# 29. Method Combination

In this section we will primarily study *method combination*. Secondarily we will touch on a more specialized, related problem called *parameter variance*.

## 29.1. Method Combination

Lecture 8 - slide 2

If two or more methods, of the same name, located different places in a class hierarchy, cooperate to solve some problem we talk about *method combination*.

A typical (and minimal) scene is outlined in Figure 29.1. Class B is a subclass of A, and in both classes there is a method named *Op*. Both *Op* methods have the same signature.

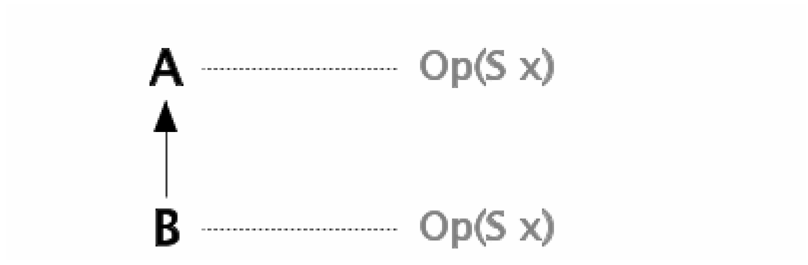

Figure 29.1 *Class B is a subclass of class A*

Overall, and in general, there are several ways for *Op* in class A and B to cooperate. We can, for instance, imagine that whenever a B-object receives an *Op* message, both operations are called automatically. We can also imagine that *Op* in class A is called explicitly by *Op* in class B, or the other way around.

Along the lines outlined above, we summarize two different method combination ideas. The first is known as *imperative method combination,* and the second is known as *declarative method combination*.

- Programmatic (imperative) control of the combination of *Op* methods
    - *Superclass controlled*: The *Op* method in class A controls the activation of *Op* in class B
    - *Subclass controlled*: The *Op* method in class B controls the activation of *Op* in class A
    - *Imperative method combination*
- An overall (declarative) pattern controls the mutual cooperation among *Op* methods
    - A.*Op* does not call B.*Op* - B.*Op* does not call A.*Op*.
    - A separate abstraction controls how *Op* methods in different classes are combined
    - *Declarative method combination*

Mainstream object-oriented programming languages, including C#, support imperative method combination. Most of them support the variant that we call subclass-controlled, imperative method combination.

Beta [Kristensen87] is an example of programming language with superclass-controlled, imperative method combination. CLOS [Steele90, Keene89] is one of the few examples of programming languages with declarative method combination. (The interested reader can consult Chapter 28 of [Steele90] to learn much more about declarative method combination in CLOS. )

C# supports subclass controlled, imperative method combination via use of the notation **`base.Op(...)`**

The notion **`base.Op(...)`** has been discussed in Section 28.7 and it has been illustrated in Program 26.2 (line 17), Program 28.13 (line 20), and Program 28.14 (line 20).

## 29.2. Parameter Variance

Lecture 8 - slide 3

We will continue the discussion of the scene outlined in Figure 29.1, now refined in Figure 29.2 shown below. The question is how the parameters of *Op* in class `A` and `B` vary in relation the variation of type `A` and type `B`.

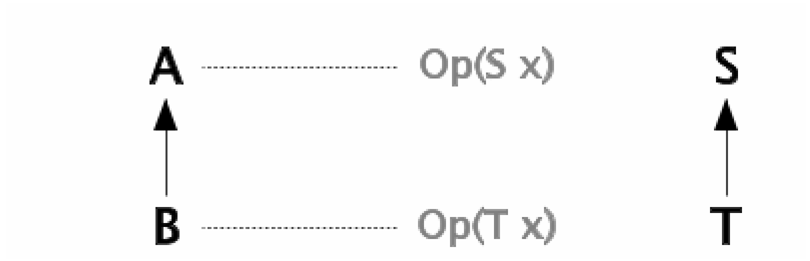

Figure 29.2 *Class B is a subclass of class A, and T is a subclass of S.*

In Program 29.1 we create an object of the specialized class `B` (in line 2), and we assign it to a variable of static type `A` (line 5) This is possible due to polymorphism. In line 6 we send the `Op` message to the `B` object. We assume that `Op` is virtual, and therefore we expect that `Op` in class `B` is called.

So far so good. The thing to notice is that `Op` takes a single parameter. If we pass an instance of class `S` to `B.Op` we may be in deep trouble. A problem occur if `B.Op` applies some operation from class `T` on the `S` object.

```
1  A aref;
2  B bref = new B();
3  S sref = new S();
4
5  aref = bref;     // aref is of static type A and dynamic type B
6  aref.Op(sref);  // B.Op is called with an S-object as parameter.
7                  // What if an operation from T is activated on the S-object?
```

Program 29.1 *An illustration of the problems with covariance.*

In Program 29.2 (only on web) in the web-edition we show a complete C# program which illustrates the problem.

The story told about the scene in Program 29.1 and Program 29.2 (only on web) turns out to be flawed in relation to C#! I could have told you the reason, but I will not do so right away. You should rather take a look at Exercise 8.1 and learn the lesson the hard way. (When access is granted to the exercise solutions, you will be able to get my explanation).

It turns out that parameter variance is not really a relevant topic in C#...

## 29.3. Covariance and Contravariance

Lecture 8 - slide 4

The situation encountered in Figure 29.2 of Section 29.2 is called *covariance*, because the types `S` and `T` (as occurring in the parameters of `Op` in class `A` and `B`) vary the same way as classes `A` and `B`. (The parameter type `T` of `Op` in class `B` is a subclass of the parameter type `S` of `Op` in class `A`; The class `B` is a subclass of class `A`; Therefore we say that `T` and `S` vary the same way as `A` and `B`. )

- **Covariance:** The parameters `S` and `T` vary the same way as `A` and `B`

As a contrast, the situation in Figure 29.3 below is called *contravariance*, because - in this variant of the scene - `S` and `T` vary in the opposite way as `A` and `B`. Please compare carefully Figure 29.2 with Figure 29.3.

- **Contravariance**: The parameters `S` and `T` vary the opposite way as `A` and `B`

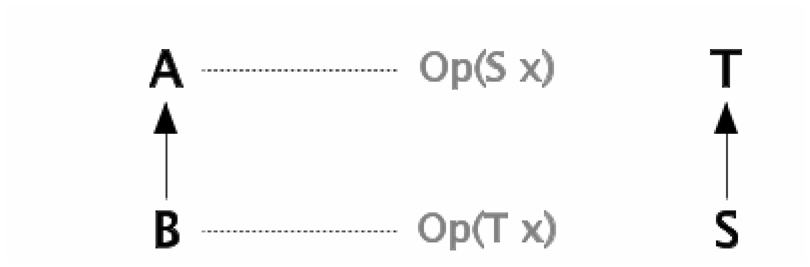

Figure 29.3 *Class B is a subclass of class A, and the parameter class S is a subclass of T.*

As we will see in Exercise 8.1 the distinction between covariance and contravariance is less relevant in C#. However, covariance and contravariance show up in other contexts of C#. See Section 42.6.

**Exercise 8.1.** *Parameter variance*

First, be sure you understand the co-variance problem stated above. Why is it problematic to execute `aref.Op(sref)`in the class Client?

The parameter variance problem, and the distinction between covariance and contravariance, is not really a topic in C#. The program with the classes A/B/S/T on the previous page compiles and runs without problems. Explain why!

## 29.4. References

| | |
|---|---|
| [Keene89] | Sonya E. Keene, *Object-Oriented Programming in Common Lisp*. Addison-Wesley Publishing Company, 1989. |
| [Steele90] | Guy L. Steele, *Common Lisp, the language, 2nd Edition*. Digital Press, 1990. |
| [Kristensen87] | Bent Bruun Kristensen, Ole Lehrmann Madsen, Birger Møller-Pedersen and Kristen Nygaard, "The BETA Programming Language". In *Research Directions in Object-Oriented Programming*, The MIT Press, 1987. Bruce Shriver and Peter Wegner (editors) |

# 30. Abstract Classes - Sealed Classes

This chapter is about *abstract classes*. At the end of the chapter we also touch on *sealed classes*. Relative to our interests, sealed classes are less important than abstract classes.

## 30.1. Abstract Classes

Lecture 8 - slide 6

When we program in the object-oriented paradigm it is important to work out concepts as general as possible. Programming at a general level promotes reusability (see Section 2.4).

In object-oriented programming languages we organize classes in hierarchies. The classes closest to the root are the most general classes. Take, as an example, the bank account class hierarchy in Section 25.3, where the class `BankAccount` is more general than `CheckAccount`, `SavingsAccount`, etc. It is worth noticing, however, that we were able to fully implement all operations in the most general class, `BankAccount`. In the rest of this chapter we will study even more general classes, for which we cannot (or will not) implement all the operations. The non-implemented operations are stated as *declarations of intent* at the most general level. These declarations of intent should be realized in less general subclasses.

Abstract classes are used for concepts that we cannot or will not implement in full details

Here follows our definition of an abstract class and an abstract operation.

An *abstract class* is a class with one or more abstract operations

An *abstract operation* is specially marked operation with a name and with formal parameters, but without a body

An abstract class

- may announce a number of abstract operations, which must be supplied in subclasses
- cannot be instantiated
- is intended to be completed/finished in a subclass

We will sometimes use the term *concrete class* for a class which is not abstract.

You should be aware that the definition of an abstract class, as given below, is not 100% accurate in relation to C#. In C# a class can be abstract without announcing abstract operations. More about that in Section 30.2 below, where we discuss abstract classes in C#.

The fact that an abstract class cannot be instantiated is the most tangible, operational consequence of working with abstract classes. Many OOP programmers tend to think of the `abstract` modifier as a mark, to be associated with those classes, he or she does not wish to instantiate. Surely, this is a consequence, but it is not the essential idea behind abstract classes.

## 30.2. Abstract classes and abstract methods in C#

Lecture 8 - slide 7

We will first study an example of an abstract class. We pick an abstract stack. (This is indeed a very popular example class in many contexts. We have tried to avoid it, but here it fits nicely).

The abstract class `Stack`, shown in Program 30.1, is remarkable of two reasons:

1. There is no data representation in the class (no instance variables).
2. There is a fully implemented operation in the class, despite the fact that the class has no data for the operation to work on.

The **blue** parts of Program 30.1 are the abstract operations. These operations make up the classical stack operations `Push`, `Pop`, and `Top` together with `Full`, `Empty`, and `Size`. (Notice that `Top`, `Full`, `Empty` and `Size` are announced as properties, cf. Section 30.3). The abstract operations have signatures (method heads), but no body blocks. In a real-life version of the program we would certainly have supplied documentation comments with some additional explanations of the roles of the abstract operations in the class.

The **purple** part represents a fully implemented, "normal" method, called `ToggleTop`. This method swaps the order of the two top-most elements of the stack (if available). Notice that `ToggleTop` can be implemented solely in terms of the `Push`, `Pop`, `Top` and `Size`. In other words, it is not necessary for the implementation of `ToggleTop` to know details of the concrete data representation of stacks.

```
using System;

public abstract class Stack{

  abstract public void Push(Object el);

  abstract public void Pop();

  abstract public Object Top{
    get;}

  abstract public bool Full{
    get;}

  abstract public bool Empty{
    get;}

  abstract public int Size{
    get;}

  public void ToggleTop(){
    if (Size >= 2){
      Object topEl1 = Top;  Pop();
      Object topEl2 = Top;  Pop();
      Push(topEl1); Push(topEl2);
    }
  }

  public override String ToString(){
    return String.Format("Stack[{0}]", Size );
  }
}
```

Program 30.1 *An abstract class Stack - without data representation - with a non-abstract ToggleTop method.*

In Program 30.1 the method `ToString` is also an example of a fully implemented method, which relies on an abstract method, namely `Size`.

It is left as an exercise to implement a non-abstract subclass of the abstract stack, see Exercise 8.3.

Let us state some more detailed - a perhaps slightly surprising - observations about abstract classes and abstract operations. Each of them will be discussed below.

- Abstract classes
  - can be derived from a non-abstract class
  - do not need not to have abstract members
  - can have constructors
- Abstract methods
  - are implicitly virtual

In relative rare situations an abstract class can inherit from a non-abstract class. Notice, however, that even abstract classes inherit (at least implicitly) from class `Object`, which is a non-abstract class in C#. (In principle, it would make good sense for the designers of C# to implement class `Object` as abstract class. But they did not! We only rarely make instances of class `Object`).

The next observation is about fully implemented classes, which we mark as being abstract. As discussed above, the purpose of this marking is to prevent instantiation of the class.

You may ask if it makes sense to have constructors in a class which never is instantiated. The answer is yes, because the data encapsulated in an abstract class `A` should be initialized when a concrete subclass of `A` is instantiated. Due to the rules of constructor cooperation, see Section 28.4 and Section 28.5, a constructor of class `A` will be activated. If no constructor is present in `A`, this falls back on the parameter-less default constructor.

Finally, we observe that the abstract methods are implicitly virtual. This is natural, because such a method has to be (re)defined in a subclass. In C# it is not allowed explicitly to write "`virtual abstract`" in front of an abstract method. Let us also observe, that an abstract method *M* cannot be private. This is because *M* need to be visible in the classes that override *M*.

