

- There is *no way to clearly identify* what makes a good design for a class
  - There are *always alternatives*
  - *Several alternatives may be equally valid* and/or *equally good*
    - However, they *may not work well for other systems*!
  - *Several alternatives may be equally invalid* and/or *equally bad*
    - However, they *may still work well in other systems*!
- It is however possible to suggest ways to identify areas that are potentially bad design

Bad Smells in Code

- Here are some of the "bad smells in code" that are offered by Kent Beck and Martin Fowler in the textbook "***Refactoring: Improving the Design of Existing Code***"
  - Note that *a "bad smell" does not immediately identify bad design*, only potential bad design

Bad Smells in Code

- **Duplicated code** – the *same code appears in more than one place* in your program;
- **Long method** – a method *containing a large number of statements*;
- **Large class** – a class that has a *large number of members* can often lead to other problems such as duplicate code;
- **Long parameter list** – a method with *many parameters often has more parameters than required*, some information can be obtained from elsewhere such as an object;
- **Divergent change** – *modifying one class for two or more very different reasons*;

Bad Smells in Code

- **Shotgun surgery** – to make *one change requires modifications to many different classes*;
- **Feature envy** – a *method in one class constantly working with members of another class*;
- **Data clumps** – when *several pieces of data seen together in different areas of a program*;
- **Primitive obsession** – new OO programmers *hesitant to create very small classes, preferring instead to use simple types (primitive types)*;
- **Switch statements** – regular *modification to several switch statements may be replaceable with inheritance/polymorphism* (examined next week)

Bad Smells in Code

- **Parallel inheritance hierarchies** – *adding a subclass always needs another subclass* in a second hierarchy;
- **Lazy class** – a *class with very little functionality to perform*;
- **Speculative generality** – *classes made too general, providing functionality that will never be used* but was added “just in case”
- **Temporary field** – an *instance variable that is only occasionally used*

Summary

- Session 05. Inheritance
  - Objectives
  - Reusability
  - Inheritance
  - Substitutability and Delegation
  - Bad Smells in Code

Summary

- Training Videos:
  - C#: Inheritance