Лекция 8. Интерфейсы

Интерфейсы выглядят подобно объявлениям классов, где ключевое слово `class` заменено `interface`, а методы не имеют тела. Назначение интерфейса состоит в том, чтобы позволить двум объектам взаимодействовать друг с другом.

В рекомендованных соглашениях об оформлении программного кода имена интерфейсов должны начинаться с буквы `I`. Таким образом, интерфейсные типы очень легко обнаружить в коде. Интерфейсы могут иметь присоединённые к ним модификаторы доступа. Они указывают на то, видимо ли объявление интерфейса вне сборки. Поскольку большинство интерфейсов представляют собой контракты взаимодействия между поставщиками и потребителями, интерфейсы обычно объявляются как `public`. Члены интерфейса не могут иметь никаких модификаторов доступа и всегда неявно общедоступны (`public`).

На практике обычно имеет смысл помещать объявления интерфейсов в отдельную сборку, содержащую только определения интерфейсов и константы, чтобы потребитель и поставщик могли основывать свои реализации в точности на одной и той же версии интерфейса.

Объявления интерфейсов могут включать ноль или более методов, свойств, событий и индексаторов. Все они неявно общедоступны и не могут быть статическими. Интерфейсы могут наследоваться от одного и более других интерфейсов. Синтаксис точно такой же, как при наследовании класса. Когда речь идет об интерфейсах, то предпочтительнее считать, что если интерфейс `B` наследуется от интерфейса `A`, это значит, что если реализован интерфейс `B`, то также должен быть реализован и интерфейс `A`. Наследование классов просто подразумевает отношение «является» (is—a), где базовая реализация также наследуется. Хотя при наследовании интерфейсов применяется тот же синтаксис, что и при наследовании классов, было бы неправильно рассматривать их как одно и то же, поскольку наследование интерфейсов объявляет обобщение, а никакой реализации при этом не наследуется. Таким образом, всякий раз, когда речь идет о *наследовании интерфейса*, следует думать об этом в терминах отношения *реализует*.

Ниже приведён пример объявления интерфейса:

```
public delegate void DBEvent(IMyDatabase sender);
public interface IMyDatabase : ISerializable, IDisposable
{
    void Insert(object element);
    int Count { get; }
    object this[int index] { get; set; }
    event DBEvent dbChanged;

}
```

В данном примере `IMyDatabase` также реализует `ISerializable` и `IDisposable`. Поэтому любой конкретный тип, реализующий `IMyDatabase`, также должен реализовать `ISerializable` и `IDisposable`; в противном случае этот конкретный тип не скомпилируется.

Интерфейсы поддерживают *множественное наследование* от других интерфейсов в синтаксическом смысле. Как и с множественным наследованием в C++, можно также строить ромбовидные решеточные иерархии, что показано в следующем коде:

```
public interface IUIControl
{
    void Paint();
}

public interface IEditBox : IUIControl
{
}

public interface IDropList : IUIControl
{
}
```

```
public class ComboBox : IEditBox, IDropList
{
    public void Paint()
    {
        // рисовать реализацию ComboBox }
    }
}
```

В этом примере оба интерфейса IEditBox и IDropList реализуют интерфейс IUIControl. И поскольку ComboBox реализует оба эти интерфейса, он должен реализовать объединение всех методов, объявленных в интерфейсах, которые он реализует непосредственно, плюс методы тех интерфейсов, от которых они наследуются, и т.д. В данном случае сюда относится только метод IUIControl.Paint.

Всё довольно просто: методы из всех интерфейсов объединяются вместе в одно большое объединение, формируя набор методов, который должен реализовать конкретный класс или структура. Поэтому класс ComboBox получает только одну реализацию метода Paint. Если выполнить приведение экземпляра ComboBox к ссылке IEditBox и ссылке IDropList, то вызов Paint через любую из них приведёт к вызову одной и той же реализации метода.