---

**Exercise 8.2.** *Course and Project classes*

In the earlier exercise about courses and projects (found in the lecture about classes) we programmed the classes `BooleanCourse`, `GradedCourse`, and `Project`. Revise and reorganize your solution (or the model solution) such that `BooleanCourse` and `GradedCourse` have a common abstract superclass called `Course`.

Be sure to implement the method `Passed` as an abstract method in class `Course`.

In the `Main` method (of the client class of `Course` and `Project`) you should demonstrate that both boolean courses and graded courses can be referred to by variables of static type `Course`.

---

**Exercise 8.3.** *A specialization of Stack*

On the slide to which this exercise belongs, we have shown an abstract class `Stack`.

It is noteworthy that the abstract `Stack` is programmed without any instance variables (that is, without any data representation of the stack). Notice also that we have been able to program a single non-abstract method `ToggleTop`, which uses the abstract methods `Top`, `Pop`, and `Push`.

Make a non-abstract specialization of `Stack`, and decide on a reasonable data representation of the stack.

In this exercise it is OK to ignore exception/error handling. You can, for instance, assume that the capacity of the stack is unlimited; That popping an empty stack an empty stack does nothing; And that the top of an empty stack returns the string "Not Possible". In a later lecture we will revisit this exercise in order to introduce exception handling. Exception handling is relevant when we work on full or empty stacks.

Write a client of your stack class, and demonstrate the use of the inherited method `ToggleTop`. If you want, you can also adapt my stack client class which is easily available to you in the web-edition of this material.

---

## 30.3. Abstract Properties

Lecture 8 - slide 8

Properties were introduced in Chapter 18. Recall that properties allow us to get and set data of a class through getter and setter abstractions. From an application point of view, properties are used in the same way as variables - both on the left and right hand sides of assignments. Underneath, a property is realized as two methods - one "getter" and one "setter".

Properties can be abstract in the same way as methods. It means that we can announce a number of properties which must be fully defined in subclasses. We will in Program 30.2 study an example of abstract properties, namely in a `Point` class called `AbstractPoint`, which can be accessed both in a rectangular (*x*, *y*) and a polar (*r*, *a*) way. *r* and *a* means radius and angle respectively. There is no data (variables) in class `AbstractPoint`. We announce `X`, `Y`, `R` and `A` as abstract properties. These are emphasized using **purple** color. All of these are announced as both getters and setters. Notice the `get; set;` syntax. We could alternatively announce these as only getters, or as only setters. We notice that the syntax of abstract properties is similar to the syntax used for automatic properties, see Section 18.3.

Following the abstract properties comes three noteworthy methods `Move`, `Rotate` and `ToString`. They are shown in **blue**. They all use make heavy use the abstract properties. The assignment `X += dx` in Move, for instance, expands to `X = X + dx`. It first uses the getter of the `X` property on the right hand side of the assignment. Next, it uses the `X` setter on the left hand side. In Program 30.2 we only know that the `X` getter and the `X` setter exist. The actual implementation details will be found in a subclass.

In the web-edition of this material, we show a version of class `AbstractPoint` with four additional protected, static methods which are useful for the implementation of the subclasses.

```
using System;

abstract public class AbstractPoint {

  public enum PointRepresentation {Polar, Rectangular}

  // We have not yet decided on the data representation of Point

  public abstract double X {
    get ;
    set ;
  }

  public abstract double Y {
    get ;
    set ;
  }

  public abstract double R {
    get ;
    set ;
  }

  public abstract double A {
    get ;
    set ;
  }

  public void Move(double dx, double dy){
    X += dx;  Y += dy;
  }

  public void Rotate(double angle){
    A += angle;
  }

  public override string ToString(){
    return  "(" + X + ", " + Y + ")" + " " +  "[r:" + R + ", a:" + A + "]  ";
  }

}
```

Program 30.2 *The abstract class Point with four abstract properties.*

In Program 30.3 we see a subclass of `AbstractPoint`. It is called `Point`. It happens to represent points the polar way. But this is an internal (private) detail of class `Point`.

Class `Point` is a non-abstract class, and therefore we program a constructor, which is emphasized in **black**. The constructor is a little unconventional, because the first parameter allows us to specify if parameter two and three means `x`, `y` or `radius`, `angle`. It is desirable if this could be done more elegantly. (It can! Use of static factory methods, see Section 16.4, is better). Notice that `PointRepresentation` is an enumeration type defined in line 5 of Program 30.2.

Emphasized in **purple** we show the actual implementation of the `X` and `Y` properties. Let us look at `X`. The getter of `X` is called whenever `X` is used as a right-hand side value. It calculates the x-coordinate of a point from the radius and the angle. The setter of `X` is called when `X` is used in left-hand side context, such as `X = `*e*. The value of expression *e* is bound to the pseudo variable **`value`**. The setter calculates new radius and angle values which are assigned to the instance variables of class `Point`.

Emphasized in **blue** we show the implementation of the `R` and `A` properties. These are trivial compared to the `X` and `Y` properties, because we happen to represent points in the polar way.

```
using System;

public class Point: AbstractPoint {

  // Polar representation of points:
  private double radius, angle;            // radius, angle

  // Point constructor:
  public Point(PointRepresentation pr, double n1, double n2){
    if (pr == PointRepresentation.Polar){
      radius = n1; angle = n2;
    }
    else if (pr == PointRepresentation.Rectangular){
      radius = RadiusGivenXy(n1, n2);
      angle  = AngleGivenXy(n1, n2);
    } else {
      throw new Exception("Should not happen");
    }
  }

  public override double X {
    get {
      return XGivenRadiusAngle(radius, angle);}
    set {
      double yBefore = YGivenRadiusAngle(radius, angle);
      angle = AngleGivenXy(value, yBefore);
      radius = RadiusGivenXy(value, yBefore);
    }
  }

  public override double Y {
    get {
      return YGivenRadiusAngle(radius, angle);}
    set {
      double xBefore = XGivenRadiusAngle(radius, angle);
      angle = AngleGivenXy(xBefore, value);
      radius = RadiusGivenXy(xBefore, value);
    }
  }

  public override double R {
    get {
     return radius;}
    set {
     radius = value;}
  }

  public override double A {
    get {
     return angle;}
    set {
     angle = value;}
  }

}
```

Program 30.3 *A non-abstract specialization of class Point (with private polar representation).*

In the web-edition we show a client of `AbstractPoint` and `Point`, which is similar to Program 11.3 from Section 11.6. It shows how to manipulate instances of class `Point` via its abstract interface.

Let us summarize what we have learned from the examples in Program 30.2, Program 30.3, and Program 30.4 (only on web). First and foremost, we have seen an abstract class in which we are able to implement useful functionality (`Move`, `Rotate`, and `ToString`) *at a high level of abstraction*. The implementation details in the mentioned methods rely on abstract properties, which are implemented in subclasses. We have also seen a sample subclass that implements the four abstract properties.

## 30.4. Sealed Classes and Sealed Methods

Lecture 8 - slide 9

We will now briefly, as the very last part of this chapter, describe sealed classes and sealed methods.

A sealed class C prevents the use of C as base class of other classes

- Sealed classes
  - Cannot be inherited by other classes
- Sealed methods
  - Cannot be redefined and overridden in a subclass
  - The modifier **`sealed`** must be used together with **`override`**

Sealed classes are related to static classes, see Section 11.12, in the sense that none of them can be subclassed. However, static classes are more restrictive because a static class cannot have instance members, a static class cannot be used as a type, and a static class cannot be instantiated. Sealed classes and methods correspond to final classes and final methods in Java.

In some sense, abstract and sealed classes represent opposite concepts. At least this holds in the following sense: A sealed class cannot be subclassed; An abstract class must be subclassed in order to be useful.

If a class is abstract it does not make sense that it is sealed. And the other way around, if a class is sealed it does not make sense that it, in addition, is abstract. Notice that it does not make sense either to have virtual methods in a sealed class.

A sealed class is not required to have sealed methods. Moreover, a class with a sealed method does not itself need to be sealed.

Finally, notice, that in C# a method cannot be sealed without also being overridden. Thus, the **`sealed`** modifier always occurs as an "extra modifier" of **`override`**. The intention of sealed methods is to prevent further overriding of virtual methods.

# 31. Interfaces

Interfaces form a natural continuation of abstract classes, as discussed in Chapter 30. In this chapter we will first introduce the interface concept. Then follows an example, which illustrates the power of interfaces. Finally, we review the use of interfaces in the C# libraries.

## 31.1. Interfaces

Lecture 8 - slide 11

An interface announces a number of operations in terms of their signatures (names and parameters). An interface does not implement any of the announced operations. An interface only declares an intent, which eventually will be realized by a class or a struct.

A class or struct can implement an arbitrary number of interfaces. Inheritance of multiple classes may be problematic in case the same variable or (fully defined) operation is inherited from several superclasses, see Section 27.5. Inheritance of the same intent from multiple interfaces is less problematic. In a nutshell, this explains one of the reasons behind having interfaces in C# and Java, instead of having multiple class-inheritance, like in for instance C++ and Eiffel.

An interface can inherit an arbitrary number of other interfaces. This makes it convenient to organize a small set of inter-dependent operations in a single interfaces, which then can be combined (per inheritance) with several other interfaces, classes or structs.

An interface corresponds to a class where all operations are abstract, and where no variables are declared. In Section 30.1 we argued that abstract classes are useful as general, high-level program contributions. This is therefore also the case for interfaces.

An *interface* describes signatures of operations, but it does not implement any of them

Here follows the most important characteristics of interfaces:

- Classes and structs can implement one or more interfaces
- An interface can be used as a type, just like classes
    - Variables and parameters can be declared of interface types
- Interfaces can be organized in multiple inheritance hierarchies

Let us dwell on the observation that an interface serves as a type. We already know that classes and structs can be used as types. It means that we can have variables and parameters declared as class or struct types. The observation from above states that interfaces can be used the same way. Thus, it is possible to declare variables and parameters of an interface type. But wait a moment! It is not possible to instantiate an interface. So which values can be assigned to variables of an interface type? The answer is that objects or values of class of struct types, which implement the interface, can be assigned to variables of the interface type. This gives a considerable flexibility in the type system, because arbitrary types in this way can be made compatible, by letting them implement the same interface(s). We will see an example of that in Section 31.3.

**Exercise 8.4.** *The interface ITaxable*

For the purpose of this exercise you are given a couple of very simple classes called `Bus` and `House`. Class `Bus` specializes the class `Vehicle`. Class `House` specializes the class `FixedProperty`. The mentioned classes can easily be accessed from the web-edition of the material..

First in this exercise, program an interface `ITaxable` with a parameterless operation `TaxValue`. The operation should return a decimal number.

Next, program variations of class `House` and class `Bus` which implement the interface `ITaxable`. Feel free to invent the concrete taxation of houses and busses. Notice that both class `House` and `Bus` have a superclass, namely `FixedProperty` and `Vehicle`, respectively. Therefore it is essential that taxation is introduced via an interface.

Demonstrate that taxable house objects and taxable bus objects can be used together as objects of type `ITaxable`.

---

## 31.2. Interfaces in C#

Lecture 8 - slide 12

Let us now be more specific about interfaces in C#. The operations, described in a C# interface, can be methods, properties, indexers, or events.

| Both classes, structs and interfaces can implement one or more interfaces |
|---|
| Interfaces can contain signatures of methods, properties, indexers, and events |

The syntax involved in definition of C# interfaces is summarized in Syntax 31.1. The first few lines describe the structure of an interface as such. The remaining part of Syntax 31.1 outlines the descriptions of interface methods, properties, indexers and events respectively.

```
modifiers interface interface-name : base-interfaces {
  method-descriptions
  property-descriptions
  indexer-descriptions
  event-descriptions
}

return-type method-name(formal-parameter-list);

return-type property-name{
  get;
  set;
}

return-type this[formal-parameter-list]{
  get;
  set;
}
```

```
event delegate-type event-name;
```

Syntax 31.1 *The syntax of a C# interface, together with the syntaxes of method, property, indexer, and event descriptions in an interface*

## 31.3. Examples of Interfaces

Lecture 8 - slide 13

Earlier in this material we have programmed dice and playing cards, see Program 10.1 and Program 12.7. Do the concepts behind these classes have something in common? Well - they are both used in a wide variety of games. This observation causes us to define an interface, `IGameObject`, which is intended to capture some common properties of dice, playing cards, and other similar types. Both class `Die` and class `Card` should implement the interface `IGameObject`.

As a matter of C# coding style, all interfaces start with a capital 'I' letter. This convention makes it obvious if a type is defined by an interface. This naming convention is convenient in C#, because classes and interface occur together in the inheritance clause of a class. (Both the superclass and the interfaces occur after a colon, the class first, cf. Syntax 28.1). In this respect, C# is different from Java. In Java, interfaces and classes are marked with the keywords **`extends`** and **`implements`** respectively in the inheritance clause of a class.

Two or more unrelated classes can be used together if they implement the same interface

```
public enum GameObjectMedium {Paper, Plastic, Electronic}

public interface IGameObject{

  int GameValue{
    get;
  }

  GameObjectMedium Medium{
    get;
  }
}
```

Program 31.1 *The interface IGameObject.*

The `IGameObject` interface in Program 31.1 prescribes two named properties: `GameValue` and `Medium`. Thus, classes that implement the `IGameObject` must define these two properties. Notice, however, that no semantic constraints on `GameValue` or `Medium` are supplied. (It means that no *meaning* is prescribed). Thus, classes that implement the interface `IGameObject` are, in principle, free to supply arbitrary bodies of `GameValue` and `Medium`. This can be seen as a weakness. In Chapter 50 we will see how to remedy this by specifying the semantics of operations in terms of preconditions and postconditions.

