

Session 6. Polymorphism

In this session we examine polymorphism and the mechanisms related to polymorphism. Importantly, polymorphism was historically seen as a key feature of object-oriented applications, however more recently it has been understood that polymorphism also significantly contributes to the complexity of applications and must be applied carefully. We also examine some of the mechanisms for controlling inheritance and polymorphism, including abstract methods and classes, and sealed methods and classes, which are important to ensure any design assumptions in your developed software is adhered to in future modifications and corrections performed either by yourself or other developers.

Session Objectives

In this session, you will learn:

- The concept of polymorphism, how it is achieved, and how to apply it in solving problems;
- How abstract and sealed methods and classes provide the ability to express the desirable semantics of inheritance and how to use them;
- How interfaces provide an alternative to inheritance and how they are applied; and
- How to define extra functionality for the C# operators and such functionality can improve code readability.

Unit Learning Outcomes

- *ULO 1. Apply object-oriented concepts including abstraction, encapsulation, inheritance, and polymorphism.*

We continue developing our knowledge of object-oriented concepts, focusing on polymorphism.

- *ULO 2. Solve programming problems using object-oriented techniques and the C# programming language.*

We continue developing our understanding of object-oriented techniques and how they are applied in the C# programming language in this session.

- *ULO 3. Modify an existing object-oriented application to satisfy new functional requirements.*

We will not be directly addressing this ULO, however understanding of the C# programming language will contribute to the skills required for this outcome later.

- *ULO 4. Construct object-oriented designs and express them using standard UML notation.*

We will not be directly addressing this ULO, however polymorphism is often applied in object-oriented applications and will inform your knowledge of object-oriented design.

Required Reading

Additional reading is required from the textbook Deitel, P., and Deitel, H., "Visual C# 2012: How to Program", Fifth Edition, Pearson, 2014, as follows:

Required Reading: Chapter 12

This chapter introduces the concept of polymorphism and related concepts such as abstract classes and interfaces. Polymorphism is a critical feature of object-oriented programming so take care to understand this content.

Required Tasks

The following tasks are mandatory for this week:

- Task 6.1. Polymorphism (Part 1)
- Task 6.2. Polymorphism (Part 2)
- Task 6.3. Polymorphism Challenge

6.1. Introduction

In this session we continue our study of inheritance by examining polymorphism, which provides the ability to perform operations on objects without knowing their type. This is followed by an examination of abstract methods and classes, useful for hierarchies when there is no clear implementation for a method in the base class, and sealed methods and classes, where it is desirable to prevent any further inheritance or overriding of methods. Interfaces are then considered, which provide an alternative to inheritance where only the interface (signatures of public members) is inherited without any implementation being inherited. Operator overloading then provides us with the ability to define new functionality for the operators provided by the C# programming language. Finally, we examine several techniques of refactoring as a follow-up to our coverage of bad smells in code in the previous session.

6.2. Polymorphism

In Session 5, we examined the concept of inheritance and how it allows us to reuse parts of the programs that we write by exploiting the commonality that exists between related classes. We considered the example of different types of bank accounts, including a savings account, credit card account, and term deposit account. In this example, we exploited the fact that each account has a balance, such as the balance of money available (savings account), money owed (credit card), or money invested (term deposit).

However, there are other aspects of bank accounts that we can consider, such as the ability to prepare/produce a statement. For a statement, the savings account and credit card accounts would involve similar output, listing a collection of increases (deposits and payments) and decreases (withdrawals and purchases) to the bank account's balance. However, for a term deposit, which would not have such a list of transactions, the output may be quite different. This is where **polymorphism** becomes useful.

For a computer language to be considered an object-oriented programming language, there are four concepts that are expected: abstraction, encapsulation, inheritance, and polymorphism. The term polymorphism is derived from the Greek words *polus* (many) and *morphe* (forms). There are two common views of polymorphism:

- i. Allows two or more objects of different types to respond to the same request; and
- ii. The ability to operate on and manipulate different objects in a uniform way.