Иногда в интерфейсе требуется объявить метод, который скрывает метод унаследованного интерфейса. Чтобы предотвратить при этом выдачу предупреждения компилятором, можно воспользоваться ключевым словом new. В следующем примере IEditBox по той или иной причине должен объявлять метод Paint, чья сигнатура в точности совпадает с сигнатурой метода из IUIControl. Поэтому здесь необходимо использовать ключевое слово new:

```
public interface IUIControl
{
    void Paint();
}

public interface IEditBox : IUIControl
{
    new void Paint();
}

public interface IDropList : IUIControl
{
}

public class ComboBox : IEditBox, IDropList
{
    public void Paint()
    {
        Console.WriteLine("ComboBox.IEditBox.Paint()");
    }
}

...

ComboBox cb = new ComboBox();
cb.Paint();
((IEditBox)cb).Paint();
((IDropList)cb).Paint();
((IUIControl)cb).Paint();
```

При всех вызовах Paint они всегда превращаются в вызовы ComboBox.Paint. Это потому, что все обязательные методы, которые ComboBox должен реализовать, сливаются в один большой набор. Обе сигнатуры Paint — одна из IEditBox, и одна из IUIControl — сливаются в один элемент обязательного списка. В конце оба они отображаются на ComboBox.Paint. Существует возможность изменить такое поведение, применив явную реализацию интерфейса, где ComboBox может выбирать между двумя разными версиями Paint — одной из IEditBox, и одной из IUIControl.

Когда интерфейс IEditBox объявляет метод Paint, используя ключевое слово new, тем самым он скрывает метод Paint, объявленный в IUIControl. Когда вы вызываете ComboBox.Paint, то вызывается метод IEditBox.Paint, как если выбран IEditBox-путь в иерархии наследования. По сути, в любой момент, когда любой путь скрывает метод, то он скрывает метод для всех путей.

Реализация интерфейсов

При реализации интерфейсов в C# необходимо выбрать его реализацию одним из двух способов. По умолчанию реализации интерфейсов являются неявными реализациями. Реализация метода — часть общедоступного контракта класса, но также она реализует интерфейс неявно. Альтернативно вы можете реализовать интерфейс явным образом, причём реализация метода является приватной по отношению к реализующему классу и не становится частью общедоступного интерфейса. Явная реализация обеспечивает некоторую гибкость, особенно при реализации двух интерфейсов с одноимёнными методами в них.

Когда конкретный тип реализует методы наследуемых интерфейсов, и эти методы помечены как public, это известно как *неявная реализация интерфейса*. Что хорошего, когда конкретный тип реализует контракт определённого интерфейса, а потребитель объектов этого типа не может вызвать методы этого контракта? Например, следующий код некорректен:

```
public interface IUIControl
{
    void Paint();
}

public class StaticText : IUIControl
{
    void Paint(); // код не компилируется
}
```

При компиляции компилятор немедленно пожалуется на то, что класс StaticText не реализовал все методы унаследованного интерфейса — в данном случае, IUIControl. Чтобы это заработало, вы должны переписать код следующим образом:

```
public interface IUIControl
{
    void Paint();
}

public class StaticText : IUIControl
{
    public void Paint()
    {
    }
}
```

Теперь код не только скомпилируется, но при вызове Paint через ссылку на StaticText или через ссылку на IUIControl будет вызываться метод StaticText.Paint. Таким образом, потребители могут трактовать экземпляры StaticText полиморфно как экземпляры IUIControl.

Когда конкретный тип *реализует интерфейс явно*, его методы также становятся частью общедоступного контракта самого конкретного типа. Однако не всегда нужно, чтобы методы реализации интерфейса становились частью общедоступного интерфейса класса, реализующего этот интерфейс. Например, класс System.IO.FileStream реализует IDisposable, но вы не должны вызывать Dispose через экземпляр FileStream. Вместо этого вы должны сначала выполнить приведение ссылки на объект FileStream к интерфейсу IDisposable, а затем можно вызывать Dispose. Когда вам понадобится такое поведение для ваших собственных типов, вы должны реализовать интерфейсы, используя явную реализацию.