Notice also that there are no visibility modifiers of the operations `GameValue` and `Medium` in the interface shown above. All operations are implicitly public.

Below, in Program 31.2, we show a version of class `Die`, which implements the interface `IGameObject`. In line 3 it is stated that class `Die` implements the interface. The actual implementations of the two operations are shown in the bottom part of Program 31.2 (from line 33 to 44). Most interesting, the `GameValue` of a die is the current number of eyes.

```
using System;

public class Die: IGameObject {
  private int numberOfEyes;
  private Random randomNumberSupplier;
  private readonly int maxNumberOfEyes;

  public Die (): this(6){}

  public Die (int maxNumberOfEyes){
    randomNumberSupplier =
      new Random(unchecked((int)DateTime.Now.Ticks));
    this.maxNumberOfEyes = maxNumberOfEyes;
    numberOfEyes = NewTossHowManyEyes();
  }

  public void Toss (){
    numberOfEyes = NewTossHowManyEyes();
  }

  private int NewTossHowManyEyes (){
    return randomNumberSupplier.Next(1,maxNumberOfEyes + 1);
  }

  public int NumberOfEyes() {
    return numberOfEyes;
  }

  public override String ToString(){
    return String.Format("Die[{0}]: {1}", maxNumberOfEyes, numberOfEyes);
  }

  public int GameValue{
    get{
      return numberOfEyes;
    }
  }

  public GameObjectMedium Medium{
    get{
      return
       GameObjectMedium.Plastic;
    }
  }

}
```

Program 31.2 *The class Die which implements IGameObject.*

In Program 31.3 we show a version of class `Card`, which implements our interface. The `GameValue` of a card is, quite naturally, the card value.

```
using System;

public class Card: IGameObject{
  public enum CardSuite { spades, hearts, clubs, diamonds };
  public enum CardValue { two = 2, three = 3, four = 4, five = 5,
                          six = 6, seven = 7, eight = 8, nine = 9,
                          ten = 10, jack = 11, queen = 12, king = 13,
                          ace = 14 };

  private CardSuite suite;
  private CardValue value;

  public Card(CardSuite suite, CardValue value){
    this.suite = suite;
    this.value = value;
  }

  public CardSuite Suite{
    get { return this.suite; }
  }

  public CardValue Value{
    get { return this.value; }
  }

  public override String ToString(){
    return String.Format("Suite:{0}, Value:{1}", suite, value);
  }

  public int GameValue{
    get { return (int)(this.value); }
  }

  public GameObjectMedium Medium{
    get{
      return GameObjectMedium.Paper;
    }
  }
}
```

Program 31.3 *The class Card which implements IGameObject.*

Below, in Program 31.4 we have written a program that works on game objects of type `IGameObject`. In order to be concrete - and somewhat realistic - we make an `IGameObject` array with three die objecs and three card objects. In the bottom part of the program we exercise the common operations of dice and playing cards, as prescribed by the interface `IGameObject`. The output of the program is shown in Listing 31.5.

```
using System;
using System.Collections.Generic;

class Client{

  public static void Main(){

    Die d1 = new Die(),
        d2 = new Die(10),
        d3 = new Die(18);

    Card c1 =  new Card(Card.CardSuite.spades, Card.CardValue.queen),
         c2 =  new Card(Card.CardSuite.clubs, Card.CardValue.four),
         c3 =  new Card(Card.CardSuite.diamonds, Card.CardValue.ace);

    IGameObject[] gameObjects = {d1, d2, d3, c1, c2, c3};

    foreach(IGameObject gao in gameObjects){
      Console.WriteLine("{0}: {1} {2}",
                        gao, gao.GameValue, gao.Medium);
    }
  }
}
```

Program 31.4 *A sample Client program of Die and Card.*

```
Die[6]: 5: 5 Plastic
```

```
Die[10]: 9: 9 Plastic
Die[18]: 15: 15 Plastic
Suite:spades, Value:queen: 12 Paper
Suite:clubs, Value:four: 4 Paper
Suite:diamonds, Value:ace: 14 Paper
```

Listing 31.5 *Output from the sample Client program of Die and Card.*

Above, both `Die` (see Program 31.2) and `Card` (see Program 31.3) are classes. We have in Exercise 4.2 noticed that it would be natural to implement the type `Card` as a struct, because a playing card - in contrast to a die - is immutable. The client class shown in Program 31.4 will survive if we program `Card` as a struct, and it will produce the same output as shown in Listing 31.5. Recall in this context that interfaces in C# are reference types, see Section 13.3. When a variable of static type `IGameObject` is assigned to a value of struct type `Card`, the card value is boxed. Boxing is described in Section 14.8.

In the example above, where both the types `Die` and `Card` are implemented as classes, `IGameObject` could as well have been implemented as an abstract superclass. This is the theme in Exercise 8.5.

---

**Exercise 8.5.** *An abstract GameObject class*

On the slide, to which this exercise belongs, we have written an interface `IGameObject` which is implemented by both class `Die` and class `Card`.

Restructure this program such that class `Die` and class `Card` both inherit an abstract class `GameObject`. You should write the class `GameObject`.

The client program should survive this restructuring. (You may, however, need to change the name of the type `IGameObject` to `GameObject`). Compile and run the given client program with your classes.

---

## 31.4. Interfaces from the C# Libraries

Lecture 8 - slide 14

The C# library contains a number of important interfaces which are used frequently in many C# programs

In this section we will discuss some important interfaces from the C# libraries. First, we give an itemized overview, and in the sections following this one more details will be provided.

- `IComparable`
  - An interface that prescribes a `CompareTo` method
  - Used to support general sorting and searching methods
- `IEnumerable`
  - An interface that prescribes a method for accessing an enumerator
- `IEnumerator`
  - An interface that prescribes methods for traversal of data collections
  - Supports the underlying machinery of the **`foreach`** control structure
- `IDisposable`

    - An interface that prescribes a `Dispose` method
    - Used for deletion of resources that cannot be deleted by the garbage collector
    - Supports the C# **`using`** control structure
- `ICloneable`
    - An interface that prescribes a `Clone` method
- `IFormattable`
    - An interface that prescribes an extended `ToString` method

`IComparable` is touched on in Section 31.5, primarily via an exercise. In Section 31.6 we focus on the interfaces `IEnumerable` and `IEnumerator` and their roles in the realization of **`foreach`** loops. Type parameterized versions of these interfaces are discussed in Section 45.2. The interfaces `IDisposable` is discussed in the context of IO in Section 37.5. `ICloneable` is discussed in a later chapter, see Section 32.7.

All the interfaces mentioned above can be thought of as *flavors* that can be added to many different classes.

## 31.5. Sample use of IComparable

Lecture 8 - slide 15

Object of classes that implement **`IComparable`** can be sorted by a method such as **`Array.Sort`**

In many contexts it is important to be able to state that two objects or values, say *x* and *y* of a particular type *T*, can be compared to each other. Thus, we may be curious to know if $x < y$, $y < x$, or if $x = y$. But what does $x < y$, $y > x$, and $x = y$ mean if, for instance type *T* is `BankAccount` or a `Die`?

The way we approach this problem is to arrange that the type *T* (a class or a struct) implements the interface `IComparable`. In that way, the implementation of *T* must include the method `CompareTo`, which can be used in the following way:

```
x.CompareTo(y)
```

In the tradition of, for instance, the string comparison function `strcmp` in the C standard library `string.h` the expression *x*.`CompareTo`(*y*) returns a negative integer result if *x* is considered less than *y*, a positive integer if *x* is considered greater than *y*, and integer zero if *x* and *y* are considered to be equal.

The interface `IComparable` is reproduced in Program 31.6. This shows you how simple it is. Don't use this or a similar definition. Use the interface `IComparable` as predefined in the `System` namespace.

```
using System;

public interface IComparable{
  int CompareTo(Object other);
}
```

Program 31.6 *A reproduction of the interface IComparable.*

The parameter of `CompareTo` is of type `Object`. This is irritating because we will almost certainly want the parameter to be of the same type as the class, which implements `Icomparable`. When you solve Exercise 8.6 you will experience this.

There is actually two versions of the interface `IComparable` in the C# libraries. The one similar to Program 31.6 and a type parameterized version, which constrains the parameter of the `CompareTo` method to a given type `T`. We have more say about these two interfaces in Section 42.8.

It is also worthwhile to point out the interface `IEquatable`, which simply prescribes an `Equals` method. The interface `IEqualityComparer` is a cousin interface which in addition to `Equals` also prescribes `GetHashCode`. In some sense `IEquatable` and `IEqualityComparer` are more fundamental than `IComparable`. It turns out that `IEquatable` only exists as a type parameterized (generic) interface.

**Exercise 8.6.** *Comparable Dice*

In this exercise we will arrange that two dice can be compared to each other. The result of `die1.CompareTo(die2)` is an integer. If the integer is negative, `die1` is considered less than `die2`; If zero, `die1` is considered equal to `die2`; And if positive, `die1` is considered greater than `die2`. When two dice can be compared to each other, it is possible sort an array of dice with the standard `Sort` method in C#.

Program a version of class `Die` which implements the interface `System.IComparable`.

Consult the documentation of the (overloaded) static method `System.Array.Sort` and locate the `Sort` method which relies on `IComparable` elements.

Make an array of dice and sort them by use of the `Sort` method.

## 31.6. Sample use of IEnumerator and IEnumerable

Lecture 8 - slide 16

In this section we will study the interfaces called `IEnumerator` and `IEnumerable`. The interface `IEnumerator` is central to the design pattern called ***Iterator***, which we will discuss in the context of collections, see Section 48.1. As already mentioned above, the interface `IEnumerator` also prescribes the operations behind the **`foreach`** control structure.

```
using System;

public interface IEnumerator{
  Object Current{
    get;
  }

  bool MoveNext();

  void Reset();
}
```

Program 31.7 *A reproduction of the interface IEnumerator.*

We have reproduced `IEnumerator` from the `System.Collections` namespace in Program 31.7. The operations `Current`, `MoveNext`, and `Reset` are used to traverse a collection of data. Hidden behind the interface should be some simple bookkeeping which allows us to keep track of the current element, and which element is next. You can think of this as a *cursor*, which step by step is moved through the collection. The property `Current` returns the element pointed out by the cursor. The method `MoveNext` advances the cursor, and it returns true if it has been possible to move the cursor. The method `Reset` moves the cursor to the first element, and it resets the bookkeeping variables.

You are not allowed to modify the collection while it is traversed via a C# enumerator. Notice, in particular, that you are not allowed to delete the element obtained by `Current` during a traversal. In that respect, C# enumerators are more limited than the `Iterator` counterpart in Java which allows for exactly one deletion for each movement in the collection. It can also be argued that the `IEnumerator` interface is too narrow. It would be nice to have a boolean `HasNext` property. It could also be worthwhile to have an extended enumerator with a `MovePrevious` operation.

Like it was the case for the interface `Comparable`, as discussed in Section 31.5, there is also a type parameterized version of `IEnumerator`. See Section 45.2 for additional details.

```
using System.Collections;

public interface IEnumerable{
  IEnumerator GetEnumerator();
}
```

Program 31.8 *A reproduction of the interface IEnumerable.*

The `IEnumerable` interface, as reproduced in Program 31.8, only prescribes a single method called `GetEnumerator`. This method is intended to return an object (value), the class (struct) of which implements the `IEnumerator` interface. Thus, if a type implements the `IEnumerable` interface, it can deliver an iterator/enumerator object via use of the operation `GetEnumerator`.

As mentioned above, the **`foreach`** control structure is implemented by means of enumerators. The **`foreach`** form

```
  foreach(ElementType e in collection) statement
```

is roughly equivalent with

```
  IEnumerator en = collection.GetEnumerator();
  while (en.MoveNext()){
    ElementType e = (ElementType) en.Current();
    statement;
  }
```

The type of the collection is assumed to implement the interface **`IEnumerable`**. Additional fine details should be taken into consideration. Please consult section 15.8.4 of the C# Language Specification [ECMA-334] or [Hejlsberg06] for the full story.

We will now present a realistic example that uses `IEnumerator` and `IEnumerable`. We return to the `Interval` type, which we first met when we discussed overloaded operators in Section 21.3. The original `Interval` struct appeared in Program 21.3. Recall that an interval, such as [5 - 10] is different from [10 -5]. The former represents the sequence 5, 6, 7, 8, 9, 10 while the latter represents 10, 9, 8, 7, 6, 5. In the version we show in Program 31.9 we have elided the operators from Program 21.3.

The enumerator functionality is programmed in a private, local class called `IntervalEnumerator`, starting at line 39. This class implements the interface `IEnumerator`. The class `IntervalEnumerator` has a reference to the surrounding interval. (The reference to the surrounding object is provided via the constructor in line 44 and 68). It also has the instance variable `idx`, which represents of the cursor. Per convention, the value -1 represents an interval which has been reset. The property `Current` is now able to calculate and return a value from the interval. Notice that we have to distinguish between rising and falling intervals in the conditional expression in line 50-52. Both `MoveNext` and `Reset` are easy to understand if you have followed the details until this point.