These two views are in fact referring to the same concept, only from different perspectives. If we return to the example of preparing bank account statements, we can have a `SavingsAccount` object, a `CreditCard` object, and a `TermDeposit` object to output a different statement to a method call `GetStatement()`, i.e., they each have their own implementation:

```
class SavingsAccount : Account
{
    ...
    public override string GetStatement()
    {
        ...
    }
    ...
}

class CreditCard : Account
{
    ...
    public override string GetStatement()
    {
        ...
    }
    ...
}

class TermDeposit : Account
{
    ...
    public override string GetStatement()
    {
        ...
    }
    ...
}
```

With these methods defined, we can then operate on these objects in a uniform way, i.e., the same way, as follows:

```
Account[] accounts = new Account[3];
accounts[0] = new SavingsAccount(...);
accounts[1] = new CreditCard(...);
accounts[2] = new TermDeposit(...);
...
Console.WriteLine(accounts[0].GetStatement());
Console.WriteLine(accounts[1].GetStatement());
Console.WriteLine(accounts[2].GetStatement());
```

This example shows several new concepts required for polymorphism. First, notice the use of the `Account` class for the type of the array, even though we are creating `SavingsAccount`, `CreditCard`, and `TermDeposit` objects. Initially this may seem strange, because the type of object created on the right hand side of the assignment operator does not strictly match the type of the variable on the left hand side. However, recall the use of the phrases ‘is a’ and ‘is a kind of’ for testing inheritance. Applying these phrases, we can see that a `SavingsAccount` **is an** `Account`, thus the statement now appears valid. In terms of polymorphism, the key concepts here are (i) the different types of objects accessed polymorphically must be related via inheritance, and (ii) the objects are accessed by using a reference of the base class type.

Another question that may occur to you at this time, is that given we are referring to the objects now as `Account` objects, surely the `GetStatement()` method defined in the `Account` class would be invoked instead? Although this may appear a natural conclusion, it is imperative at this point to understand the difference between **static binding** and **dynamic binding**.

The word binding refers to how the computer knows where to find the code for a method body. Every method that we write in a program is converted from the high level programming language (C#) into machine code instructions that are directly executed by the computer¹. This code will be located in the memory image of your process. Each location in memory is numbered, known as a memory address, and when the method is invoked the CPU begins executing the code at the specified memory address.

Static binding refers to the ability of this memory address to be determined by the compiler, and the address is then stored in the executable file, simplifying the invocation of a method. Dynamic binding however is where the type of an object is not known at compile time, i.e., when polymorphism is used, and must be determined at run-time. Only once the object type is known can the method’s memory address be determined and invoked.

The use of polymorphism in C# requires that we examine closely two keywords: `virtual` and `override`. The `virtual` keyword is used to indicate to the compiler that the method being defined can change in derived classes, for which we use the `override` keyword to achieve this task. When a `virtual` method is invoked, the system will first check through the inheritance hierarchy to see if there are any overrides for this method, i.e., dynamic binding is used. The method defined using the `override` keyword in the most derived class (for this object type) is then called. Polymorphic methods are defined in the C# programming language using the following syntax. In the base class, a method is defined using the keyword `virtual`, as follows:

```
[access_modifier] class base class name
{
```

¹ In C# this is quite a lengthy process, with the C# compiler converting your program code into Common Intermediate Language (CIL) which is run on the Common Language Runtime (CLR). The CLR then converts your code to machine code (directly executed by the CPU) at the last minute using a Just In Time (JIT) compiler.

```

...
[access_modifier] virtual return_type method_name([parameter[, ...]])
{
    method_body
}
...
}

```

The method can then be redefined using the `override` keyword in the derived class, as follows:

```

[access_modifier] class derived_class_name[ : base_class_name]
{
    ...
    [access_modifier] override return_type method_name([parameter[, ...]])
    {
        method_body
    }
    ...
}

```

Note that the *return_type*, *method_name*, and *parameter* list must match precisely between the base class and derived class. Note also that the `override` keyword can only be used for methods that have been defined in a base class as `virtual`.

