Object Oriented Software Development

5. Interfaces, polymorphism and inheritance

Types of interface

- The word “interface” has more than one meaning in programming
- User interface
  - The way in which the user interacts with the program
- Programming interface
  - The way in which software components in an application interact with each other
- We are looking at the latter here, and will look at the former later on

Interfaces example

- **InterfaceDemo project**

- **Employee.cs**
- **ITimeSheet.cs**
- **DatabaseTimeSheet.cs**
- **FileTimeSheet.cs**

The interface provided by a component

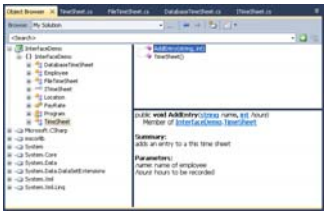
- In previous examples we saw an Employee class which **uses** a TimeSheet object
- Method parameter of type TimeSheet
- Within its RecordOvertime method an Employee calls the AddEntry method of the TimeSheet object
- Employee only needs to know that the TimeSheet class provides a method of that name, and the signature of the method

The interface provided by a component

- Employee does not need to know anything about how the AddEntry method works
- Details are hidden behind the class interface
- Details of how AddEntry works could be changed without affecting Employee
- If a programmer wants to use the TimeSheet class in another class, only needs to know the class interface

Documentation of interface

- Good documentation makes it easy for programmer to understand the interface of a class and how to use it
- XML comments help to make documentation useful



Programming to an interface

- Employee doesn't care about the details of a TimeSheet, just the interface
- Make this explicit by defining the interface as an item in its own right
- Convention (in .NET anyway) is to name the interface with an I - ITimeSheet
- Refer to the interface name, not the actual object type

Defining an interface

- Interface definition

```
public interface ITimeSheet
{
    /// <summary>
    /// adds an entry to a this time sheet
    /// </summary>
    /// <param name="name">name of employee</param>
    /// <param name="hours">hours to be recorded</param>
    void AddEntry(string name, int hours);

}
```

method is empty – no code block

- Use ITimeSheet as reference type

```
public void RecordOvertime(ITimeSheet timeSheet, int hours,
    PayRate payRate)
{
```


Defining properties in an interface

- A property is defined in an interface by specifying the property type, and empty get and/or set declarations

```
public interface ISampleInterface
{
    // Property declaration:
    string Name
    {
        get;
        set;
    }
}
```

Rules for interfaces

- Interfaces **can** contain:
  - Methods – declaration only
  - Properties – declaration only
  - Events
- Interfaces **can't** contain
  - Instance variables
  - Constants
  - Constructors
  - Static members
  - Access modifiers

Implementing an interface

- Interface doesn't actually do anything – need to create class which **implements** the interface
- Must provide implementation for all members (methods/properties) defined in the interface
- Class which implements ITimeSheet must (at least) provide a method called AddEntry with the signature defined in ITimeSheet
- Must provide code block for method

Implementing an interface

```
public class DatabaseTimeSheet : ITimeSheet
{
    private string database;

    /// <summary> ...
    public DatabaseTimeSheet(string database)
    {
        this.database = database;
    }

    /// <summary> ...
    public void AddEntry(string name, int hours)
    {
        // store information in database - incomplete
        Console.WriteLine("recorded that {0} worked {1} hours",
        Console.WriteLine("stored in database: {0}", database);
    }
}
```

"implements ITimeSheet"

class can have fields, constructors, etc

method has code block

Implementing multiple interfaces

- A class can implement more than one interface
- Class must provide implementations of **all** the methods declared in **all** the interfaces it implements

Interface as a contract

- A contract is an **agreement**
- By implementing an interface, a class **agrees** that it will provide the defined members
- This class can then be used by other classes with confidence because they know that it has agreed to provide the required members
- Employee can use any class which implements ITimeSheet in the knowledge that it will have AddEntry method

Reference and runtime types

- Method definition in Employee

```
public void RecordOvertime(ITimeSheet timeSheet, int hours,
    PayRate payRate)
{
    if (payRate == PayRate.Holiday)
    {
        timeSheet.AddEntry(name, hours * 3);
```

reference type is ITimeSheet

- Method call in Program

```
DatabaseTimeSheet dts = new DatabaseTimeSheet(@"mydb");
Employee empl = new Employee(1, "Michael", "michael",
    loc, "1234");
empl.RecordOvertime(dts, 8, PayRate.Weekend);
```

create DatabaseTimeSheet object at runtime

can call AddEntry because we know runtime type must have agreed to provide this method

pass DatabaseTimeSheet object as parameter at **runtime** – implements ITimeSheet

Interface polymorphism

- Polymorphism means “many forms”
- Reference to an interface type can point, as the program runs, to any object of a type which implements that interface
- Can pass any object which implements ITimeSheet as a parameter to RecordOvertime method in Employee
- Can declare a variable of type ITimeSheet and set it to an object of any type which implements ITimeSheet