The method `GetEnumerator` (line 67-69), which is prescribed by the interface, `IEnumerable` (see line 4), just returns an instance of the private class `IntervalEnumerator` discussed above. Notice that we in line 68 pass `this` (the current instance of the `Interval`) to the `IntervalEnumerator` object.

We show how to make simple traversals of intervals in Program 31.10.

```
1   using System;
2   using System.Collections;
3
4   public struct Interval: IEnumerable{
5
6     private readonly int from, to;
7
8     public Interval(int from, int to){
9       this.from = from;
10      this.to = to;
11    }
12
13    public int From{
14      get {return from;}
15    }
16
17    public int To{
18      get {return to;}
19    }
20
21    public int Length{
22      get {return Math.Abs(to - from) + 1;}
23    }
24
25    public int this[int i]{
26      get {if (from <= to){
27             if (i >= 0 && i <= Math.Abs(from-to))
28                 return from + i;
29             else throw new Exception("Error"); }
30           else if (from > to){
31             if (i >= 0 && i <= Math.Abs(from-to))
32                 return from - i;
33             else throw new Exception("Error"); }
34           else throw new Exception("Should not happen"); }
35    }
36
37    // Overloaded operators have been hidden in this version
38
39    private class IntervalEnumerator: IEnumerator{
40
41      private readonly Interval interval;
42      private int idx;
43
44      public IntervalEnumerator (Interval i){
45        this.interval = i;
46        idx = -1;   // position enumerator outside range
47     }
48 
49     public Object Current{
50          get {return (interval.From < interval.To) ?
51                        interval.From + idx :
52                        interval.From - idx;}
53     }
54 
55     public bool MoveNext (){
56       if ( idx < Math.Abs(interval.To - interval.From))
57          {idx++; return true;}
58       else
59          {return false;}
60     }
61 
62     public void Reset(){
63       idx = -1;
64     }
65   }
66 
67   public IEnumerator GetEnumerator (){
68     return new IntervalEnumerator(this);
69   }
70 
71 }
```

Program 31.9 *IEnumerator in the type Interval.*

While we are here, we will discuss the nested, local class `IntervalEnumerator` of class `Interval` a little more careful. Why is it necessary to pass a reference to the enclosing `Interval` in line 68? Or, in other words, why can't we access the `from` and `to` `Interval` instance variables in line 6 from the nested class? The reason is that an `IntervalEnumerator` object is not a 'subobject' of an `Interval` object. An `IntervalEnumerator` object is not really part of the enclosing `Interval` object. The `IntervalEnumerator` can, however, access (both public and private) class variables (static variables) of class `Interval`.

We could as well have placed the class `IntervalEnumerator` outside the class `Interval`, simply as a sibling class of `Interval`. But class `IntervalEnumerator` would just pollute the enclosing namespace. The `IntervalEnumerator` is only relevant inside the interval. Therefore we place it as a member of class `Interval`. By making it private we, furthermore, prevent clients of class `Interval` to access it.

Nested classes are, in general, a more advanced topic. It has, in part, something to do with scoping rules in relation to the outer classes, and in relation to superclasses. Java is more sophisticated than C# in its support of nested classes. In java, an inner class `I` in the surrounding class `C` is a nested class for which instances of `I` is connected to (is part of) a particular instance of `C`. See also our discussion of Java in relation to C# in Section 7.3.

```
using System;
using System.Collections;

public class app {

  public static void Main(){

    Interval iv1 = new Interval(14,17);

    foreach(int k in iv1){
      Console.Write("{0,4}", k);
    }
    Console.WriteLine();

    IEnumerator e = iv1.GetEnumerator();
    while (e.MoveNext()){
      Console.Write("{0,4}", (int)e.Current);
    }
    Console.WriteLine();
  }

}
```

Program 31.10 *Iteration with and without foreach based on the enumerator.*

## 31.7. Sample use of IFormattable

Lecture 8 - slide 17

The `IFormattable` interface prescribes a `ToString` method of two parameters. As such, the `ToString` method of `IFormattable` is different from the well-known `ToString` method of class `Object`, which is parameterless, see Section 28.3. Both methods produce a text string. The new `ToString` method is used when we need more control of the textual result.

Here follows a reproduction of `IFormattable` from the `System` namespace.

```
using System;

public interface IFormattable{
  string ToString(string format, IFormatProvider formatProvider);
}
```

Program 31.11 *A reproduction of the interface IFormattable.*

We can characterize the `ToString` method in the following way:

- The first parameter is typically a single letter formatting string, and the other is an `IFormatProvider`
- The `IformatProvider` can provide culture sensible information.
- `ToString` from `Object` typically calls `ToString(null, null)`

The first parameter of `ToString` is typically a string with a single character. For simple types as well as `DateTime`, a number of predefined formatting strings are defined. We have seen an example in Section 6.10. For the types we program we can define our own formatting letters. This is known as *custom formatting*. Below, in Program 31.12 we will show how to program custom formatting of a playing card struct.

The second parameter of `ToString` is of type `IFormatProvider`, which is another interface from the `System` namespace. An object of type `IFormatProvider` typically provides culture sensible formatting information. For simple types and for `DateTime`, a format provider represents details such as the currency symbol, the decimal point symbol, or time-related formatting symbols. If the second parameter is `null`, the object bound to `CultureInfo.CurrentCulture` should be used as the default format provider.

Below we show how to program custom formatting of struct `Card`, which we first met in the context of structs in Section 14.3. Notice that struct `Card` implements `Iformattable`. The details in the two `Tostring` methods should be easy to understand.

```
using System;

public enum CardSuite:byte
          {Spades, Hearts, Clubs, Diamonds };
public enum CardValue: byte
          {Ace = 1, Two = 2, Three = 3, Four = 4, Five = 5,
           Six = 6, Seven = 7, Eight = 8, Nine = 9, Ten = 10,
           Jack = 11, Queen = 12, King = 13};

public struct Card: IFormattable{
  private CardSuite suite;
  private CardValue value;

  public Card(CardSuite suite, CardValue value){
   this.suite = suite;
   this.value = value;
  }

  // Card methods and properties here...

  public System.Drawing.Color Color (){
   System.Drawing.Color result;
   if (suite == CardSuite.Spades || suite == CardSuite.Clubs)
     result = System.Drawing.Color.Black;
   else
     result = System.Drawing.Color.Red;
   return result;
  }

  public override String ToString(){
    return this.ToString(null, null);
  }

  public String ToString(string format, IFormatProvider fp){
    if (format == null || format == "G" || format == "L")
        return String.Format("Card Suite: {0}, Value: {1}, Color: {2}",
                             suite, value, Color().ToString());

    else if (format == "S")
        return String.Format("Card {0}: {1}", suite, (int)value);

    else if (format == "V")
        return String.Format("Card value: {0}", value);

    else throw new FormatException(
                     String.Format("Invalid format: {0}", format));
  }

}
```

Program 31.12 *The struct Card that implements IFormattable.*

In Program 31.13 we show how to make use of custom formatting of playing card objects. The resulting output can be seen in Listing 31.14.

```
using System;

class CardClient{

  public static void Main(){
    Card c1 = new Card(CardSuite.Hearts, CardValue.Eight),
         c2 = new Card(CardSuite.Diamonds, CardValue.King);

    Console.WriteLine("c1 is a {0}", c1);
    Console.WriteLine("c1 is a {0:S}", c1); Console.WriteLine();

    Console.WriteLine("c2 is a {0:S}", c2);
    Console.WriteLine("c2 is a {0:L}", c2);
    Console.WriteLine("c2 is a {0:V}", c2);


  }

}
```

Program 31.13 *A client of Card which applies formatting of cards.*

```
c1 is a Card Suite: Hearts, Value: Eight, Color: Color [Red]
c1 is a Card Hearts: 8

c2 is a Card Diamonds: 13
c2 is a Card Suite: Diamonds, Value: King, Color: Color [Red]
c2 is a Card value: King
```

Listing 31.14 *Output from the client program.*

## 31.8. Explicit Interface Member Implementations

Lecture 8 - slide 18

Interfaces give rise to multiple inheritance, and therefore we need to be able to deal with the challenges of multiple inheritance. These have already been discussed in Section 27.5.

The problems, as well as the C# solution, can be summarized in the following way:

If a member of an interface collides with a member of a class, the member of the interface can be implemented as an explicit interface member

Explicit interface members can also be used to implement several interfaces with colliding members

The programs shown below illustrate the problem and the solution. The class `Card`, in Program 31.15 has a `Value` property. The interface `IGameObject` in Program 31.16 also prescribes a `Value` property. (It is similar to the interface of Program 31.1 which we have encountered earlier in this chapter). When class `Card` implements `IGameObject` in Program 31.17 the new version of class `Card` will need to distinguish between its own `Value` property and the `Value` property it implements because of the interface `IGameObject`. How can this be done?

The solution to the problem is called *explicit interface member implementation*. In line 30-32 of Program 31.17, emphasized in **purple**, we use the `IgameObject.Value` syntax to make it clear that here we implement the `Value` property from `IGameObject`. This is an explicit interface implementation.

In the client classes of class `Card` we need access to both `Value` operations. In order to access the explicit interface implementation of `Value` from the `Card` variable `cs` (declared in line 6) we need to cast `cs` to the interface `IGameObject`. This is illustrated in line 14 of Program 31.18. The output of Program 31.18 in Listing 31.19 reveals that everything works as expected.

```
using System;

public class Card{
  public enum CardSuite { spades, hearts, clubs, diamonds };
  public enum CardValue { two = 2, three = 3, four = 4, five = 5,
                          six = 6, seven = 7, eight = 8, nine = 9,
                          ten = 10, jack = 11, queen = 12, king = 13,
                          ace = 14 };

  private CardSuite suite;
  private CardValue value;

  public Card(CardSuite suite, CardValue value){
    this.suite = suite;
    this.value = value;
  }

  public CardSuite Suite{
    get { return this.suite; }
  }

  public CardValue Value{
    get { return this.value; }
  }

  public override String ToString(){
    return String.Format("Suite:{0}, Value:{1}", suite, value);
  }
}
```

Program 31.15 *The class Playing card with a property Value.*

```
public enum GameObjectMedium {Paper, Plastic, Electronic}

public interface IGameObject{

  int Value{
    get;
  }

  GameObjectMedium Medium{
    get;
  }
}
```

Program 31.16 *The Interface IGameObject with a conflicting Value property.*

```
using System;

public class Card: IGameObject{
  public enum CardSuite { spades, hearts, clubs, diamonds };
  public enum CardValue { two = 2, three = 3, four = 4, five = 5,
                          six = 6, seven = 7, eight = 8, nine = 9,
                          ten = 10, jack = 11, queen = 12, king = 13,
                          ace = 14 };

  private CardSuite suite;
  private CardValue value;

  public Card(CardSuite suite, CardValue value){
    this.suite = suite;
    this.value = value;
  }

  public CardSuite Suite{
    get { return this.suite; }
  }

  public CardValue Value{
    get { return this.value; }
  }

  public override String ToString(){
    return String.Format("Suite:{0}, Value:{1}", suite, value);
  }

  int IGameObject.Value{
    get { return (int)(this.value); }
  }

  public GameObjectMedium Medium{
    get{
      return GameObjectMedium.Paper;
    }
  }
}
```

Program 31.17 *A class Card which implements IGameObject.*

```
using System;

class Client{

  public static void Main(){
    Card cs =
       new Card(Card.CardSuite.spades, Card.CardValue.queen);

    // Use of Value from Card
    Console.WriteLine(cs.Value);

    // Must cast to use the implementation of
    // Value from IGameObject
    Console.WriteLine(((IGameObject)cs).Value);
  }
}
```

Program 31.18 *Sample use of class Card in a Client class.*

```
queen
12
```

Listing 31.19 *Output of Card Client.*

In some situations, an explicit interface implementation can also be used to "hide" an operation that we are forced to implement because the interface requests it. We will meet an example in Section 45.14, where we want to make it difficult to use the `Add` operation on a linked list. Another example is presented in the context of dictionaries in Section 46.3.

## 31.9. References

| | |
|---|---|
| [Hejlsberg06] | Anders Hejlsberg, Scott Wiltamuth and Peter Golde, *The C# Programming Language*. Addison-Wesley, 2006. |
| [Ecma-334] | "The C# Language Specification", June 2005. ECMA-334. |

# 32. Patterns and Techniques

This chapter is the last one in our second lecture about inheritance. The chapter is about patterns and programming techniques related to inheritance. Similar chapters appeared in Chapter 16 and Chapter 24 for classes/objects and for operations respectively.

## 32.1. The Composite design pattern

Lecture 8 - slide 20

The ***Composite*** design pattern, which we are about to study, is probably the most frequently occurring GOF design pattern at all. Most real-life programs that we write benefit from it. Recall from Section 16.2 that the GOF design patterns are the ones described in the original design pattern book [Gamma96].

A ***Composite*** deals with hierarchical structures of objects. In more practical terms, the pattern deals with tree-tructures whose nodes are objects. The main idea behind the pattern is to provide a *uniform interface* to both leaves and inner nodes in the tree.

From a client point of view, it is easy to operate on the nodes of a ***Composite***. The reason is that all participating objects share the interface provided by the abstract *`Component`* class.