Importantly the difference between polymorphic methods, using the `virtual` and `override` keywords, and method hiding (see Section 5.3), using the `new` keyword, may not be entirely clear at this point. Both of these approaches involve defining a method in a derived class that replaces the same method defined in the base class. The key difference is that polymorphic methods exploit dynamic binding whereas method hiding only uses static binding.

A clear example of the difference between polymorphic methods and method hiding is shown in the code extracts shown in Figure 6.1. In this example are two classes: `BaseClass` and `DerivedClass`, which inherits from `BaseClass`. The `BaseClass` class then defines two methods: `SimpleMethod()` and `PolymorphicMethod()`, a polymorphic method defined with the `virtual` keyword. In the `DerivedClass` class, `SimpleMethod()` is replaced using method hiding and `PolymorphicMethod()` is replaced by overriding.

```

class BaseClass
{
    public void SimpleMethod()
    {
        Console.WriteLine("This is BaseClass.SimpleMethod()");
    }

    public virtual void PolymorphicMethod()
    {
        Console.WriteLine("This is BaseClass.PolymorphicMethod()");
    }
}

class DerivedClass : BaseClass

```

```
{
    public new void SimpleMethod()
    {
        Console.WriteLine("This is DerivedClass.SimpleMethod()");
    }

    public override void PolymorphicMethod()
    {
        Console.WriteLine("This is DerivedClass.PolymorphicMethod()");
    }
}

static void Main(string[] args)
{
    BaseClass bcObject = new BaseClass();
    DerivedClass dcObject = new DerivedClass();
    BaseClass baseReference = dcObject;

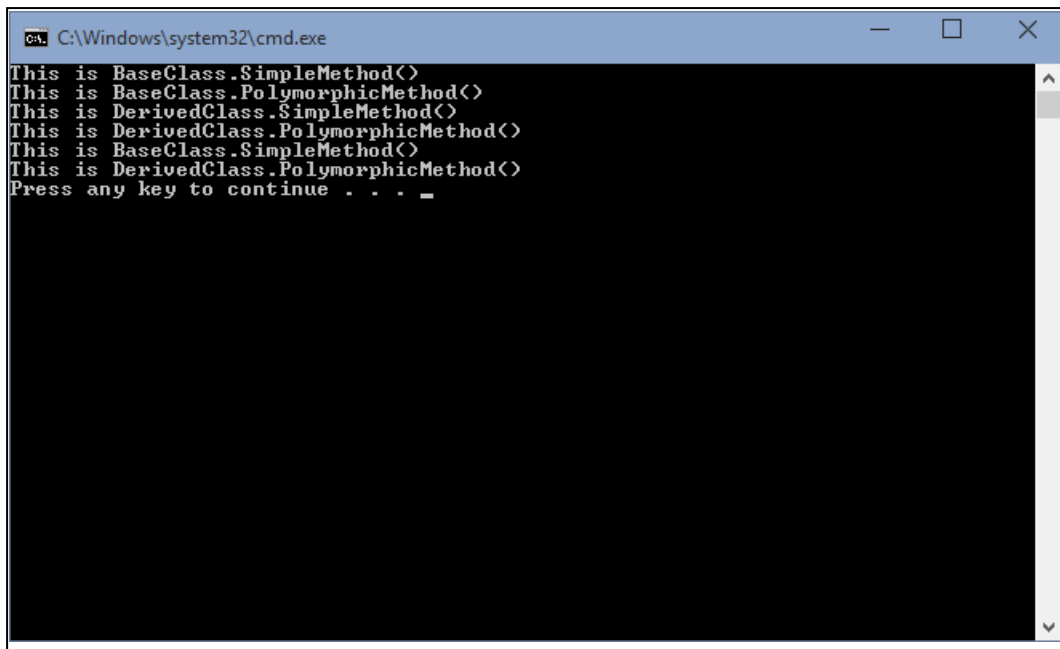
    bcObject.SimpleMethod();
    bcObject.PolymorphicMethod();

    dcObject.SimpleMethod();
    dcObject.PolymorphicMethod();

    baseReference.SimpleMethod();
    baseReference.PolymorphicMethod();
}
```