Также можно использовать явную реализацию для предоставления отдельных реализаций перекрывающихся методов из унаследованных интерфейсов. Давайте ещё раз вернемся к предыдущему

примеру. Если вы хотите представить отдельные реализации для IEditBox.Paint и IUIControl.Paint внутри ComboBox, то для этого нужно использовать явную реализацию интерфейсов:

```
public interface IUIControl
{
    void Paint();
}

public interface IEditBox : IUIControl
{
    new void Paint();
}

public interface IDropList : IUIControl
{
}

public class ComboBox : IEditBox, IDropList
{
    void IEditBox.Paint()
    {
        Console.WriteLine("ComboBox.IEditBox.Paint()");
    }

    void IUIControl.Paint()
    {
        Console.WriteLine("ComboBox.IUIControl.Paint()");
    }

    public void Paint()
    {
        ((IUIControl)this).Paint();
    }
}

...

ComboBox cb = new ComboBox();
cb.Paint();
((IEditBox)cb).Paint();
((IDropList)cb).Paint();
((IUIControl)cb).Paint();
```

Обратите внимание на изменение в синтаксисе. Теперь ComboBox содержит три реализации Paint. Одна специфична для интерфейса IEditBox, другая — специфична для интерфейса IUIControl, а последняя предназначена просто для удобства, чтобы предоставить метод Paint общедоступному интерфейсу класса ComboBox. Когда вы реализуете методы интерфейса явно, то не только добавляете имя интерфейса, отделённое точкой, к имени метода, но также удаляете модификатор доступа. Это исключает его из общедоступного контракта ComboBox. Однако явные реализации интерфейсов не являются точно приватными в том смысле, что вы можете вызывать их после приведения экземпляра ComboBox к требуемому типу интерфейса. В примере реализации ComboBox.Paint — той, что относится к общедоступному контракту ComboBox — мы выбираем, какую версию Paint вызвать — в данном случае IUIControl.Paint. Точно также можно было бы выбрать реализацию IEditBox.Paint явно и IUIControl.Paint неявно, тогда не понадобилась бы третья реализация Paint. При компиляции и запуске предыдущего примера будет выведено следующее:

```
ComboBox.IUIControl.Paint()
ComboBox.IEditBox.Paint()
ComboBox.IUIControl.Paint()
ComboBox.IUIControl.Paint()
```

Конечно, этот пример довольно надуманный, но он предназначен для того, чтобы продемонстрировать сложность явных реализаций интерфейсов и сокрытие членов при множественном наследовании интерфейсов.

Переопределение реализаций интерфейсов в производных классах

Предположим, что у вас есть удобная реализация `ComboBox`, как в предыдущем разделе, и разработчик решил не герметизировать этот класс, чтобы вы могли наследоваться от него. Теперь представим, что вы создаёте новый класс `FancyComboBox` и хотите, чтобы он как-то иначе себя рисовал. Вы можете попробовать что-нибудь вроде следующего:

```
public interface IUIControl
{
    void Paint();
    void Show();
}

public interface IEditBox : IUIControl
{
    void SelectText();
}

public interface IDropList : IUIControl
{
    void ShowList();
}

public class ComboBox : IEditBox, IDropList
{
    public void Paint()
    {
    }

    public void Show()
    {
    }

    public void SelectText()
    {
    }

    public void ShowList()
    {
    }
}

public class FancyComboBox : ComboBox
{
    public void Paint()
    {
    }
}

...

FancyComboBox cb = new FancyComboBox();
```

Однако компилятор предупредит вас о том, что `FancyComboBox.Paint` скрывает `ComboBox.Paint`, и что вы, возможно, подразумевали использование ключевого слова `new`. Это покажется неожиданным, если вы предполагаете, что методы, реализующие методы интерфейса, должны быть автоматически виртуальными. В C# это не так. Столкнувшись с подобной проблемой, у вас есть два выбора. Один — заново реализовать интерфейс `IUIControl` в классе `FancyComboBox`:

```
public interface IUIControl
```

```
{
    void Paint();
    void Show();
}

public interface IEditBox : IUIControl
{
    void SelectText();
}

public interface IDropList : IUIControl
{
    void ShowList();
}

public class ComboBox : IEditBox, IDropList
{
    public void Paint()
    {
        Console.WriteLine("ComboBox.Paint()");
    }

    public void Show()
    {
    }

    public void SelectText()
    {
    }

    public void ShowList()
    {
    }
}

public class FancyComboBox : ComboBox, IUIControl
{
    public new void Paint()
    {
        Console.WriteLine("FancyComboBox.Paint()");
    }
}

...

FancyComboBox cb = new FancyComboBox();
cb.Paint();
((IUIControl)cb).Paint();
((IEditBox)cb).Paint();
```