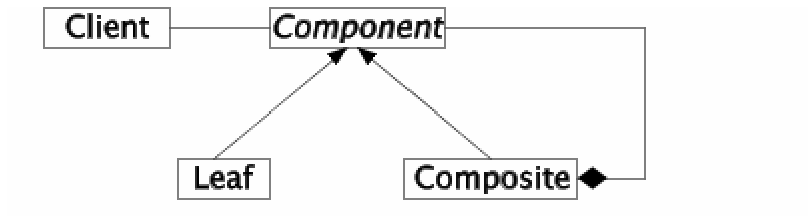

Figure 32.1 *A template of the class structure in the Composite design pattern.*

In Figure 32.1 we show the three classes that - at the principled level - make up a ***Composite:*** The abstract class *`Component`* and its two subclasses `Leaf` and `Composite`. The important things to notice are:

- The diagram in Figure 32.1 is a class diagram, not an object diagram.
- Clients access both `Leaf` nodes and `Composite` nodes (inner nodes in the tree) via the interface provided by the abstract class *`Component`* .
- The `Composite` (inner) node aggregates one or more `Components` , either `Leaf` nodes or (recursively) other `Composite` nodes. This makes up the tree structure. It is important the you are able to grasp the idea that the aggregation in Figure 32.1 gives rise to a recursive tree structure of objects.

In the following sections we will study an example of a composite design pattern which allows us to represent songs of notes and pauses. In appendix Section 58.3 we discuss another example, involving a sequence of numbers and the type `Interval`.

The tree structure may be non-mutable and built via constructors

Alternatively, the tree structure may be mutable, and built via `Add` and `Remove` operations

## 32.2. A Composite Example: Music Elements

Lecture 8 - slide 21

The example in this section stems from the mini project programming (MIP) exam of January 2008 [mip-jan-08]. Imagine that we are interested in a representation of music in terms of notes and pauses. Such a representation can - in a natural way - be described as a ***Composite***, see Figure 32.2. In this composite structure, both a `Note` and a `Pause` are `MusicElements`. A `SequentialMusicElement` consists of a number of `MusicElements`, such as `Notes`, `Pauses`, and other `MusicElements`. The immediate constituents of a `SequentialMusicElement` are played sequentially, one after the other. A `ParallelMusicElement` is composed similar to `SequentialMusicElement`. The immediate constituents of a `ParallelMusicElement` are played at the same time, however.

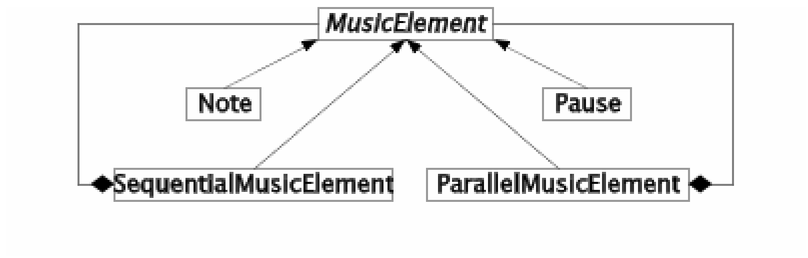

Figure 32.2 *The class diagram of Music Elements*

As we will see in Program 32.3 a `Note` is characterized by a duration, value, volume, and instrument. A `Pause` is characterized by a duration. As such, it may make sense to have a common superclass of `Note` and `Pause`. In the same way, it may be considered to have a common superclass of `SequentialMusicElement` and `ParallelMusicElement` which captures their common aggregation of `MusicElements`.

A number of different operations can be applied uniformly on all `MusicElements`: `Play`, `Transpose`, `TimeStretch`, `NewInstrument`, `Fade`, etc. Below, in Program 32.3 we program the operations `Linearize`, `Duration`, and `Transpose`. The `Linearize` operations transforms a music element to a sequence of lower-level objects which represent MIDI events. A sequence of MIDI events can be played on most computers. In this way, `Linearize` becomes the indirect `Play` operation.

## 32.3. An application of Music Elements

Lecture 8 - slide 22

As we already realized in Section 32.1 the objects in a ***Composite*** are organized in a tree structure. In Figure 32.3 we show an example of a `SequentialMusicElement`. When we play the `SequentialMusicElement` in Figure 32.3 we will first hear note N1. After N1 comes a pause P followed by the notes N2 and N3. Following N3 we will hear N4, N5 and N6 which are all played simultaneously. As such, N4-N6 may form a musical chord. In the web edition of the material we link to a MIDI file of a structure similar to Figure 32.3 [midi-sample].

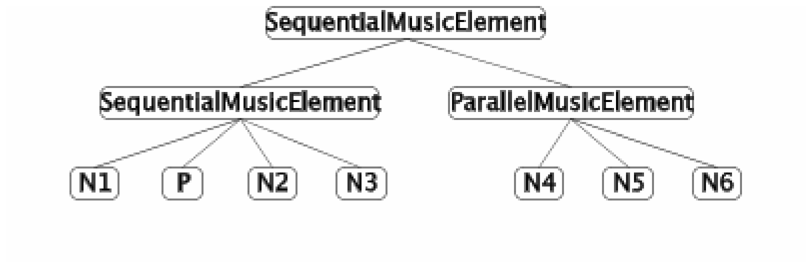

Figure 32.3 *A possible tree of objects which represent various music elements. Nodes named Ni are `Note` instances, and the node named P is a `Pause` instance*

Below, in Program 32.1 we show a program that creates a `SequentialMusicElement` similar to the tree-structure drawn in Figure 32.3 The program relies on the auxiliary class `Song`. The class `Song` and another supporting class `TimedNote` are available to interested readers [song-and-timednote-classes]. Using these two classes it is easy to generate MIDI files from `MusicElement` objects.

```
public class Application{

  public static void Main(){

    MusicElement someMusic =
     SequentialMusicElement.MakeSequentialMusicElement(
       SequentialMusicElement.MakeSequentialMusicElement(
         new Note(60, 480),
         new Pause(480),
         new Note(64, 480),
         new Note(60, 480)),
       ParallelMusicElement.MakeParallelMusicElement(
         new Note(60, 960),
         new Note(64, 960),
         new Note(67, 960)
       ));

    Song aSong = new Song(someMusic.Linearize(0));
    aSong.WriteStandardMidiFile("song.mid");
  }
}
```

Program 32.1 *An application of some MusicElement objects.*

## 32.4. Implementation of MusicElement classes

Lecture 8 - slide 23

In this section we show an implementation of the `MusicElement` classes of Figure 32.2. The classes give rise to non-mutable objects, along the lines of the discussion in Section 12.5.

We start by showing the abstract class `MusicElement`, see Program 32.2. It announces the property `Duration`, the method `Transpose`, and the method `Linearize`. Other of the mentioned music-related operations are not included here. As you probably expect, `Duration` returns the total length of a `MusicElement`. `Transpose` changes the value (the pitch) of a `MusicElement`. `Linearize` transforms a `MusicElement` to an array of (lower-level) `TimeNote` objects [song-and-timednote-classes].

```
public abstract class MusicElement{

  public abstract int Duration{
    get;
  }

  public abstract MusicElement Transpose(int levels);

  public abstract TimedNote[] Linearize(int startTime);
}
```

Program 32.2 *The abstract class MusicElement.*

The class `Note` is shown next, see Program 32.3. `Note` encapsulates the note value, duration, volume, and instrument (see line 5-8). Following two constructors, we see the property `Duration` which simply returns the value of the instance variable `duration`. The method `Linearize` carries out the transformation of the `Note` to a singular array of `TimedNote`. The `Transpose` method adds to the value of the `Note`. The shown activation of `ByteBetween` enforces that the value is between 0 and 127.

```
1  using System;
2
3  public class Note: MusicElement{
4
5    private byte value;
6    private int duration;
7    private byte volume;
8    private Instrument instrument;
9
10   public Note(byte value, int duration, byte volume,
11               Instrument instrument){
12     this.value = value;
13     this.duration = duration;
14     this.volume = volume;
15     this.instrument = instrument;
16   }
17
18   public Note(byte value, int duration):
19    this(value, duration, 64, Instrument.Piano){
20   }
21
22   public override int Duration{
23     get{
24       return duration;
25     }
26   }
27
28   public override TimedNote[] Linearize(int startTime){
29     TimedNote[] result = new TimedNote[1];
30     result[0] = new TimedNote(startTime, value, duration, volume,
31                               instrument);
32     return result;
33   }
34
35   public override MusicElement Transpose(int levels){
36      return new Note(Util.ByteBetween(value + levels, 0, 127),
37                      duration, volume, instrument);
38   }
39 }
```

Program 32.3 *The class Note.*

The class `Pause` shown in Program 32.4 is almost trivial.

```
using System;

public class Pause: MusicElement{

  private int duration;

  public Pause(int duration){
    this.duration = duration;
  }

  public override int Duration{
    get{
      return duration;
    }
  }

  public override TimedNote[] Linearize(int startTime){
    return new TimedNote[0];
  }

  public override MusicElement Transpose(int levels){
     return new Pause(this.Duration);
  }
}
```

Program 32.4 *The class Pause.*

The class `SequentialMusicElement` represents the sequence of `MusicElements` as a list of type `List<T>`. Besides the constructor, `SequentialMusicElement` offers a *factory method* for convenient creation of an instance. Factory methods have been discussed in Section 16.4. Program 32.1 shows how the factory method can be applied. `Duration` adds the duration of the `MusicElement` parts together. Notice that this may cause recursive addition. Likewise, `Transpose` carries out recursive transpositions of the `MusicElement` parts.

```
using System;
using System.Collections.Generic;

public class SequentialMusicElement: MusicElement{
  private List<MusicElement> elements;

  public SequentialMusicElement(MusicElement[] elements){
    this.elements = new List<MusicElement>(elements);
  }

  // Factory method:
  public static MusicElement
    MakeSequentialMusicElement(params MusicElement[] elements){
      return new SequentialMusicElement(elements);
  }

  public override TimedNote[] Linearize(int startTime){
    int time = startTime;
    List<TimedNote> result = new List<TimedNote>();

    foreach(MusicElement me in elements){
      result.AddRange(me.Linearize(time));
      time = time + me.Duration;
    }

    return result.ToArray();
  }

  public override int Duration{
    get{
      int result = 0;

      foreach(MusicElement me in elements){
        result += me.Duration;
      }

      return result;
    }
  }

  public override MusicElement Transpose(int levels){
    List<MusicElement> transposedElements = new List<MusicElement>();

    foreach(MusicElement me in elements)
      transposedElements.Add(me.Transpose(levels));

    return new SequentialMusicElement(transposedElements.ToArray());
  }
}
```

Program 32.5 *The class SequentialMusicElement.*

The class `ParallelMusicElement` resembles `SequentialMusicElement` a lot. Notice, however, the different implementation of `Duration` in line 29-39.

```
1  using System;
2  using System.Collections.Generic;
3  
4  public class ParallelMusicElement: MusicElement{
5    private List<MusicElement> elements;
6  
7    public ParallelMusicElement(MusicElement[] elements){
8      this.elements = new List<MusicElement>(elements);
9    }
10 
11   // Factory method:
12   public static MusicElement
13     MakeParallelMusicElement(params MusicElement[] elements){
14       return new ParallelMusicElement(elements);
15   }
16 
17   public override TimedNote[] Linearize(int startTime){
18     int time = startTime;
19     List<TimedNote> result = new List<TimedNote>();
20 
21     foreach(MusicElement me in elements){
22       result.AddRange(me.Linearize(time));
23       time = startTime;
24     }
25 
26     return result.ToArray();
27   }
28 
29   public override int Duration{
30     get{
31       int result = 0;
32 
33       foreach(MusicElement me in elements){
34         result = Math.Max(result, me.Duration);
35       }
36 
37       return result;
38     }
```

```
  }

  public override MusicElement Transpose(int levels){
    List<MusicElement> transposedElements = new List<MusicElement>();

    foreach(MusicElement me in elements)
      transposedElements.Add(me.Transpose(levels));

    return new ParallelMusicElement(transposedElements.ToArray());
  }
}
```

Program 32.6 *The class ParallelMusicElement.*

This completes our discussion of the `MusicElement` composite. The important things to pick up from the example are:

1. The tree structure of objects defined by the subclasses of `MusicElement` .
2. The uniform interface of music-related operations provided to clients of `MusicElement` .

As stressed in Section 32.1 these are the primary merits of ***Composite***.

In Section 58.3 of the appendix we present an additional and similar example of a composite which involves an `Interval`. `Interval` is the type we encountered in Section 21.3 when we discussed operator overloading.

## 32.5. A Composite Example: A GUI

Lecture 8 - slide 27

We will study yet another example of a ***Composite*** design pattern. A graphical user interface (GUI) is composed of a number of *forms*, such as buttons and textboxes. The classes behind these forms make up a ***Composite*** design pattern.

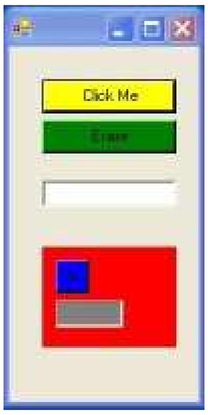

Figure 32.4 *A Form (Window) with two buttons, a textbox, and a panel.*

We construct the simple GUI depicted in Figure 32.4. The actual hierarchy of objects involved are shown in Figure 32.5. Thus, the GUI is composed of three buttons (yellow, green, and blue) and two textboxes (white and grey). The blue button and the grey textbox are aggregated into a so-called panel (which has red background in Figure 32.4).