Figure 6.1: Comparison of Method Hiding and Polymorphic Methods

The `Main()` method then creates an object of each type, and a second reference to a `BaseClass` object to which the address of the `DerivedClass` object is stored (for polymorphism). The `SimpleMethod()` and `PolymorphicMethod()` methods are then invoked using each of the three object references. The output of this program is shown in Figure 6.2. If you compare the code in the program with the output, you will notice that the first two method calls result in the expected output, providing the output of the two `BaseClass` methods for a `BaseClass` object. Similarly, the second two method calls result in the expected output, providing the output of the two `DerivedClass` methods for a `DerivedClass` object.



```

C:\Windows\system32\cmd.exe
This is BaseClass.SimpleMethod()
This is BaseClass.PolymorphicMethod()
This is DerivedClass.SimpleMethod()
This is DerivedClass.PolymorphicMethod()
This is BaseClass.SimpleMethod()
This is DerivedClass.PolymorphicMethod()
Press any key to continue . . . _

```

Figure 6.2: Output of Method Hiding vs Polymorphic Methods

The final two method calls however may initially appear quite strange. Although we are still talking about the `DerivedClass` object, the output appears incorrect as the first call, to the `SimpleMethod()` method, is producing the text defined in the `BaseClass` class, not the text defined in the `DerivedClass` class.

Reconsider the code extracts appearing in Figure 6.1 again. Using the ‘is a’ phrase, a `DerivedClass` object is a `BaseClass` object. This means that a variable of type `BaseClass` can refer to either objects of `BaseClass` type or `DerivedClass` type. At compile time, there is no way to tell exactly which type the variable is referencing. For method hiding, which uses static binding, the compiler must determine the address of the method, for which it can only assume it is the data type of the variable used for the reference (the `BaseClass` method in this case). However for polymorphic methods, the compiler generates the necessary code for the object to be determined at run-time, in turn invoking the method defined for the object’s actual data type rather than the method defined for the variable’s data type.

Importantly, we have already been using polymorphism for several weeks: the `ToString()` method is a polymorphic method, hence the need for the `override` keyword when implementing `ToString()` for our classes. This means that if you are using an inheritance hierarchy that you have defined polymorphically, you can invoke the `ToString()` method for the objects and the correct implementation will be selected. Importantly, as indicated above, classes must be related to each other in order to use polymorphism. The `ToString()` method is no exception and the base class in this case is the `object` class (found in Microsoft.Net as the `System.Object` class). Every single type that exists in a C# application, including simple and custom types, inherit from this class. This is why even empty classes have the `ToString()` method defined, as discussed in Section 4.7.2.

When dealing with inheritance hierarchies, and particularly with polymorphism, there is often a need to either test the data type of an object, e.g., is this `Account` object a `SavingsAccount` object?, and/or to change the type of reference used to refer to an object, e.g., for a `SavingsAccount` object change its reference from the `Account` type back to a `SavingsAccount` reference. To test the type of an object, the C# programming language provides the `is` operator, e.g.,

```
foreach (Account acc in accounts)
    if (acc is CreditCard)
        Console.WriteLine("Credit card found!");
```

To change the type of reference used, C# provides either the `as` operator, e.g.,

```
foreach (Account acc in accounts)
{
    if (acc is CreditCard)
    {
        CreditCard cc = acc as CreditCard;
        cc.Payment(500.00M);
    }
}
```

or provides casting (discussed in Section 3.6), e.g.,

```
foreach (Account acc in accounts)
{
    if (acc is CreditCard)
    {
        CreditCard cc = (CreditCard)acc;
        cc.Payment(500.00M);
    }
}
```

