C# 코딩 연습

대리자와 이벤트

2007-12-13

김과장

(kimgwajang@hotmail.com)

A. 대리자

사람을 나타내는 간단한 Person 클래스로부터 시작하겠습니다.

```
01 public class Person
02 {
03     private int _age;
04
05     public int Age
06     {
07         get { return _age; }
08         set { _age = value; }
09     }
10
11     private string _name;
12
13     public string Name
14     {
15         get { return _name; }
16         set { _name = value; }
17     }
18
19     public Person(int age, string name)
20     {
21         _age = age;
22         _name = name;
23     }
24 }
```

코드 1

이 클래스에 나이를 한 살 더하는 메서드가 있다고 하지요.

```
1 public void IncreaseAge()
2 {
3     _age++;
4 }
```

코드 2

이제는 IncreaseAge 메서드에 나이가 변경되었음을 알리는 코드를 추가하는 것에 대해서 생각해 봅시다.

```
1 public void IncreaseAge()
2 {
3     int oldAge = _age;
4     _age++;
5
6     Console.WriteLine(string.Format("{0} -> {1}", oldAge, _age));
7 }
```

코드 3

직관적인 형태라서 좋긴 하지만, 6 번 라인이 문제가 됩니다. 콘솔에 출력을 하는 UI 로직이 클래스 내부에 추가됨으로써 이제 이 Person 클래스는 Console 이라는 UI 클래스와 강한 결합을 가지게 되었습니다. 즉 Person 클래스를 윈폼이나 웹페이지에서 사용하기가 어렵다는 이야기이지요.

이러한 클래스 간의 강한 결합을 피하기 위해서는, Person 클래스 내부에서 UI 관련 코드를 제거하여야 합니다. 대신 Person 클래스는 UI 코드에 대한 참조를 가지고 있는 것으로 충분합니다. 이러한 참조를 닷넷은 대리자라는 이름으로 제공을 하고 있습니다.

위 예를 대리자를 사용하는 형태로 변경을 해봅시다. 먼저 우리가 하고자 하는 작업에 적당한 대리자를 선언하여야 겠지요. 6 번 라인을 보고 짐작을 할 때 아마도 다음과 같은 시그니처를 가지는 대리자이면 적당할 것 같습니다.

```
public delegate void AgeChangedDelegate(int oldAge, int newAge);
```

또한 IncreaseAge 메서드는 UI 관련 코드를 제거하고 대신 대리자를 호출하는 형태로 바꿀 수 있습니다.

```
01 public AgeChangedDelegate AgeChanged;
02
03 public void IncreaseAge()
04 {
05     int oldAge = _age;
06     _age++;
```

```
07
08     if (AgeChanged != null)
09         AgeChanged(oldAge, _age);
10 }
```

코드 4

1 번 라인과 같이 대리자 필드를 선언하고 9 번 라인과 같이 이를 호출하고 있습니다.

이제 이 Person 클래스를 사용하는 코드를 생각해 봅시다.

```
01 static void Main(string[] args)
02 {
03     Person person = new Person(32, "김과장");
04     person.AgeChanged = new AgeChangedDelegate(Person_AgeChanged);
05     person.IncreaseAge();
06 }
07
08 private static void Person_AgeChanged(int oldAge, int newAge)
09 {
10     Console.WriteLine(string.Format("{0} -> {1}", oldAge, newAge));
11 }
```

코드 5

C# 2.0 부터는 코드 5 의 4 번 라인을 다음과 같이 줄여서 쓸 수 있습니다.

```
person.AgeChanged = Person_AgeChanged;
```

코드 5 의 실행 결과는 아래와 같습니다.

```
32 - > 33
계속하려면 아무 키나 누르십시오 . . .
```

B. 무명 메서드

C# 2.0 에 도입된 무명 메서드를 사용하면 코드 5 는 아래와 같이 고칠 수 있습니다.

```
01 01 static void Main(string[] args)
02 {
03     Person person = new Person(32, "김과장");
```

```
04     person.AgeChanged = delegate(int oldAge, int newAge)
05     {
06         Console.WriteLine(string.Format("{0} -> {1}", oldAge, newAge));
07     };
08
09     person.IncreaseAge();
10 }
```