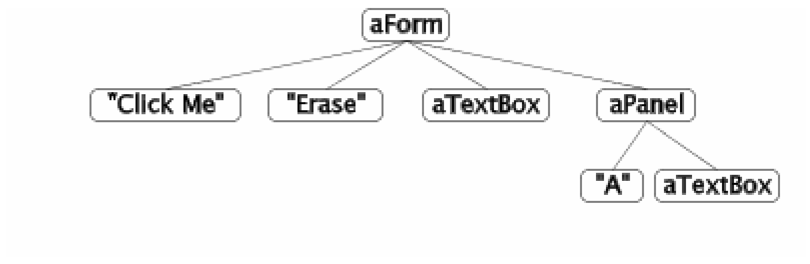

Figure 32.5 *The tree of objects behind the graphical user interface. These objects represents a composite design pattern in an executing program.*

The Form class hierarchy of .NET and C# is very large. A small extract is shown in Figure 32.6. Apart from class `Component`, all classes are from the namespace `System.Windows.Forms`.

There are two ***Composites*** in Figure 32.6. The first one is (object) rooted by class `Form`, which may aggregate an arbitrary number of Windows form objects. The class `Form` represents a window. The class `Control` is the superclass of GUI elements that displays information on the screen. There are approximate 25 immediate and direct subclasses of class `Control`. In reality the classes `TextBox`, `Button`, and `Panel` are all indirect subclasses of `Control`.

The other Composite is, symmetrically, (object) rooted by `Panel`, which like `Form` may aggregate an arbitrary number of `Form` objects. Class `Pane` is intended for grouping of a collection of controls.

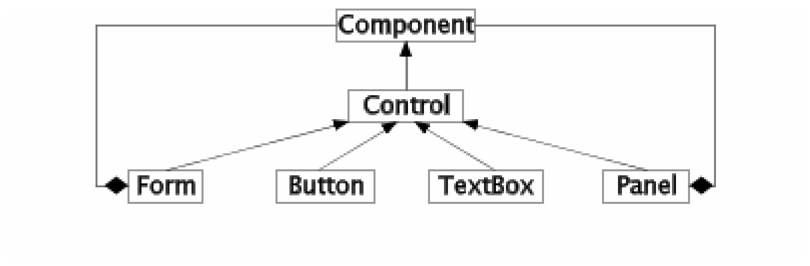

Figure 32.6 *An extract of the Windows Form classes for GUI building. We see two Composites among these classes.*

Below, in Figure 32.6 we show how to construct the form object tree shown in Figure 32.5, which gives rise to the GUI of Figure 32.4. We program a class which we name `Window`. Our `Window` class inherits from class `Form`. Thus, our `Window` ***is a*** `Form`. Shown in **blue** we highlight instantiation of GUI elements. Shown in **purple** we highlight the actual construction of the tree structure of Figure 32.5. The `Controls` property of a Form, referred in line 60 - 67, give access to a collection of controls, of type `ControlCollection`.

As it appears in line 23 and 31, we also add a couple of event handlers, programmed as private methods from line 70 - 83. We have discussed event handlers in Chapter 23. The associated event handlers just acknowledge when we click on of the three buttons of the GUI.

```
1  using System;
2  using System.Windows.Forms;
3  using System.Drawing;
4
5  // In System:
6  // public delegate void EventHandler (Object sender, EventArgs e)
7
8  public class Window: Form{
9
10   Button b1, b2, paBt;
11   Panel pa;
```

```
TextBox tb, paTb;

// Constructor
public Window (){
  this.Size=new Size(150,300);

  b1 = new Button();
  b1.Text="Click Me";
  b1.Size=new Size(100,25);
  b1.Location = new Point(25,25);
  b1.BackColor = Color.Yellow;
  b1.Click += ClickHandler;
                          // Alternatively:
                          // b1.Click+=new EventHandler(ClickHandler);
  b2 = new Button();
  b2.Text="Erase";
  b2.Size=new Size(100,25);
  b2.Location = new Point(25,55);
  b2.BackColor=Color.Green;
  b2.Click += EraseHandler;
                          // Alternatively:
                          // b2.Click+=new EventHandler(EraseHandler);
  tb = new TextBox();
  tb.Location = new Point(25,100);
  tb.Size=new Size(100,25);
  tb.BackColor=Color.White;
  tb.ReadOnly=true;
  tb.RightToLeft=RightToLeft.Yes;

  pa = new Panel();
  pa.Location = new Point(25,150);
  pa.Size=new Size(100, 75);
  pa.BackColor=Color.Red;

  paBt = new Button();
  paBt.Text="A";
  paBt.Location = new Point(10,10);
  paBt.Size=new Size(25,25);
  paBt.BackColor=Color.Blue;
  paBt.Click += PanelButtonClickHandler;

  paTb = new TextBox();
  paTb.Location = new Point(10,40);
  paTb.Size=new Size(50,25);
  paTb.BackColor=Color.Gray;
  paTb.ReadOnly=true;
  paTb.RightToLeft=RightToLeft.Yes;

  this.Controls.Add(b1);
  this.Controls.Add(b2);
  this.Controls.Add(tb);

  pa.Controls.Add(paBt);
  pa.Controls.Add(paTb);

  this.Controls.Add(pa);
}

// Eventhandler:
private void ClickHandler(object obj, EventArgs ea) {
  tb.Text = "You clicked me";
}

// Eventhandler:
private void PanelButtonClickHandler(object obj, EventArgs ea) {
    paTb.Text += "A";
  }

  // Eventhandler:
  private void EraseHandler(object obj, EventArgs ea) {
    tb.Text = "";
  }

}

class ButtonTest{

  public static void Main(){
    Window win = new Window();
    Application.Run(win);
  }

}
```

Program 32.7 *A program that builds a sample composite graphical user interface.*

## 32.6. Cloning

Lecture 8 - slide 30

We briefly discussed copying of objects in Section 13.4 of the lecture about classes and objects. In this section we will continue this discussion. First we will distinguish between different types of object copying. Later, in Section 32.7, we will see how to enable the pre-existing `MemberwiseClone` operation to client classes.

Instead of the word "copy" we often use the word "clone":

*Cloning* creates a copy of an existing object

There are different kinds of cloning, distinguished by the copying depth:

- **Shallow cloning**:
  - Instance variables of value type: Copied bit-by-bit
  - Instance variables of reference types:
    - The reference is copied
    - The object pointed at by the reference is **not** copied
- **Deep cloning**:
  - Like shallow cloning
  - But objects referred by references are copied recursively

Shallow cloning is the variant supported by the `MemberwiseClone` operation in Section 32.7. Only a single object is copied.

Deep cloning copies a network of objects, and it may, in general, involve many objects.

Recall that cloning is only relevant for instances of classes, for which reference semantics apply (see Chapter 13). Values of structs obey value semantics, and as such struct values are (shallow) copied by assignments and by parameter passing. See Chapter 14 for additional details.

## 32.7. Cloning in C#

Lecture 8 - slide 31

Shallow cloning is supported "internally" by any object in a C# program execution. The reason is that any object inherit from class `Object` in which the protected method `MemberwiseClone` implements shallow cloning. (See Section 28.3 for an overview of the methods in class `Object` ). Recall from Section 27.3 that a protected method of a class C is visible in C and in the subclasses of c, but not in clients of C.

In this section we will see how we can unleash the protected `MemberwiseClone` operation as a public operation of an arbitrary class.

Below, in Program 32.8 we show how to implement a cloneable `Point` class. First, notice that `Point` implements the interface `ICloneable`, which prescribes a single method called `Clone`. We have already in Section 31.4 seen `ICloneable` in the context of other flavoring interfaces from the C# libraries. The public method `Clone` of class `Point`, shown in **purple**, delegates its work to the protected method `MemberwiseClone`. In other words, our `Clone` methods send a `MemberwiseClone` message to the current `Point` object. `MemberWiseClone` makes the bitwise, shallow copy of the point, and it returns it. Notice that from a static point of view, the returned object is of type `Object`. As we will see below, this will typically imply a need to cast the returned object to a `Point`.

Although a `Clone` method typically delegates its work to `MemberwiseClone`, it is not necessary to do so. `Clone` may, alternatively, use a constructor and appropriate object mutations in order to produce the copy, which makes sense for the class in question (which is class `Point` in the example shown below).

```
using System;

public class Point: ICloneable {
  private double x, y;

  public Point(double x, double y){
   this.x = x; this.y = y;
  }

  public double X {
    get {return x;}
    set {x = value;}
  }

  public double Y {
    get {return y;}
    set {y = value;}
  }

  public Point move(double dx, double dy){
    Point result = (Point)MemberwiseClone();  // cloning from within Point is OK.
    result.x = x + dx;
    result.y = y + dy;
    return result;
  }

  // public Clone method that delegates the work of
  // the protected method MemberwiseClone();
  public Object Clone(){
    return MemberwiseClone();
  }

  public override string ToString(){
    return "Point: " + "(" + x + "," + y + ")" + ".";
  }
}
```

Program 32.8    *A cloneable class Point.*

In Program 32.9 we show how a client of class `Point` uses the implemented `Clone` operation. Notice the casting, and notice that the subexpression `p1.Clone()` is evaluated before the casting. (A possible misconception would be that `(Point)p1` is evaluated first). The evaluation order is due to the precedence of the cast operator in relation to the precedence of the dot operator, see Table 6.1.

```
using System;

public class Application{

  public static void Main(){
    Point p1 = new Point(1.1, 2.2),
          p2, p3;

    p2 = (Point)p1.Clone();  // First p1.Clone(), then cast to Point.
    p3 = p1.move(3.3, 4.4);
    Console.WriteLine("{0} {1} {2}", p1, p2, p3);
  }

}
```

Program 32.9    *A sample client of class Point.*

It may be tempting to circumvent the `ICloneable` interface, the implementation of our own clone operation, and delegation to `MemberwiseClone`. This temptation is illustrated in Program 32.10. The compiler will find out, and it tells that we cannot just call `MemberwiseClone`, because it is not a public operation.

Why make life so difficult? Why not support shallow copying of all objects in an easy way, by making `MemberwiseClone` public in class `Object`? The reason is that the designers of the programming language (C#, and Java as well) have decided that the programmer of a class should make an explicit decision about which objects should be cloneable.

There are almost certainly some classes for which we do not want copying of instances, singletons (see Section 16.3 ) for instance. There are also some classes in which we do not want the standard bitwise copying provided by `MemberwiseClone`. Such classes should behave like Program 32.8 shown above, but instead of delegating the cloning to `MemberwiseClone`, the copy operation should be programmed in the `Clone` method to suit the desired copying semantics.

```
using System;

public class Application{

  public static void Main(){
    Point p1 = new Point(1.1, 2.2),
          p2, p3;

    p2 = (Point)p1.MemberwiseClone();
    // Compile-time error.
    // Cannot access protected member 'object.MemberwiseClone()'
    // via a qualifier of type 'Point'

    p3 = p1.move(3.3, 4.4);
    Console.WriteLine("{0} {1} {2}", p1, p2, p3);
  }

}
```

Program 32.10    *Illegal cloning with MemberwiseClone.*

## 32.8. Cloning versus use of copy constructors

Lecture 8 - slide 32

In Section 32.7 we found out that cloning of class instances - on purpose - is rather cumbersome in C#. Therefore we have earlier recommended the use of *copy constructors* as an alternative means. See Section 12.5 for details and for an example.

In this section we will evaluate and exemplify the power of copying by cloning (as in Section 32.7) relative to copying by use of copy constructors.