The difference between using the `as` operator and using casting can be found in what happens if an invalid conversion is requested. For example:

```
Account acc = new SavingsAccount(...);
CreditCard cc = acc as CreditCard;
```

or

```
Account acc = new SavingsAccount(...);
CreditCard cc = (CreditCard)acc;
```

In this example, an attempt is made to change the type of reference to a `SavingsAccount` object (referred to by an `Account` reference) to a `CreditCard` reference. Given that `SavingsAccount` and `CreditCard` are siblings, inheriting from the same parent class, this is an invalid operation. When the system detects the error, the result of using the `as` operator is that a value of `null` is assigned to the `cc` reference above. However, if casting were used, an exception is thrown instead (we examine exceptions in Section 9.3).

6.3. Abstract Methods and Classes

In the last section we introduced the concepts of polymorphism using an example where a `GetStatement()` method could be added to the bank account example from Session 5. Although this presents a straightforward example where each different type of `Account` object could produce a different statement, consider carefully what you would put into the `Account` class' implementation of the `GetStatement()` method? Given that the `Account` class does not represent any actual type of bank account, it is not clear exactly what kind of statement should be produced.

For a clearer example, consider the creation of an inheritance hierarchy to represent different geometric shapes: circle, rectangle, and triangle. We can define a simple inheritance hierarchy for these shapes, as shown in Figure 6.3. In considering the shapes hierarchy, we could implement a number of polymorphic methods: `GetArea()` to calculate the area of a shape, `GetPerimeter()` to calculate the perimeter of a shape, and `Draw()` to draw a graphic representation of a shape for GUI to name a few. However, what would the implementation of these methods be for the base `Shape` class? Without knowing what type of shape object is being considered, there is no sensible implementation for either of these methods. For this reason, object-oriented languages provide **abstract methods**.

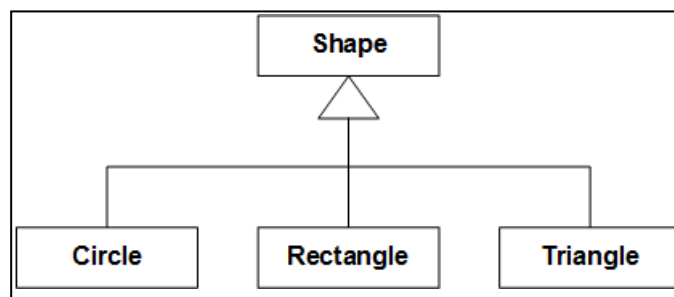


Figure 6.3: Simple Inheritance Hierarchy for Shapes

An abstract method is a method that does not have an implementation. Where an abstract method is used, the implementation of the method is deferred and must be implemented by derived classes. This also means that the class is incomplete (contains a method without an implementation), and is referred to as an **abstract class** or **abstract base class**. A method that is not abstract can then be referred to as a **concrete method**, similarly a class that is not abstract can be referred to as a **concrete class**. The terms concrete method and concrete class are not particularly common however.

In C#, we declare abstract classes with the addition of the `abstract` keyword, i.e., an abstract class is defined using the following syntax:

```

[access_modifier] abstract class derived class name [ : base class name ]
{
    ...
    [access_modifier] class_member
    ...
}
  
```

```
}
```

Similarly, an abstract method is defined using the following syntax:

```
[access_modifier] abstract return_type method_name([parameter[, ...]]);
```

Methods that are defined as `abstract` are implicitly `virtual`, requiring use of the `override` keyword when implementing them in derived classes.