В этом примере следует отметить два момента. Во-первых, `FancyComboBox` перечисляет `IUIControl` в списке наследования. Так вы указываете, что `FancyComboBox` собирается заново реализовать интерфейс `IUIControl`. Если бы `IUIControl` наследовался от другого интерфейса, то `FancyComboBox` пришлось бы повторно реализовать методы унаследованного интерфейса. Также нужно было бы использовать ключевое слово `new` для `FancyComboBox.Paint`, поскольку он скрывает `ComboBox.Paint`. Это не было бы проблемой, если бы `ComboBox` реализовал метод `IUIControl.Paint` явно, поскольку он не был бы частью общедоступного контракта. Когда компилятор находит соответствие метода класса методу интерфейса, он также просматривает общедоступные методы базовых классов. В реальности `FancyComboBox` должен был бы указать, что он заново реализует `IUIControl.Paint`, но без повторного объявления его методов, так что компилятор должен был бы просто связать интерфейс с методами базового класса. Конечно, это было бы бессмысленно, поскольку причина повторной реализации интерфейса в производном классе — изменение поведения.

Если методы контракта интерфейса реализованы неявно, они должны быть общедоступными. Пока они отвечают этим требованиям, они также могут иметь другие атрибуты, включая ключевое слово `virtual`. Фактически реализация интерфейса `IUIControl` в классе `ComboBox` с применением виртуальных методов, в противоположность не виртуальным, несколько облегчает решение предыдущей задачи:

```
public interface IUIControl
{
    void Paint();
    void Show();
}

public interface IEditBox : IUIControl
{
    void SelectText();
}

public interface IDropList : IUIControl
{
    void ShowList();
}

public class ComboBox : IEditBox, IDropList
{
    public virtual void Paint()
    {
        Console.WriteLine("ComboBox.Paint()");
    }

    public void Show()
    {
    }

    public void SelectText()
    {
    }

    public void ShowList()
    {
    }
}
public class FancyComboBox : ComboBox
{
    public override void Paint()
    {
        Console.WriteLine("FancyComboBox.Paint() ");
    }
}

...

FancyComboBox cb = new FancyComboBox();
cb.Paint();
((IUIControl)cb).Paint();
((IEditBox)cb).Paint();
```

В этом случае класс `FancyComboBox` не обязан реализовать интерфейс `IUIControl`. Он должен просто переопределить виртуальный метод `ComboBox.Paint`. Намного яснее для `ComboBox` сразу объявлять `Paint` как `virtual`. Всякий раз, когда приходится использовать ключевое слово `new` для подавления предупреждений компилятора о сокрытии метода, следует рассмотреть возможность объявления метода базового класса как `virtual`.

Конечно, разработчику `ComboBox` нужно было бы заранее подумать о том, что кто-то пожелает провести наследование от класса `ComboBox`, и предвидеть эти сложности. Существует мнение, что лучше герметизировать

класс и избежать любых сюрпризов от тех, кто попытается выполнить от него наследование, когда класс никоим образом не предназначен для этого.

Во всех приведенных ранее примерах было показано, как классы могут реализовывать методы интерфейсов. На самом деле типы значений также могут реализовывать интерфейсы. Однако при этом возникает один главный побочный эффект. Приведение типа значений к интерфейсному типу вызывает упаковку. Хуже того, модификация значения через ссылку на интерфейс приводит к модификации упакованной копии, а не оригинала. Учитывая сложности, присущие упаковке, такое поведение может быть сочтено нежелательным.