Interface polymorphism

interface as reference type

alternative implementations of a time sheet

```
Employee emp1 = new Employee(1, "Michael", "michael",
    loc, "1234");
FileTimeSheet fts = new FileTimeSheet(@"c:\file.dat");
DatabaseTimeSheet dts = new DatabaseTimeSheet(@"mydb");
ITimeSheet ts = new FileTimeSheet(@"d:\file.dat");

emp1.RecordOvertime(fts, 5, PayRate.Weekend);
emp1.RecordOvertime(dts, 8, PayRate.Weekend);
emp1.RecordOvertime(ts, 10, PayRate.Day);
```

can pass any of these objects at runtime when parameter type is ITimeSheet

interface can be implemented by class **or** struct

```
public struct FileTimeSheet : ITimeSheet
{
```


Advantages of using interfaces

- Application code written based on the functionality guaranteed by interface rather than functionality of a particular class
- Reduces code dependencies
- Code standardisation
- Easier to write reusable code
- *Interfaces are used extensively in the .NET Framework library*

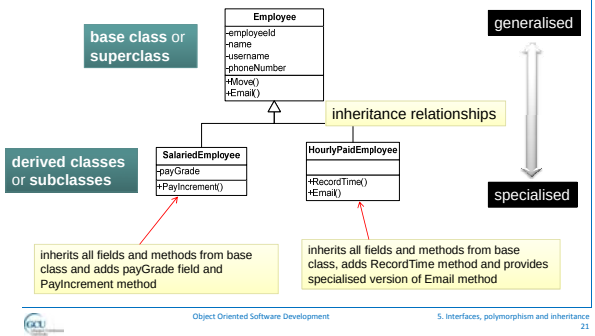
Inheritance

- Often have classes which have some common features
  - e.g. different kinds of employee – SalariedEmployee, HourlyPaidEmployee
- These are more specific versions of Employee
  - *"is-a"* relationship – SalariedEmployee is a type of Employee
  - They will share some common features
  - Each will require some specific features

Inheritance example

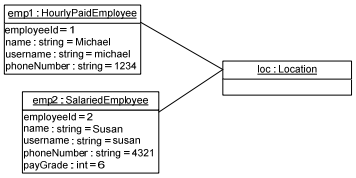
- **InheritanceDemo project**
- **Employee.cs**
- **SalariedEmployee.cs**
- **HourlyPaidEmployee.cs**

Inheritance hierarchy



Inheritance - object diagram

- Instance of subclass has all fields defined in subclass **and** all fields defined in superclass
- No actual instance of superclass at runtime



Inheritance hierarchies

- A class can have many subclasses
- A class can only have one superclass
  - No multiple inheritance (in C# anyway)
  - Compare this with implementing interfaces
- Subclass can itself have subclasses
- Subclass can:
  - Inherit members of superclass
  - Add new members
  - Override members of superclass

Advantages of using inheritance

- Reduced code duplication
- Increased code reuse
- Models real-world situations
- Polymorphism

- *Inheritance is used extensively in the .NET Framework library*

.NET base types

- All .NET types are derived from System.Object class
- All .NET value types are derived from System.ValueType class
- System.ValueType is a subclass of System.Object
- But, inheritance from structs is **not** allowed

Implementing inheritance

```
public class Employee
{
    //CONSTANTS
    protected const string EMAIL_SUFFIX = "@example.com";

    //INSTANCE VARIABLES
    protected int employeeId;
    protected string name;
    protected string username;
    protected Location currentLocation;
    protected string phoneNumber;


public class SalariedEmployee : Employee
{
    // CONSTANTS
    private const int maxGrade = 10;

    // INSTANCE VARIABLES
    private int payGrade;
```

protected – means variables are accessible from subclasses, but not from other classes

SalariedEmployee inherits all of these

"extends Employee"

also define public PayGrade property, other properties inherited

Subclass constructor

- Call base class constructor to do initialisation of inherited fields
  - Only need to write code for specific subclass initialisation

```
public SalariedEmployee(int employeeId, string name,
    string username, Location location, string phoneNumber,
    int payGrade) :
    base(employeeId, name, username,location, phoneNumber)
{
    if (payGrade >= 0 && payGrade <= maxGrade)
    {
        this.payGrade = payGrade;
    }
}
```

```
SalariedEmployee emp2 = new SalariedEmployee(2, "Susan", "susan",
    loc, "5678", 6);
```

Inherited methods

- SalariedEmployee and HourlyPaidEmployee both inherit Move method from Employee
- Can call Move method on instance of subclass

```
HourlyPaidEmployee emp1 = new HourlyPaidEmployee(1, "Michael",
    "michael", loc, "1234");
SalariedEmployee emp2 = new SalariedEmployee(2, "Susan", "susan",
    loc, "5678", 6);

emp1.Move(newloc);
emp2.Move(newloc);
```