Example code for the above shapes example is provided in Figure 6.4. An `abstract` base class has been defined, `Shape`, which has a single `abstract` method `GetArea()`. Three classes are then defined for the different shapes: `Circle` (containing a radius), `Rectangle` (length and width), and `Triangle` (three side lengths). Each of these classes has a custom constructor for initialising their attributes and implement the `GetArea()` method using appropriate formulae. The `Main()` method creates an array of `Shape` objects and stores one `Circle`, one `Rectangle`, and one `Triangle` in the array. Finally, a `foreach` loop is used to access the shapes polymorphically, displaying the type of shape (using the default `ToString()` implementation) and the area (calculated using the `GetArea()` method).

```
abstract class Shape
{
    public abstract double GetArea();
}

class Circle : Shape
{
    public Circle(double radius)
    {
        _Radius = radius;
    }

    private double _Radius;
    public double Radius
    {
        get { return _Radius; }
        set { _Radius = value; }
    }

    public override double GetArea()
    {
        return Math.PI * _Radius * _Radius;
    }
}

static void Main(string[] args)
{
    Shape[] someShapes = new Shape[3];

    someShapes[0] = new Circle(5.0);
    someShapes[1] = new Rectangle(5.0, 6.0);
    someShapes[2] = new Triangle(5.0, 5.0, 6.0);

    foreach (Shape s in someShapes)
        Console.WriteLine("{0,-16} area is {1:f3}", s, s.GetArea());
}
```

Figure 6.4: Example of Applying Abstract Methods and Classes for Shapes

There are a few rules for using `abstract` classes and methods in C#, as follows:

- Any `class` containing an `abstract` method must also be defined as `abstract`, however an `abstract class` does not require an `abstract` method;
- Any derived class that does not implement all inherited `abstract` methods must also be defined as `abstract`;
- It is not possible to create an instance of an `abstract` class, however it is possible to have a reference to an object of an `abstract` class (for polymorphism); and
- It is possible to `override` a `virtual` method in a base class with an `abstract` method, making the base `virtual` method unavailable and forcing a new implementation to be provided in derived classes.

Task 6.1. Polymorphism (Part 1)

Objective: Polymorphism is a relatively simple mechanism to understand, however many programmers new to object-oriented programming find it difficult to comprehend exactly how to exploit polymorphism. We begin with a simple extension to an existing example.

You are required to extend the Shapes example presented in Figure 6.3 and Figure 6.4 to also display the perimeter for each of the shapes using a polymorphic method `GetPerimeter()`. Note that the perimeters can be calculated using the following formulae:

- Circle:
- Rectangle:
- Triangle:

Note: The full code for the Shapes example can be found in CloudDeakin in the Weekly Resources folder for this week (in the Code Examples ZIP file).

6.4. Sealed Methods and Classes

The concepts of a **sealed method** and **sealed class** are in some respects the opposite of abstract methods and abstract classes. Where an abstract method/class must be inherited before it can be used (implementing any abstract methods in the process), a sealed method is one which cannot be overridden and a sealed class is one which cannot be inherited from. Sealed methods and sealed classes are written in C# with the use of the `sealed` keyword. Only one rule exists for the use of sealed methods and classes, which is that the method/class cannot also be abstract. The syntax for sealed methods is as follows:

```
[access_modifier] sealed return_type method_name([parameter[, ...]])  
{  
    method_body  
}
```

```
}
```

Similarly, sealed classes are as follows:

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

6.5. Interfaces

Interfaces are similar to inheritance in a number of respects. Consider another term that can be used to describe inheritance: **implementation inheritance**. The idea of this term is to illustrate that the use of inheritance results in the implementation of one class, the base class, is being duplicated in another, the derived class. Interfaces can similarly be described as **interface inheritance**, where only the interface (signature of members) is duplicated from the interface to the class that implements that interface. Interfaces are also often described as being equivalent to an abstract class consisting only of abstract methods, i.e., there is no implementation to be duplicated (consisting of attributes, property accessor bodies, and method bodies). Simply put, an **interface** defines the signature/s for one or more members that must be present in any class that chooses to implement the interface. Unlike inheritance, classes can choose to implement any number of interfaces.

The syntax for defining an interface in C# is as follows:

```
[access_modifier] interface interface_name [ : base_interface_name [, ...]]
{
    interface_member
    ...
}
```

In the above syntax, the *access_modifier* represents the access modifier that applies to all members defined in the interface, which defaults to `internal` the same as for classes (see Section 2.3). If more than access modifier is required, i.e., to define members with different access modifiers, separate interfaces must be defined. The *interface_name* is a symbolic name with the same requirements as variables, classes, methods, etc., however the name is usually preceded by the letter 'I', e.g., `IDisposable` which was examined in Section 4.5 for defining `Dispose()` methods. Finally, the *base_interface_name* represents one or more interfaces which are inherited by this interface, i.e., it is possible to build an interface using one or more other interfaces as a base.