Правила сопоставления членов интерфейсов

Каждый язык, поддерживающий определения интерфейсов, имеет правила сопоставления реализаций методов с методами интерфейсов. Сопоставление членов интерфейсов в C# достаточно прямолинейно и сводится к нескольким простым правилам. Однако, чтобы определить, какой именно метод в действительности вызывается во время выполнения, необходимо также учитывать правила CLR. Эти правила действуют только во время компиляции. Предположим, что есть иерархия классов и интерфейсов. Чтобы найти реализацию `SomeMethod` в `ISomeInterface`, начинайте со дна иерархии и ищите первый тип, реализующий нужный интерфейс. В данном случае этим интерфейсом является `ISomelnterface`. Это уровень, с которого начинается сопоставление метода. Как только тип найден, рекурсивно перемещайтесь вверх по иерархии типов и ищите метод с совпадающей сигнатурой, отдавая предпочтение явным реализациям членов интерфейса. Если ничего не найдено, обратитесь к общедоступным методам экземпляра, соответствующим той же сигнатуре.

Компилятор C# использует этот алгоритм при сопоставлении реализаций методов с реализациями интерфейсов. Выбранный им метод должен быть общедоступным методом экземпляра или явно реализованным методом экземпляра, причем он может быть (а может и не быть) помечен в C# модификатором `virtual`. Но даже если метод не помечен как виртуальный в смысле C#, среда CLR трактует вызовы интерфейса как виртуальные. Это может вызвать путаницу, как продемонстрировано в следующем надуманном примере:

```
public interface IMylnterface
{
    void Go();
}

public class A : IMylnterface
{
    public void Go()
    {
        Console.WriteLine("A.Go()");
    }
}

public class B : A
{
}

public class C : B, IMylnterface
{
    public new void Go()
    {
        Console.WriteLine("C.Go()");
    }
}

...

B b1 = new B();
C c1 = new C();
B b2 = c1;
b1.Go();
c1.Go();
b2.Go();
```

```
((IMylnterface)b2).Go();
```

Ниже показан вывод этого примера:

```
A.Go()
C.Go()
A.Go()
C.Go()
```

Первый вызов на b1 очевиден, как и следующий — на c1. Однако третий вызов — на b2 — не очевиден вовсе. Поскольку метод `A.Go` не помечен как `virtual`, компилятор генерирует код, вызывающий `A.Go`. Четвёртый, и последний, вызов почти так же запутан, но только если не учитывать тот факт, что CLR обрабатывает виртуальные вызовы на ссылках типа класса существенно иначе, чем вызовы на интерфейсных ссылках. Код для четвёртого вызова выполняет вызов `IMylnterface.Go`, который, в данном случае обращается в `C.Go`, потому что b2 — на самом деле `C`, а `C` реализует `IMylnterface`.

При поиске метода, вызываемого в действительности, следует проявлять осторожность, поскольку должно учитываться, является тип ссылки типом класса или типом интерфейса. Компилятор C# генерирует вызовы виртуальных методов, чтобы вызовы шли через интерфейсные методы, а CLR внутренне использует таблицы интерфейсов для обеспечения этого.

Содержимое этих интерфейсных таблиц определяется компилятором с использованием правил сопоставления методов. Сокрытие метода в одном пути ромбовидной решёточной иерархии позволяет скрыть метод во всех путях наследования. Правила устанавливают, что при проходе по иерархии поиск прекращается, как только на определённом уровне обнаруживается подходящий метод. Эти простые правила также объясняют, как повторная реализация интерфейса может существенно повлиять на процесс сопоставления методов, тем самым сокращая поиск компилятора при его проходе по иерархии. Рассмотрим пример такого действия:

```
public interface ISomeInterface
{
    void SomeMethod();
}

public interface IAnotherInterface : ISomeInterface
{
    void AnotherMethod();
}

public class SomeClass : IAnotherInterface
{
    public void SomeMethod()
    {
        Console.WriteLine("SomeClass.SomeMethod()");
    }

    public virtual void AnotherMethod()
    {
        Console.WriteLine("SomeClass.AnotherMethod()");
    }
}

public class SomeDerivedClass : SomeClass
{
    public new void SomeMethod()
    {
        Console.WriteLine("SomeDerivedClass.SomeMethod()");
    }

    public override void AnotherMethod()
    {
        Console.WriteLine("SomeDerivedClass.AnotherMethod()");
    }
```

```
}

...

SomeDerivedClass obj = new SomeDerivedClass();
ISomeInterface isi = obj;
IAnotherInterface iai = obj;
isi.SomeMethod();
iai.SomeMethod();
iai.AnotherMethod();
```