코드 6

무명 메서드를 사용함으로써 코드가 한결 간결해지기도 했지만, 무명 메서드의 진가는 이 이상입니다. 예를 들어, 코드 6 을 이전 나이와 새 나이 외에도 생년까지 같이 표시하도록 고친다고 해봅시다. 이 때 Person 와 AgeChangedDelegate 대리자는 수정할 수 없는 상황이라고 가정을 합니다.

```
01 static void Main(string[] args)
02 {
03     int birthYear = 1976;
04
05     Person person = new Person(32, "김과장");
06     person.AgeChanged = delegate(int oldAge, int newAge)
07     {
08         Console.WriteLine(string.Format("{0} -> {1} : {2}년생", oldAge,
newAge, birthYear));
09     };
10
11     person.IncreaseAge();
12 }
```

코드 7

실행 결과는 다음과 같습니다.

```
32 - > 33 : 1976 년생
계속하려면 아무 키나 누르십시오 . . .
```

3 번 라인에서 선언한 변수 birthYear 를 무명 메서드 내에서 사용할 수 있는 것에 주목하여 주십시오. 만일 아래 코드 8 와 같이 명명된 메서드를 사용한다면 Person_AgeChanged 메서드에서는 birthYear 변수에 바로 접근할 수 없습니다. yearBirth

변수를 Person_AgeChanged 메서드에 전달하기 위해서는 yearBirth 변수를 클래스의 필드로 만든다든지 하는 방법을 사용하여야 할 것입니다.

```
01 static void Main(string[] args)
02 {
03     int birthYear = 1976;
04
05     Person person = new Person(32, "김과장");
06     person.AgeChanged = Person_AgeChanged;
07
08     person.IncreaseAge();
09 }
10
11 private static void Person_AgeChanged(int oldAge, int newAge)
12 {
13     // birthYear 변수가 알려지지 않았기 때문에 컴파일 에러
14     Console.WriteLine(string.Format("{0} -> {1} : {2}년생", oldAge,
newAge, birthYear));
15 }
```

코드 8

C. 이벤트

위 예제의 경우 대리자를 사용하여 구현하여도 로직과 UI 의 분리라는 일차적인 목표는 달성할 수 있지만, C#에는 이러한 작업을 위한 지원이 언어 차원에서 제공됩니다. 바로 이벤트 메커니즘이라고 하는 것이지요.

Person 클래스를 이벤트를 사용하는 형태로 변경해 봅시다.

```
01 public class Person
02 {
03     ...
04
05     public event AgeChangedEventHandler AgeChanged;
06
07     public void IncreaseAge()
08     {
09         int oldAge = _age;
```

```
10          _age++;
11
12          if (AgeChanged != null)
13              AgeChanged(oldAge, _age);
14      }
15 }
16
17 public delegate void AgeChangedEventHandler(int oldAge, int newAge);
```

코드 9

먼저 대리자의 이름을 AgeChangedDelegate 에서 AgeChangedEventHandler 로 변경하였습니다. 이는 닷넷 프레임웍의 명명 지침법을 따른 것입니다.

5 번 라인을 보면 이벤트를 선언하고 있습니다. 12~13 라인은 이전과 달라진 것이 없지만, 이제는 대리자가 아니라 이벤트를 호출하고 있다는 것도 확인하시기 바랍니다.

닷넷 프레임웍의 이벤트 작성 지침에 의하면 이벤트 대리자의 시그니처는 아래와 같은 형태를 따르도록 권장됩니다.

```
public delegate void AgeChangedEventHandler(object sender, EventArgs e);
```

첫번째 매개변수인 sender 로는 해당 이벤트를 발생시킨 객체를 전달하고, 그 외 이벤트 핸들러에서 사용할 값들은 EventArgs (또는 그의 자식 클래스) 객체의 필드로 포함시켜 두번째 매개변수로 전달합니다. 우리의 예제에서는 이전 나이와 새 나이를 멤버로 가지는 AgeChangedEventAgrs 형의 객체를 전달하겠네요.