Interface members can consist of either properties and/or methods, but not instance variables. The syntax for properties is as follows:

```
type_name { [get;] [set;] }
```

and for methods:

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

Implementing an interface for a class uses the same syntax as for inheritance, as follows:

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

Defining the actual class members is identical to what we have already seen, except the data types and names in the property and method signatures must match. Importantly, note that the implementation of the properties and methods for an interface do not require the `override` keyword (there is no `virtual/abstract` member from a base class to override!).

A class can inherit from one class and implement any number of interfaces by placing the base class name and the interfaces in a comma separated list, e.g.,

```
class DerivedClass : BaseClass, ISomeInterface, IDisposable
{
    ...
}
```

Finally, polymorphism can be achieved using interfaces using the same syntax as for inheritance. The only difference is that instead of using the base class type for an array, `List`, etc., the interface type is used instead.

Task 6.2. Polymorphism (Part 2)

Objective: As indicated above, interfaces can be used in place of an abstract method which only contains abstract methods. In this task you will explore interfaces by modifying a problem that uses such an abstract class.

Your solution to Task 6.1 currently includes an abstract base class (`Shape`) and three derived classes (`Circle`, `Rectangle`, and `Triangle`). In this task, you are required to modify your solution to replace the `Shape` class with an interface named `IShape`, which should be defined to include only the two methods `GetArea()` and `GetPerimeter()`. Upon completion of this task, the `Shape` class will no longer exist.

Task 6.3. Polymorphism Challenge

Objective: Now that you have considered polymorphism both using inheritance and interfaces, this task requires you to develop your own solution and demonstrate your understanding.

You are required to develop an application that exploits polymorphism for a vehicle rental agency. The rental agency offers two types of rentals: by-day, and by-kilometre.

By-day rentals are charged at the flat rate of \$100.00/day. By-kilometre rentals are charged an initial \$30.00 fee to cover the fuel loaded in the vehicle, a rate of \$0.50/kilometre, and 10.0% added to the total cost (initial cost plus per km rate) to cover ware and tare. Your program must satisfy the following requirements:

- Five classes must be used: `RentalAgency` (contains `Main()`), `Person` (represents a client who rents a vehicle), `Rental` (base class), `RentalByDay`, and `RentalByKm`.
- An association should be maintained from `Person` to `Rental`, such that up to five rentals can be recorded for a single person.

The following `Main()` method must be used:

```
static void Main()
{
    Person jane = new Person("Bloggs", "Jane");
    Person joe = new Person("Bloggs", "Joe");
    Person peter = new Person("Piper", "Peter");
    Person penny = new Person("Piper", "Penny");

    new RentalByDay(jane, 5);
    new RentalByDay(jane, 2);
    jane.PrintRentals();

    new RentalByDay(joe, 8);
    new RentalByKm(joe, 15);
    joe.PrintRentals();

    new RentalByDay(peter, 1);
    new RentalByKm(peter, 85);
    peter.PrintRentals();

    new RentalByDay(penny, 5);
    new RentalByKm(penny, 42);
    penny.PrintRentals();

    Console.WriteLine("Quote for {0}", new RentalByDay(null, 10));
    Console.WriteLine("Quote for {0}", new RentalByKm(null, 10));
}
```

The following output should result (or similar):

```
Bloggs, Jane
  Rental for 5 days - $500.00
  Rental for 2 days - $200.00
```

```
Bloggs, Joe
  Rental for 8 days - $800.00
  Rental for 15km - $41.25
```

```
Piper, Peter
  Rental for 1 days - $100.00
  Rental for 85km - $79.75
```

```
Piper, Penny
  Rental for 5 days - $500.00
  Rental for 42km - $56.10
```

Quote for Rental for 10 days - \$1,000.00
Quote for Rental for 10km - \$38.50

Hint: You will need to consider carefully how to define the constructors for RentalByDay and RentalByKm. Also, if you are struggling to determine how to model the above problem, make sure you are applying incremental development, e.g., comment out all of the code in Main except for the last two WriteLine() methods, and get those working. Now uncomment the creation of the Person objects and then the RentalByDay object creations, and get those working. Finally uncomment the RentalByKm object creations, and get those working.