Added methods

- HourlyPaidEmployee has additional method RecordTime, which is not in Employee
- SalariedEmployee does not have this method

```
HourlyPaidEmployee emp1 = new HourlyPaidEmployee(1, "Michael",
    "michael", loc, "1234");
SalariedEmployee emp2 = new SalariedEmployee(2, "Susan", "susan",
    loc, "5678", 6);

emp1.RecordTime(dts, 5, PayRate.Weekend);
emp2.RecordTime(dts, 5, PayRate.Weekend);
```

gives compiler error – emp2 is an instance of SalariedEmployee

Overriding a method

- HourlyPaidEmployee has a specialised version of the Email method
- Constructs email address in a way which is specific to this type of employee
- Need to override the Email method

Overriding a method

- Declare method as **virtual** in base class

```
public virtual string Email()
{
    return username + "@example.com";
}
```

- Declare method as **override** in subclass

```
public override string Email()
{
    return  username + "_h_" + "@example.com";
}
```

```
HourlyPaidEmployee emp1 = new HourlyPaidEmployee(1, "Michael",
    "michael", loc, "1234");
SalariedEmployee emp2 = new SalariedEmployee(2, "Susan", "susan",
    loc, "5678", 6);
string email1 = emp1.Email();
string email2 = emp2.Email();
```

runs method defined in HourlyPaidEmployee

runs method defined in Employee

Abstract classes

- It is likely in the example that every employee will be one or other of the specific kinds
- Can declare Employee as an **abstract class**

```
public abstract class Employee
{
```

- This means that no instances can be created
- Abstract classes are meant to be base classes which provide basic functionality to be inherited by concrete classes

Sealed classes

- If you don't want a class to be subclassed, you can declare it as a **sealed class**

```
public sealed class HourlyPaidEmployee : Employee
{
```

- Can also define methods as sealed when class is not sealed
- Subclasses cannot override sealed methods

Inheritance polymorphism

- Similar to interface polymorphism
- Reference to any type can point, as the program runs, to any object of a type which derives from that type
- Can pass an instance of any subclass of Employee to AddEmployee method in Department
- Can declare a variable of type Employee and set it to refer to an instance of any subclass of Employee

Inheritance polymorphism

base class as reference type

```
HourlyPaidEmployee empl = new HourlyPaidEmployee(1, "Michael",
    "michael", loc, "1234");
SalariedEmployee emp2 = new SalariedEmployee(2, "Susan", "susan",
    loc, "5678", 6);
Employee emp3 = new SalariedEmployee(3, "Ahmad", "ahmad",
    loc, "4321", 7);
Department dep = new Department();
dep.AddEmployee(emp1);
dep.AddEmployee(emp2);
dep.AddEmployee(emp3);
```

can pass any of these objects at runtime when parameter type is Employee

Polymorphism and methods

- Example:
  - **reference** type is Employee
  - **runtime** object type is HourlyPaidEmployee

```
Employee emp = new HourlyPaidEmployee(1, "Michael",
    "michael", loc, "1234");
string email = emp.Email();
emp.RecordTime(dts, 5, PayRate.Weekend);
```

runs method defined in **runtime** type*

error – method not defined in **reference** type

- You cannot call a method which is not defined in the object's reference type

*the behaviour is different if you use the word **new** instead of **override** in the method definition – look up the C# reference and try to find out why

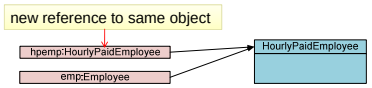
Casting

- Solution – **cast** reference to the runtime type

```
Employee emp = new HourlyPaidEmployee(1, "Michael",
    "michael", loc, "1234");
emp.RecordTime(dts, 5, PayRate.Weekend);
HourlyPaidEmployee hpemp = (HourlyPaidEmployee)emp;
hpemp.RecordTime(dts, 5, PayRate.Weekend);
```

cast operator

- Object type doesn't change, only reference type



- Cast must be valid

Upcasting and downcasting

- Upcast (widening)
  - Cast subclass to base class
  - Automatic
- Downcast (narrowing)
  - Cast base class to subclass
  - Need explicit cast

```
Employee emp = new Employee();
HourlyPaidEmployee hpemp = new HourlyPaidEmployee();
Department dept = new Department();

emp = hpemp; //widening conversion
hpemp = (HourlyPaidEmployee)emp; //narrowing conversion
hpemp = emp; // illegal conversion
dept = emp; // illegal conversion
```


Invalid casts

- In previous slide, compiler reported errors for invalid casts
- Can have code in which each operation is within the rules, but the result at runtime is invalid
- This code compiles, but causes a runtime error – why?

```
Object o = new HourlyPaidEmployee();
SalariedEmployee semp = (SalariedEmployee)o;
```

Prefix and As casting

- Previous example was prefix casting
  - reliable casting
  - reports error (throws exception) if cast is invalid
- Can use As casting
  - fast casting
  - null reference if cast is invalid – no error reported

```
Object o = new HourlyPaidEmployee();
SalariedEmployee semp = o as SalariedEmployee;
```

- Specific to .NET!