Применим правило поиска к каждому вызову метода в этом примере. Во всех случаях экземпляр SomeDerivedClass неявно преобразуется в ссылки на два интерфейса— ISomelnterface и IAnotherlnterface. Первый вызов SomeMethod выполнен через ISomelnterface. Сначала пройдёмся по иерархии классов, начиная с конкретного типа ссылки, в поисках первого класса, реализующего этот интерфейс или интерфейс, наследующий его. Это приведёт к реализации класса SomeClass, поскольку даже несмотря на то, что он не реализует ISomeInterface напрямую, всё же он реализует IAnotherlnterface, наследующий ISomelnterface. Таким образом, мы приходим к вызову SomeClass.SomeMethod. Вы можете удивиться, почему не был вызван метод SomeDerivedClass.SomeMethod. Но в соответствие с правилами, при поиске самого нижнего в иерархии класса, реализующего интерфейс, класс SomeDerivedClass будет пропущен. Чтобы вместо этого был вызван SomeDerivedClass.SomeMethod, класс SomeDerivedClass должен был бы повторно реализовать ISomelnterface. Второй вызов SomeMethod через ссылку IAnotherlnterface следует по точно тому же пути в поисках подходящего метода.

Всё становится интереснее при третьем вызове, где вызывается AnotherMethod через ссылку на IAnotherlnterface. Как и до этого, поиск начинается с самого нижнего в иерархии класса, реализующего этот интерфейс, т.е. внутри SomeClass. Поскольку в SomeClass имеется метод с подходящей сигнатурой, поиск закончен. Однако весь фокус в том, что метод с сопоставимой сигнатурой объявлен как virtual. Поэтому, когда выполняется вызов, механизм виртуальных методов перемещает точку выполнения внутрь SomeDerivedClass.AnotherMethod. Важно отметить, что, несмотря на виртуальность, AnotherMethod не изменяет правила сопоставления методов интерфейса. Не изменяет до тех пор, пока после обнаружения соответствия интерфейсного метода его виртуальная природа не повлияет на выбор реализации для вызова во время выполнения. Вывод предыдущего примера кода выглядит следующим образом:

```
SomeClass.SomeMethod()
SomeClass.SomeMethod()
SomeDerivedClass.AnotherMethod()
```

Явная реализация интерфейса с помощью типа значений

Интерфейсы общего применения, принимающие параметры в форме ссылки на System.Object, встречаются очень часто. Такие интерфейсы обычно являются широко используемыми, не обобщенными интерфейсами. Например, вот как выглядит интерфейс IComaparable:

```
public interface IComparable
{
    int CompareTo(object obj);
}
```

В том, что метод CompareTo принимает такой общий тип, есть большой смысл, потому что очень неплохо иметь возможность передавать ему почти все, что угодно, чтобы увидеть, как переданный объект сравнивается с объектом, который реализует CompareTo. Когда речь ид`т исключительно о ссылочных типах, здесь не происходит никакой потери эффективности, поскольку преобразование ссылочных типов к типу System.Object и обратно выполняется быстро во всех случаях. Но всё несколько усложняется, когда речь заходит о типах значений. Рассмотрим приведённый ниже код:

```
public struct SomeValue : IComparable
```

```
{
    private int n;

    public SomeValue(int n)
    {
        this.n = n;
    }

    public int CompareTo(object obj)
    {
        if (obj is SomeValue)
        {
            SomeValue other = (SomeValue)obj;
            return n - other.n;
        }
        else
            throw new ArgumentException("Неверный тип!");
    }
}

...

SomeValue val1 = new SomeValue(1);
SomeValue val2 = new SomeValue(2);
Console.WriteLine(val1.CompareTo(val2));
```

В простом вызове `WriteLine` производится сравнение `val1` и `val2`. Но давайте посмотрим внимательнее, сколько здесь потребуется операций упаковки. Поскольку `CompareTo` принимает объектную ссылку, `val2` нужно упаковать в точке вызова метода. В случае явной реализации метода `CompareTo` пришлось бы выполнить приведение значения `val1` к интерфейсу `IComparable`, что опять означает затраты на упаковку. Но и внутри метода `CompareTo` тоже несколько раз выполняется упаковка.