6.6. Operator Overloading

In Section 3.4.2 we examined method overloading, where an additional definition can be provided for an existing method. Similarly, **operator overloading** provides the programmer with the ability to define the functionality for an operator where one or both operands are a user-defined type. The use of operator overloading can improve the readability of code, e.g.,

```
invoice1.Add(invoice2);
```

or

```
Invoice.Add(invoice1, invoice2);
```

can become

```
invoice1 + invoice2
```

The following operators can be overloaded in C#²:

- Unary operators: + - ! ~ ++ --
- Binary operators: + - * / % & | ^ << >> == != > < >= <=

The operators that cannot be overloaded are:

- Binary operators: = && ||
- Ternary operator: ?:

The syntax for overloading a unary operator is:

```
public static data_type1 operator operator(data_type2 rhs)
{
    ...
}
```

Similarly, the syntax for overloading a binary operator is:

```
public static data_type1 operator operator(data_type2 lhs, data_type3 rhs)
```

² Note that which operators can be overridden changes between programming languages.

```
{  
    ...  
}
```

In the above syntax, up to three data types are indicated, which either be the same or different data types. However, at least one of the data types in the parameter list must be the class in which the operator is defined. Note that the parameters will carry the value of the operands to the operator overload method, with *lhs* indicating the parameter for an operand appearing to the left of the operator (left-hand-side), and *rhs* indicating an operand appearing on the right of the operator (right-hand-side). The actual parameters can have any name following the variable naming conventions of the C# language however. Also note that given none of the operators that can be overloaded involves assignment, you should never modify the parameters, and return a new object for the result.

6.7. Refactoring

Recall from Section 5.5 the concept of refactoring: the process of modifying the design of an existing program to either improve the existing design, or to carry out necessary design changes to facilitate more functionality. In Section 5.5 we considered the bad smells in code that provide hints on identifying areas of a design that need improvement. In this section, we look closer at the actual refactorings that can be done. Fowler et al³. provides a comprehensive listing of possible refactorings⁴, too many to cover here. Instead we cover a number of the more common refactorings. Note that many refactorings are actually provided in pairs, where one refactoring is the inverse/opposite of another:

- Extract method – where a method is too long and/or a fragment of code requires a comment to explain it, move the code fragment to a separate method with an appropriate name;
- Inline method – where a method is short and the functionality defined by the method body is clear, remove the method and replace all calls to the method with the method body;
- Move method – where a method is using more features of another class than the one in which it is defined, move the method to that other class;
- Remove Parameter – where a parameter is no longer required for a method but has been left in the parameter list anyway because it doesn't cause any problems, it can safely be removed;
- Substitute algorithm – where a complex algorithm is found in a method and can be replaced by a simpler algorithm;
- Extract class – where a classes has become large and difficult to understand, it can be split by creating a new class containing a subset of the members of the original class;

³ Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D., "Refactoring: Improving the Design of Existing Code", Addison-Wesley Professional, 1999.

⁴ The refactorings from the textbook are also partly summarised on the web site: <http://www.refactoring.com/catalog/index.html>

- Inline class – where a class has been reduced to only a small function, it may be better to move the members of that class directly into those classes that use it;
- Extract superclass – where two or more classes have the same members (duplicate code) it is preferable to move those common members to a common base class; and
- Extract interface – where a subset of a classes members are used regularly, it is useful to create an interface defining those members (which the class then implements) to improve reusability.