When not to use inheritance

- Inheritance is one of the cornerstones of object oriented programming
- But...
- It is not always the right solution for modelling a real-world situation
- General rule: **“don’t use inheritance to model roles”**

Roles

- Could think of the employees example as follows:

```
HourlyPaidEmployee emp1 = new HourlyPaidEmployee(1, "Michael",
    "michael", loc, "1234");
SalariedEmployee emp2 = new SalariedEmployee(2, "Susan", "susan",
    loc, "5678", 6);
```

- Michael is an employee whose role in the company is an hourly paid worker
- Susan is an employee whose role is a salaried worker

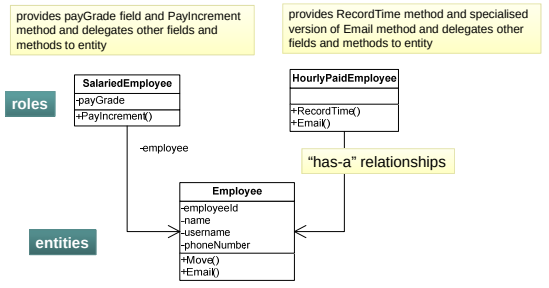
Changing roles

- What if Susan's role is to change, and she is to become an hourly paid worker?
- We would have to create a whole new HourlyPaidEmployee object to represent her, and remove the existing SalariedEmployee object
- No representation of the fact that this is still the same person
- Inheritance is a **static** structure

A better way - composition

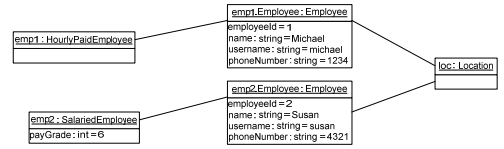
- Instead, we could have one object to represent the **entity** (the employee in this case) and one to represent the **role**
- Combine these to represent Susan
- Can change the combination at runtime
- Combine Susan's Employee object with a different role object to change role
- Employee could have more than one role
- Composition is a **dynamic** structure

Composition



Composition - object diagram

- Instances of role and entity exist at runtime



Composition compared to inheritance

- Advantages
  - dynamic structure
  - models changing roles
  - models multiple roles
- Disadvantages
  - more objects in memory
  - code can be harder to understand
  - Polymorphism doesn't work (?)
- Which is better in this example?

Composition example

- **CompositionDemo**
- **HourlyPaidEmployee.cs**
- **SalariedEmployee.cs**
- **Employee.cs**

Implementing composition

- Employee class unchanged, except methods are not declared virtual
- Role classes have instance variable of type Employee

```
class SalariedEmployee
{
    // CONSTANTS
    private const int maxGrade = 10;

    // INSTANCE VARIABLES
    private Employee employee;
    private int payGrade;
```


Delegation of properties/methods

- Pass a request for EmployeeId property to *employee* object

```
public int EmployeeId
{
    get { return employee.EmployeeId; }
}
```

- Pass a call to Move on to *employee* object

```
public void Move(Location newLocation)
{
    employee.Move(newLocation);
}
```


Constructors

- Create Employee object then assign it to role object

```
public SalariedEmployee(Employee employee, int payGrade)
{
    this.employee = employee;
    if (payGrade >= 0 && payGrade <= maxGrade)
    {
        this.payGrade = payGrade;
    }
}
```

```
Employee emp2 = new Employee(2, "Susan", "susan", loc, "5678");
SalariedEmployee semp2 = new SalariedEmployee(emp2, 6);
```

Changing role

- Create new role object and assign existing Employee object to it
- Could assign another Employee to existing role or set it to null

```
HourlyPaidEmployee hpemp2 = new HourlyPaidEmployee(emp2);
semp2 = null;
```


Polymorphism

- How does the example code achieve the advantages of polymorphism even though we have abandoned inheritance?

Further reading

- The following links lead to useful articles on interfaces in C#:
  - http://www.c-sharpcorner.com/UploadFile/rmcochran/csharp_interrfaces03052006095933AM/csharp_interrfaces.aspx?ArticleID=cd6a6952-530a-4250-a6d7-54717ef3b345
  - http://www.csharp-station.com/Tutorials/Lesson13.aspx

What's next?

- We will go on to look at structures for holding collections of similar objects, including arrays and the .NET collection types
- Some of the concepts introduced in this chapter will be very useful when looking at collections