In a nutshell, the insight can be summarized in this way:

Cloning with **`obj.Clone()`** is more powerful than use of copy constructors, because **`obj.Clone()`** may exploit polymorphism and dynamic binding

In order to illustrate the differences between cloning (by use of the `clone` method) and copying (by use of a copy constructor) we will again use the class `Point`. Below, in Section 32.7 we show a version similar to Program 32.8 but now with an additional copy constructor (line 12 - 14).

```
1  using System;
2  using System.Drawing;
3
4  public class Point: ICloneable {
5    protected double x, y;
6
7    public Point(double x, double y){
8     this.x = x; this.y = y;
9    }
10
11   // Copy constructor
12   public Point(Point p){
13    this.x = p.x; this.y = p.y;
14   }
15
16   public virtual double X {
17     get {return x;}
18     set {x = value;}
19   }
20
21   public virtual double Y {
22     get {return y;}
23     set {y = value;}
24   }
25
26   public virtual Point move(double dx, double dy){
27     Point result = (Point)MemberwiseClone();  // cloning from within Point is OK.
28     result.x = x + dx;
29     result.y = y + dy;
30     return result;
31   }
32
33   // public Clone method that delegates the work of
34   // the protected method MemberwiseClone();
35   public virtual Object Clone(){
36     return MemberwiseClone();
37   }
38
39   public override string ToString(){
40     return "Point: " + "(" + x + "," + y + ")" + ".";
41   }
42 }
```

Program 32.11 *A cloneable class Point.*

We also show a subclass of `Point` called `ColorPoint`, see Program 32.12. It adds a `color` instance variable to the instance variables inherited from class `Point`, and it has its own copy constructor in line 10 - 13.

```
1  public class ColorPoint: Point{
2    protected Color color;
3
4    public ColorPoint(double x, double y, Color c):
5        base(x,y){
6      this.color = c;
7    }
8
9    // Copy constructor
10   public ColorPoint(ColorPoint cp):
11       base(cp.x,cp.y){
12     this.color = cp.color;
13   }
14
15   // Clone method is inherited
16
17   public override string ToString(){
18     return "ColorPoint: " + "(" + x + "," + y + ")" + ":" + color;
19   }
20 }
```

Program 32.12 *A cloneable class ColorPoint.*

In the `Color` and `ColorPoint` client program, shown below in Program 32.13, you should focus on the list `pointList`, as declared in line 14. We add two `Point` objects and two `ColorPoint` objects to `pointList` in line 17 - 20. Next, in the foreach loop starting at line 23 we clone each of the four points in the list, and we add the cloned points to the initially empty list `clonedPointList`. The elements in `clonedPointList` are printed at the end of the program. The output - see Listing 32.14 - reveals that the cloning is successful. We end up having two `Point` instances and two `ColorPoint` instances in `clonedPointList`.

```
1  using System;
2  using System.Drawing;
3  using System.Collections.Generic;
4
5  public class Application{
6
```

```
  public static void Main(){
    Point p1 =       new Point(1.1, 2.2),
          p2 =       new Point(3.3, 4.4);

    ColorPoint cp1 = new ColorPoint(5.5, 6.6, Color.Red),
               cp2 = new ColorPoint(7.7, 8.8, Color.Blue);

    List<Point> pointList = new List<Point>(),
                clonedPointList = new List<Point>();

    pointList.Add(p1);
    pointList.Add(cp1);
    pointList.Add(p2);
    pointList.Add(cp2);

    // Clone the points in pointList and add them to clonedPointList:
    foreach(Point p in pointList){
      clonedPointList.Add((Point)(p.Clone()));
    }

    foreach(Point p in clonedPointList)
      Console.WriteLine("{0}", p);

  }
}
```

Program 32.13 *Polymorphic Cloning of Points.*

```
Point: (1,1,2,2).
ColorPoint: (5,5,6,6):Color [Red]
Point: (3,3,4,4).
ColorPoint: (7,7,8,8):Color [Blue]
```

Listing 32.14 *Output of both polymorphic and non-polymorphic cloning.*

Let us now attempt to replicate the functionality of Program 32.13 with use of copy constructors, see Program 32.15. The attempt, shown in Program 32.15 in line 24 - 26 fails, because the activation of the copy constructors deliver `Point` objects, even if a `ColorPoint` object is passed as input. Instead we must perform explicit type dispatch, as shown in line 29-34. Clearly, constructors cannot exhibit virtual behavior.

The solution in Program 32.13 based on the `Point` and `ColorPoint` classes in Program 32.11 and Program 32.12 works because the `Clone` method in Program 32.11 (line 35 - 37) is inherited by `ColorPoint`. As already explained, the inherited method delegates its work to `MemberwiseClone`, which always copies its receiver. Thus, if `MemberwiseClone` is activated on a `ColorPoint` object it copies a `ColorPoint` object.

```
1  using System;
2  using System.Drawing;
3  using System.Collections.Generic;
4
5  public class Application{
6
7    public static void Main(){
8      Point p1 =       new Point(1.1, 2.2),
9            p2 =       new Point(3.3, 4.4);
10
11     ColorPoint cp1 = new ColorPoint(5.5, 6.6, Color.Red),
12                cp2 = new ColorPoint(7.7, 8.8, Color.Blue);
13
14     List<Point> pointList = new List<Point>(),
15                 clonedPointList = new List<Point>();
16
```

```
    pointList.Add(p1);
    pointList.Add(cp1);
    pointList.Add(p2);
    pointList.Add(cp2);

//  Cannot copy ColorPoint objects with copy constructor of Point.
//  Compiles and runs, but gives wrong result.
//  foreach(Point p in pointList){
//    clonedPointList.Add(new Point(p));
//  }

    // Explicit type dispatch:
    foreach(Point p in pointList){
      if (p is ColorPoint)
        clonedPointList.Add(new ColorPoint((ColorPoint)p));
      else if (p is Point)
        clonedPointList.Add(new Point(p));
    }

    foreach(Point p in clonedPointList)
      Console.WriteLine("{0}", p);

  }
}
```

Program 32.15 *Non-polymorphic Cloning of Points - with use of copy constructors.*

## 32.9. The fragile base class problem

Lecture 8 - slide 33

As the next part of this Pattern and Techniques chapter we will study the so-called fragile base class problem.

The problem can be summarized in this way:

> If all methods are virtual it is possible to introduce erroneous dynamic bindings
>
> This can happen if a new method in a superclass is given the same name as a dangerous method in a subclass

The program in Program 32.16 is a principled ABC example. `B` is a subclass of `A`, and `B` has a dangerous method `M2`. As a dangerous method, clients of class `B` must be fully aware that `M2` is called, because it can have serious consequences. (A possible serious consequence may be to erase the entire harddisk). As illustrated in the `Client` class, `M2` can only be activated on a `B` object.

```
// Original program. No problems.

using System;

class A {

  public void M1(){
    Console.WriteLine("Method 1");
  }
}

```

```
class B: A {

  public void M2(){
    Console.WriteLine("Dangerous Method 2");
  }
}

class Client{

  public static void Main(){
    A a = new B();
    B b = new B();

    a.M1();  // Nothing dangerous expected
//  a.M2();  // Compile-time error
             // 'A' does not contain a definition for 'M2'
    b.M2();  // Expects dangerous operation
  }
}
```

Program 32.16    *The initial program.*

Let us now assume that we replace class `A` with a new version with the following changes:

1. A new virtual method `M2` is added to A.
2. The existing method `M1` in `A` now calls `M2`

This is shown in Program 32.17.

We will, in addition, assume that all involved methods (`M1` and `M2`) are virtual, and that `M2` in `B` overrides `M2` in `A`. In C# this is not a natural assumption, but in Java this is the default semantics (and the only possible semantics).

It is purely accidental that the new method in class `A` has the same name as the dangerous method `M2` in class `B`.

In the `Client` class in Program 32.17 `a.M1()` will - unexpectedly - call the dangerous method `M2` in class `B`, because the variable `a` has dynamic type `B`. Similarly, `a.M2()` calls `M2` in `B`. The programmer, who wrote class `A`, expected that the expression `a.M1()` would call the sibling method `M2` in class `A`! This could come as an unpleasant surprise.

```
// Dangerous program.
// M2 is virtual in A and overridden in B.
// Compiles and runs
// Default Java semantics.

using System;

// New version of A
class A {

  public void M1(){
    Console.WriteLine("Method 1");
    this.M2();
  }

  // New method in this version.
  // Same name as the dangerous operation in subclass B
  public  virtual  void M2(){
    Console.WriteLine("M2 in new version of A");
  }

}

class B: A {

  public override void M2(){
    Console.WriteLine("Dangerous Method 2");
  }
}

class Client{

  public static void Main(){
    A a = new B();
    B b = new B();

    a.M1();  // Nothing dangerous expected
             // Will, however, call the dangerous operation
             // because M2 is virtual.

    a.M2();  // Makes sense when M2 exists in class A.
             // Calls the dangerous method

    b.M2();  // Calls the dangerous method.
             // This is expected, however.
  }
}
```

Program 32.17 *The revised version with method A.M2 being virtual.*

```
Method 1
Dangerous Method 2
Dangerous Method 2
Dangerous Method 2
```

Listing 32.18 *Output of revised program.*

If we, in C#, just add the `M2` method to class `A`, and change `M1` such that it calls `M2`, as shown in Program 32.19 (only on web) it is not possible to compile class `B`. The problem is that we have a method, named `M2` in both class `A` and `B`. This is the problem that we have discussed in Section 28.9. The programmer should decide if `M2` in `B` should be declared as **`new`**, or if it should **`override`** `M2` from class `A`. In the latter case, `M2` in `A` must be declared as virtual.

If you want additional details about the fragile base class problem, the web-version of the paper contains two additional variants of Program 32.17.

The fragile base class problem illustrates a danger of using virtual methods (dynamic binding) all over the place. In rare situations, as the one constructed in Program 32.17, it may lead to dangerous results. To summarize, the problem arises if a method in a subclass is unintentionally called instead of a method in a superclass. In C#, both the superclass and the subclass must specify if dynamic binding should take place. In the superclass the involved method must be **`virtual`**, and in the subclass the method must use the **`override`** modifier. Alternatively, we may opt for static binding, as in Program 32.20. As illustrated by Program 32.19 the C# programmer is likely to get a warning in case he or she approaches the fragile base class problem.

## 32.10. Factory design patterns

Lecture 8 - slide 34

Instantiation of classes is done by the `new` operator (see Section 6.7) in cooperation with a constructor (see Section 12.4). Imagine that we need numerous instances of class `C` in a large program. This would lead to a situation where there appears may expressions of the form `new C(...)` in the program. Why can this be considered as a problem?

One problem with many occurrences of `new C(...)` is if we - eventually - would like to instantiate another class, say a subclass of `C`. In this situation we would prefer to make one change at a single place in the program, instead of a spread of changes throughout the program.

Another problem may occur if we need two or more constructors which we cannot easily distinguish by the formal parameters of the constructors. We have seen examples of such situations in Section 16.4.

As yet another problem, we may wish to introduce good names for object instantiations, beyond the possibilities of constructors.

Various uses of factory methods can be seen as solutions to the problems pointed out above. We will distinguish between the following variations of factory methods:

- Factory methods implemented with class methods (static methods) in C, or in another class
- The design pattern ***Factory Method*** which handles instantiation in instance methods of client subclasses
  - Relies on instance methods in class hierarchies with virtual methods
- The design pattern ***Abstract Factory*** which is good for instantiation of product families
  - Relies on instance methods in class hierarchies with virtual methods

As already pointed out, we have seen examples of static factory methods in Section 16.4. We will discuss the design pattern ***Factory Method*** below, in Section 32.11. In the current version of the material we do not discuss ***Abstract Factory***.

## 32.11. The design pattern Factory Method

Lecture 8 - slide 35

The ***Factory Method*** design pattern relies on virtual instance methods in a class hierarchy that take care of class instantiation. The ***Factory Method*** scene is shown diagrammatically in Figure 32.7 and programmatically in Program 32.22.

The problem is how to facilitate instantiation of different types of `Products` (line 3-13 in Program 32.22) in `SomeOperation` (line 20) of class `Creator`.

The Factory Method solution is to carry out the instantiation in overridden virtual methods in subclasses of class `Creator`. The actual instantiations take place in line 26 and 32 of Program 32.22. In `SomeOperation`, the highlighted call **`this.FactoryMethod()`** will either cause instantiation of `ConcreteProduct_1` or `ConcreteProduct_2`, depending on the dynamic type of the creator.

Subclasses of `Creator` decide which `Product` to instantiate

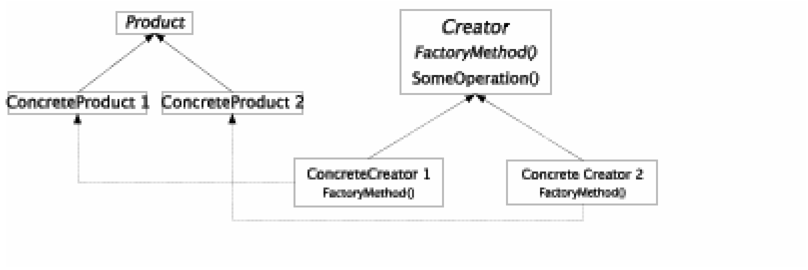

Figure 32.7 *A template of the class structure in the Factory Method design pattern.*

```
using System;

public abstract class Product{
  public Product(){}
}

public class ConcreteProduct_1: Product{
  public ConcreteProduct_1(){}
}

public class ConcreteProduct_2: Product{
  public ConcreteProduct_2(){}
}


public abstract class Creator{
  public abstract Product FactoryMethod();

  public void SomeOperation(){
    Product product =  FactoryMethod();
  }
}

public class ConcreteCreator_1: Creator{
  public override Product FactoryMethod(){
    return new ConcreteProduct_1();
  }
}

public class ConcreteCreator_2: Creator{
  public override Product FactoryMethod(){
    return new ConcreteProduct_2();
  }
}
```

Program 32.22 *Illustration of Factory Method in C#.*

***Factory Method*** calls for a quite complicated scene of parallel class hierarchies. The key mechanism behind the pattern is the activation of a virtual method from a fully defined, non-abstract method in the (abstract) class `Creator`. In many contexts, ***Factory Method*** will be too complicated to set up. If, however, the major parts of the class hierarchies already have been established, the use of ***Factory Method*** allows for flexible variations of `Product` instantiations.

## 32.12. The Visitor design pattern

Lecture 8 - slide 37

The ***Visitor*** design pattern is typically connected to the ***Composite*** design pattern, which we have discussed in Section 32.1. Recall that a ***Composite*** gives rise to a tree of objects, all of which expose a uniform client interface. The ***Visitor*** design pattern is about a particular organization of the operations that visit each node in such a tree.

Relative to the ***Composite*** class diagram, shown in Figure 32.1, we will discuss two different organizations of the tree visiting operations:

- The natural object-oriented solution:
    - One method per operation per *`Component`* class
- The ***Visitor*** solution
    - All methods that pertain to a given operation are refactored and encapsulated in its own class

The natural object-oriented solution, mentioned first, is the solution that falls out of the ***Composite*** design pattern. We will illustrate it in the context of the `IntSequence` ***Composite*** in Section 32.13.