К счастью, при сравнении `SomeValue` с определёнными типами можно воспользоваться оптимизацией. Для выполнения работы можно предусмотреть безопасную в отношении типов версию метода `CompareTo`, как показано в следующем коде:

```
public struct SomeValue : IComparable
{
    private int n;

    public SomeValue(int n)
    {
        this.n = n;
    }

    int IComparable.CompareTo(object obj)
    {
        if (obj is SomeValue)
        {
            SomeValue other = (SomeValue)obj;
            return n - other.n;
        }
        else
            throw new ArgumentException("Неверный тип!");

    public int CompareTo(SomeValue other)
    {
        return n - other.n;
    }
}

...

SomeValue val1 = new SomeValue(1);
```

```
SomeValue val2 = new SomeValue(2);
Console.WriteLine(val1.CompareTo(val2));
```

В данном примере упаковка в вызове CompareTo полностью исключена. Это объясняется тем, что компилятор выбирает версию, наиболее подходящую для типа. В данном случае, поскольку метод IComparable.CompareTo реализован явно, существует только одна перегрузка CompareTo в общедоступном контракте SomeValue. Но даже если бы IComparable.CompareTo не был реализован явно, компилятор всё равно выбрал бы версию, безопасную в отношении типов. Типичный шаблон поведения включает сокрытие бестиповых версий от нечаянного применения, так что пользователь должен выполнять явную упаковку. Эта операция преобразует интерфейсную ссылку, чтобы добраться до бестиповой версии.

Главный итог состоит в том, что такому подходу нужно следовать при любой реализации интерфейса в типе значений, когда обнаруживается, что можно определить перегрузки с более высокой безопасностью типов, чем перечисленные в объявлении интерфейса.

Выбор между интерфейсами и классами

Один контракт, т.е. правила взаимодействия объектов, можно реализовать различными способами. В среде C# и .NET двумя главными способами являются интерфейсы и классы, причем классы могут даже быть абстрактными. Поскольку C# поддерживает абстрактные классы, контракт можно легко смоделировать посредством абстрактных классов. Но какой метод мощнее? И какой из них больше подходит?

Если контракт реализуется через определение интерфейса, то этот контракт поддерживает версии. Это значит, что однажды реализованный интерфейс не должен никогда изменяться, как если бы он был высечен в камне. Разумеется, позже его можно было бы изменить, но в результате будет утрачена популярность у клиентов, когда их код перестанет компилироваться с модифицированным интерфейсом. Рассмотрим следующий пример:

```
public interface IMyOperations
{
    void Operation1();
    void Operation2();
}

public class ClientClass : IMyOperations
{
    public void Operation1()
    {
    }

    public void Operation2()
    {
    }
}
```

После того как этот интерфейс IMyOperations был представлен миру, тысячи клиентов ринулись реализовывать его. Затем от клиентов начинают поступать запросы на поддержку Operation3 в библиотеке. Может показаться, что для решения этой задачи достаточно просто добавить Operation3 к интерфейсу IMyOperations, но это будет серьёзной ошибкой. После добавления еще одной операции к IMyOperations клиентский код перестанет компилироваться до тех пор, пока не будет реализована эта новая операция. К тому же, код в другой сборке, которому известно о новом интерфейсе IMyOperations, может попытаться привести экземпляр ClientClass к ссылке IMyOperations и затем вызвать Operation3, что приведёт к сбою во время выполнения. Ясно, что модифицировать уже опубликованный интерфейс нельзя.