The ***Visitor*** solution is an alternative - and more complicated - organization which keeps all operations of a given traversal together. This is the solution of the ***Visitor*** design pattern. It will be exemplified in Section 32.15.

## 32.13. Natural object-oriented IntSequence traversals

Lecture 8 - slide 38

We have studied the integer sequence composite in appendix Section 58.3. The class diagram of this particular ***Composite*** is shown in Figure 58.1. Please recapitulate the essence of the integer sequence idea from there.

We will now discuss three different operations which need to visit each object in a integer sequence tree, such as the seven nodes of the tree shown in Figure 58.2. The operations are `Max`, `Min`, and `Sum`. `Min` and `Max` find the smallest/largest number in the tree respectively. `Sum` adds all numbers in the tree together.

Below we show the `Min`, `Max`, and `Sum` operations in four classes `IntSequence`, `IntSingular`, `IntInterval`, and `IntComposite`. All of the operation need to traverse the tree structure. Inner nodes in the composite tree are represented as instances of the class `IntCompSeq`, as shown in Program 32.26. The operations `Min`, `Max`, and `Sum` are implemented recursively in this class. A ***Composite*** is a recursive data structure which in a natural way calls for recursive processing. All this is archetypical for a composite structure.

```
public abstract class IntSequence {
  public abstract int Min {get;}
  public abstract int Max {get;}
  public abstract int Sum();
}
```

Program 32.23 *The abstract class IntSequence.*

```
public class IntInterval: IntSequence{

  private int from, to;

  public IntInterval(int from, int to){
    this.from = from;
    this.to = to;
  }

  public int From{
    get{return from;}
  }

  public int To{
    get{return to;}
  }

  public override int Min{
    get {return Math.Min(from,to);}
  }

  public override int Max{
    get {return Math.Max(from,to);}
  }

  public override int Sum(){
    int res = 0;
    int lower = Math.Min(from,to),
        upper = Math.Max(from,to);

    for (int i = lower; i <= upper; i++)
       res += i;
    return res;
  }
}
```

Program 32.24 *The class IntInterval.*

```
public class IntSingular: IntSequence{

  private int it;

  public IntSingular(int it){
    this.it = it;
  }

  public int TheInt{
     get{return it;}
  }

  public override int Min{
    get {return it;}
  }

  public override int Max{
    get {return it;}
  }

  public override int Sum(){
    return it;
  }
}
```

Program 32.25 *The class IntSingular.*

```
public class IntCompSeq: IntSequence{

  private IntSequence s1, s2;  // Binary sequence: Exactly two subsequences.

  public IntCompSeq(IntSequence s1, IntSequence s2) {
    this.s1 = s1;
    this.s2 = s2;
  }

  public IntSequence First{
    get{return s1;}
  }

  public IntSequence Second{
    get{return s2;}
  }

  public override int Min{
    get {return Math.Min(s1.Min, s2.Min);}
  }

  public override int Max{
    get {return Math.Max(s1.Max, s2.Max);}
  }

  public override int Sum(){
    return s1.Sum() + s2.Sum();
  }
}
```

Program 32.26 *The class IntCompSeq.*

In the web version of the material we show an integer sequence client which traverses a composite tree structure of integer sequences with use of the operations `Min`, `Max`, and `Sum`, see Program 32.27 (only on web). In Listing 32.28 (only on web) we also show the program output.

The programming of `Min`, `Max`, and `Sum` in the integer sequence classes, as shown above, is natural object-oriented programming of the traversals. Each of the four involved classes has a `Min`, `Max`, and a `Sum` operation. The operations are located in the immediate neighborhood of the data on which they rely. Everything is fine.

But the solution shown in this section is not a ***Visitor***. In the next section we will discuss and motivate the transition from the natural object-oriented solution to a visitor. After that we will reorganize `Min`, `Max` and `Sum` as visitors.

## 32.14. Towards a Visitor solution

Lecture 8 - slide 39

Before we study Visitors for integer sequence traversals we will discuss the transition from the natural object-oriented traversal to the ***Visitor*** design pattern.

The main idea of ***Visitor*** is to organize all methods that take part in a particular traversal, in a single `Visitor` class. In our example from Section 32.13 it means that we will have `MinVisitor`, `MaxVisitor`, and `SumVisitor` class. All of these classes share a common `Visitor` interface.

The following steps are involved the transition from natural object-oriented visiting to the ***Visitor*** design pattern:

- A `Visitor` interface and three concrete Visitor classes are defined
- The `Intsequence` classes are refactored - the traversal methods are moved to the visitor classes
- `Accept` methods are defined in the `IntSequence` classes. `Accept` takes a `Visitor` as parameter
- `Accept` passes `this` to the visitor, which in turn may activate `Accept` on components

From the web-version of the material we provide an SVG-animation that illustrates the transition.

Try the accompanying SVG animation

In the following section we will study an example. In the slipstream of the example we will explain and discuss additional details. The pros and cons of the ***Visitor*** solution are summarized in Section 32.16.

## 32.15. A Visitor example: IntSequence

Lecture 8 - slide 40

Let us now reorganize the integer sequence traversals from Section 32.13 to a ***Visitor***.

We provide three different traversals: find minimum, find maximum, and calculate sum. This will give rise to three different visitor objects: a minimum visitor, a maximum visitor, and a sum visitor of types `MinVisitor`, `MaxVisitor`, and `SumVisitor` respectively. The three classes implement a common `Visitor` interface. Each of the visitors will have `VisitIntInterval`, `VisitSingular`, and `VisitCompSeq` methods. As a naming issue, we chose to use the name `Visit` for all of them. This choice relies on method overloading. With these considerations we are able to understand the `Visitor` interface shown in Program 32.29.

```
public interface Visitor{
  int Visit (IntInterval iv);
  int Visit (IntSingular iv);
  int Visit (IntCompSeq iv);
}
```

Program 32.29 *The Visitor Interface.*

The abstract superclass in the integer sequence ***Composite*** design pattern, which we presented in Program 32.23, can now be reduced to a single method, which takes a `Visitor` object as parameter. The method is usually called `Accept`.

```
public abstract class IntSequence {
  public abstract int Accept(Visitor v);
}
```

Program 32.30 *The abstract class IntSequence.*

The idea behind the `Accept` method is to delegate the responsibility of a particular traversal to a given `Visitor` object. In the class `IntInterval`, shown below in Program 32.31, we see that `Accept` passes the current object (the `IntInterval` object) to the visitor. This is done in line 19. The same happens in `Accept` of `IntSingular` (line 14 of Program 32.32) and in `Accept` of `IntCompSeq` (line 19 of Program 32.33).

```
1  public class IntInterval: IntSequence{
2  
3    private int from, to;
4  
5    public IntInterval(int from, int to){
6      this.from = from;
7      this.to = to;
8    }
9  
10   public int From{
11     get{return from;}
12   }
13 
14   public int To{
15     get{return to;}
16   }
17 
18   public override int Accept(Visitor v){
19     return v.Visit(this);
20   }
21 }
```

Program 32.31 *The class IntInterval.*

```
1  public class IntSingular: IntSequence{
2  
3    private int it;
4  
5    public IntSingular(int it){
6      this.it = it;
7    }
8  
9    public int TheInt{
10      get{return it;}
11   }
12 
13   public override int Accept(Visitor v){
14     return v.Visit(this);
15   }
16 }
```

Program 32.32 *The class IntSingular.*

```
public class IntCompSeq: IntSequence{

  private IntSequence s1, s2;  // Binary sequence: Exactly two subsequences.

  public IntCompSeq(IntSequence s1, IntSequence s2) {
    this.s1 = s1;
    this.s2 = s2;
  }

  public IntSequence First{
    get{return s1;}
  }

  public IntSequence Second{
    get{return s2;}
  }

  public override int Accept(Visitor v){
    return v.Visit(this);
  }
}
```

Program 32.33 *The class IntCompSeq.*

It is now time to program the visitor classes (the classes that implement the `Visitor` interface of Program 32.29).

The `Visit` methods on intervals and singulars (the leafs in the composite tree) just extract information from the passed tree node. Thus, the `Visit` methods extract information from the objects that hold the essential information (this is the objects that provide the `Accept` methods). The `Visit` methods on the inner tree nodes (of type `IntCompSeq`) are more interesting. They call `Accept` methods on subtrees of the inner tree node. This is highlighted with blue color in Program 32.34, Program 32.35, and Program 32.36.

```
public class MinVisitor: Visitor{
  public int Visit (IntInterval iv){
    return Math.Min(iv.From, iv.To);
  }

  public int Visit (IntSingular iv){
    return iv.TheInt;
  }

  public int Visit (IntCompSeq iv){
    return Math.Min(iv.First.Accept(this),
                    iv.Second.Accept(this));
  }
}
```

Program 32.34 *The class MinVisitor.*

```
public class MaxVisitor: Visitor{
  public int Visit (IntInterval iv){
    return Math.Max(iv.From, iv.To);
  }

  public int Visit (IntSingular iv){
    return iv.TheInt;
  }

  public int Visit (IntCompSeq iv){
    return Math.Max(iv.First.Accept(this),
                    iv.Second.Accept(this));
  }
}
```

Program 32.35 *The class MaxVisitor.*

```
public class SumVisitor: Visitor{
  public int Visit (IntInterval iv){
    int res = 0;
    int lower = Math.Min(iv.From,iv.To),
        upper = Math.Max(iv.From,iv.To);

    for (int i = lower; i <= upper; i++)
       res += i;
    return res;
  }

  public int Visit (IntSingular iv){
    return iv.TheInt;
  }

  public int Visit (IntCompSeq iv){
    return (iv.First.Accept(this) +
            iv.Second.Accept(this));
  }
}
```

Program 32.36 *The class SumVisitor.*

As it appears, each `Accept` method in the ***Composite*** calls a `Visit` method in a `Visitor` class, which in turn may call one or more `Accept` methods on a composite constituents. This leads to *indirect recursion* in between `Accept` methods and `Visit` methods. Compared with the natural object-oriented solution, which uses *direct recursion*, this is more complicated to understand.

The indirect recursion, pointed out above, may also be understood as a simulation of *double dispatching*. First, we dispatch on the `Visitor` object and next we dispatch on an object from the composite tree structure. Most object-oriented programming language only allows *single dispatching* - corresponding to message passing via a virtual method. This can be generalized to *multiple dispatching*, where the dynamic type of several objects determine which method to activate. The object-oriented part of Common Lisp - CLOS [Keene89] - supports multiple dispatching.

In Program 32.37 we show a client program with an integer sequence composite structure (line 7-15), three visitors (line 16-18), and sample activations of tree traversals (highlighted in line 21, 22, and 28). The output of the program is shown in Listing 32.38 (only on web).

```
using System;

class SeqApp {

  public static void Main(){

    IntSequence isq =
      new IntCompSeq(
            new IntCompSeq(
              new IntInterval(3,5), new IntSingular(-7) ),
            new IntCompSeq(
              new IntInterval(12,7), new IntCompSeq(
                                             new IntInterval(18,-18),
                                             new IntInterval(3,5)
                                             )));
    Visitor min = new MinVisitor();
    Visitor max = new MaxVisitor();
    Visitor sum = new SumVisitor();


    Console.WriteLine("Min: {0} Max: {1}", isq.Accept(min),
                                           isq.Accept(max));

//  Alternative activation of Visit methods:
//  Console.WriteLine("Min: {0} Max: {1}", min.Visit((IntCompSeq)isq),
//                                         max.Visit((IntCompSeq)isq));

    Console.WriteLine("Sum: {0}", isq.Accept(sum));
  }
}
```

Program 32.37 *A sample application of IntSequences and visitors.*

## 32.16. Visitors - Pros and Cons

Lecture 8 - slide 41

As it is already clear from our explanation of ***Visitor*** in Section 32.15 there are both advantages and disadvantages of this design pattern.

We summarize the consequences of ***Visitor*** in the following items:

- A new kind of traversal can be added without affecting the classes of the ***Composite***
- A ***Visitor*** encapsulates all methods related to a particular traversal
- State related to a traversal can - in a natural way - be represented in the ***Visitor***
- If a new class is added to the ***Composite*** all ***Visitor*** classes are affected
- The indirect recursion that involves `Accept` and the `Visit` methods is more complex than the direct recursion in the natural object-oriented solution

***Visitor*** is frequently used for processing of abstract syntax trees in compilation tools

In case you are going to study compilers implemented the object-oriented way, you will most likely encounter ***Visitors*** for such tasks as type checking and code generation.

## 32.17. References

| | |
|---|---|
| [Keene89] | Sonya E. Keene, *Object-Oriented Programming in Common Lisp*. Addison-Wesley Publishing Company, 1989. |
| [Gamma96] | E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Addison Wesley, 1996. |
| [Midi-sample] | The generated MIDI file http://www.cs.aau.dk/~normark/oop-csharp/midi/song.mid |
| [Mip-jan-08] | MIP Exam January 2008 (In Danish) http://www.cs.aau.dk/~normark/oop-07/html/mip-jan-08/opgave.html |
| [Song-and-timednote-classes] | The auxiliary classes TimedNote and Song http://www.cs.aau.dk/~normark/oop-07/html/mip-jan-08/c-sharp/mip.cs |
| [Factory-method] | Wikipedia: Design pattern: Factory Method http://en.wikipedia.org/wiki/Factory_method |
| [Abstract-factory] | Wikipedia: Design pattern: Abstract Factory http://en.wikipedia.org/wiki/Abstract factory |