Справиться с этой проблемой можно было бы, определив полностью новый интерфейс, скажем, IMyOperations2. Однако для получения нового поведения классу ClientClass пришлось бы реализовывать оба интерфейса, как показано в следующем коде:

```
public interface IMyOperations
{
```

```
    void Operation1();
    void Operation2();
}

public interface IMyOperations2
{
    void Operation1();
    void Operation2();
    void Operation3();
}

public class ClientClass : IMyOperations, IMyOperations2
{
    public void Operation1()
    {
    }

    public void Operation2()
    {
    }

    public void Operation3()
    {
    }
}

public class AnotherClass
{
    public void DoWork(IMyOperations ops)
    {
    }
}
```

Модификация `ClientClass` для поддержки новой операции из `IMyOperations2` не особенно трудна, но как быть с существующим кодом вроде показанного в `AnotherClass`? Проблема в том, что метод `DoWork` принимает тип `IMyOperations`. Для того чтобы он мог вызывать новый метод `Operation3`, нужно изменить прототип `DoWork`, или же код внутри него должен выполнять приведение параметра к типу `IOperations2`, что может дать сбой во время выполнения. Чтобы компилятор мог перехватывать как можно больше ошибок, связанных с типами, лучше всё-таки изменить прототип `DoWork`, чтобы он принимал тип `IMyOperations2`.

Таким образом, сделать доступной новую операцию клиентам — весьма трудоёмкая задача. Теперь рассмотрим ту же ситуацию, но с использованием абстрактного класса:

```
public abstract class MyOperations
{
    public virtual void Operation1()
    {
    }

    public virtual void Operation2()
    {
    }
}

public class ClientClass : MyOperations
{
    public override void Operation1()
    {
    }
    public override void Operation2()
    {
    }
}
```

```
public class AnotherClass
{
    public void DoWork(MyOperations ops)
    {
    }
}
```

MyOperations — это базовый класс для ClientClass. Его преимущество состоит в том, что при необходимости он может содержать реализацию по умолчанию. В противном случае виртуальные методы MyOperations могут быть объявлены как abstract, поскольку для клиентов не имеет смысла создавать экземпляры MyOperations. Теперь предположим, что необходимо добавить новый метод Operation3 в MyOperations, не затрагивая существующих клиентов. Это можно делать до тех пор, пока добавленная операция не является абстрактной, иначе понадобится вносить изменения в производные типы, как показано ниже:

```
public abstract class MyOperations
{
    public virtual void Operation1()
    {
    }

    public virtual void Operation2()
    {
    }

    public virtual void Operation3()
    {
        // Новая реализация по умолчанию
    }
}
// Клиентский класс
public class ClientClass : MyOperations
{
    public override void Operation1()
    {
    }

    public override void Operation2()
    {
    }
}

public class AnotherClass
{
    public void DoWork(MyOperations ops)
    {
        ops.Operation3();
    }
}
```

Обратите внимание, что добавление MyOperations.Operation3 не потребует никаких изменений в ClientClass, и AnotherClass.DoWork может вызывать Operation3, не внося никаких изменений в объявление метода. Такая техника, правда, не лишена своих недостатков. Вы ограничены тем фактом, что управляемая исполняющая система допускает наследование класса только от одного базового класса. Поскольку ClientClass наследует MyOperations, чтобы получить функциональность, он использует свой единственный билет на наследование. Это может наложить сложные ограничения на клиентский код.

Иногда отыскать золотую середину между интерфейсами и классами непросто. Можно руководствоваться следующими эмпирическими правилами.

- если моделируется отношение «является» (is—a), использовать класс. Если контракт можно осмысленно назвать именем существительным, то, вероятно, его стоит моделировать с помощью класса;

- если моделируется отношение «реализует» (implements), использовать интерфейс. Если контракт можно осмысленно назвать именем прилагательным, как если бы это было некоторое качество, то, вероятно, его необходимо моделировать с помощью интерфейса;
- рассмотреть возможность помещения интерфейса и объявления абстрактного класса в отдельную сборку. Тогда реализации в других сборках смогут ссылаться на эту отдельную сборку;
- по возможности отдавать предпочтением классам перед интерфейсами. Это может поспособствовать расширяемости.

Примеры применения описанных приемов повсеместно встречаются в библиотеке базовых классов .NET Framework (Base Class Library — BCL). Старайтесь использовать их и в собственном коде.