그렇다면 AgeChangedEventAgrs 클래스를 만들어 봅시다.

```
01 public delegate void AgeChangedEventHandler(object sender,
AgeChangedEventArgs e);
02
03 public class AgeChangedEventArgs : EventArgs
04 {
05     private int _oldAge;
06     private int _newAge;
```

```
07
08     public int OldAge
09     {
10         get { return _oldAge; }
11         set { _oldAge = value; }
12     }
13
14     public int NewAge
15     {
16         get { return _newAge; }
17         set { _newAge = value; }
18     }
19
20     public AgeChangedEventArgs(int oldAge, int newAge)
21     {
22         _oldAge = oldAge;
23         _newAge = newAge;
24     }
25 }
```

코드 10

이제 Person 클래스는 이렇게 변경됩니다.

```
01 public event AgeChangedEventHandler AgeChanged;
02
03 public void IncreaseAge()
04 {
05     int oldAge = _age;
06     _age++;
07
08     if (AgeChanged != null)
09         AgeChanged(this, new AgeChangedEventArgs(oldAge, _age));
10 }
```

코드 11

Person 클래스를 사용하는 쪽의 코드는 이런 형태가 되겠지요.

```
01 static void Main(string[] args)
02 {
03     Person person = new Person(32, "김과장");
04     person.AgeChanged += Person_AgeChanged;
05
06     person.IncreaseAge();
07 }
```

```
08
09 private static void Person_AgeChanged(object sender,
AgeChangedEventArgs e)
10 {
11     Console.WriteLine(string.Format("{0} -> {1}", e.OldAge, e.NewAge));
12 }
```

코드 12

4 번 라인에서 이벤트 처리기를 추가할 때, = 연산자가 아닌 += 연산자를 사용하고 있습니다. 이벤트에 대해서는 = 연산을 수행할 수 없습니다.

실행 결과는 대리자를 사용하는 경우와 동일합니다.

```
32 - > 33
계속하려면 아무 키나 누르십시오 . . .
```

한편, C# 2.0 의 클래스 라이브러리에는 제네릭 버전의 EventHandler 대리자가 정의되어 있습니다. 이 제네릭 EventHandler 를 사용하면 AgeChangedEventHandler 는 따로 작성할 필요가 없습니다. 즉, 코드 10 의 1 번 라인은 삭제가 가능한데, 그러면 코드 11 는 아래와 같이 고칠 수 있습니다.

```
01 public event EventHandler<AgeChangedEventArgs> AgeChanged;
02
03 public void IncreaseAge()
04 {
05     int oldAge = _age;
06     _age++;
07
08     if (AgeChanged != null)
09         AgeChanged(this, new AgeChangedEventArgs(oldAge, _age));
10 }
```

코드 13

1 번 라인에서 AgeChangedEventHandler 대신 EventHandler<AgeChangedEventArgs>를 사용하고 있습니다. 이 때, Person 클래스를 사용하는 코드 12 은 수정할 필요가 없습니다.

D. 상속과 이벤트

1. Template Method 패턴

이번에는 연예인을 나타내는 클래스 Entertainer 를 작성하는 경우를 생각해 봅시다. 물론 연예인은 사람이니까 (is – a 관계) Entertainer 는 Person 을 상속받는 형태로 작성하면 좋겠지요.

```
1 public class Entertainer : Person
2 {
3     public Entertainer(int age, string name) : base(age, name)
4     {
5     }
6 }
```

코드 14

Entertainer 는 기본적으로 Person 과 동일하지만, 나이가 한 살 더해졌을 때 Person 과는 달리 이를 외부에 거짓으로 알려준다고 가정을 해볼까요. 즉 Entertainer 의 IncreaseAge 메소드가 아래처럼 구현된다고 생각합시다.

```
1 public void IncreaseAge()
2 {
3     int oldAge = _age;
4     _age++;
5
6     if (AgeChanged != null)
7         AgeChanged(this, new AgeChangedEventArgs(oldAge, oldAge));
8 }
```

코드 15

4 번 라인에서 보는 것처럼 물론 연예인도 나이가 더해지는 건 맞습니다. 하지만 이를 외부에 알려줄 때는 더해진 나이 대신에 예전 나이를 알려줍니다. 그래서 7 번 라인에서 newAge 가 들어갈 자리에 oldAge 를 넘기고 있는 것입니다.

물론 현재 IncreaseAge 메서드는 Person 과 Entertainer 에서 중복 정의되어 있습니다. 중복 문제를 해결하기 위해서는 (1) Entertainer.IncreaseAge 에 new 한정자를 붙이거나, (2) Person.IncreaseAge 를 가상 메서드로 만들고 Entertainer.IncreaseAge 가 이를 재정의(오버라이드)하는 두 가지 방법 중 하나를 선택할 수 있습니다. 나중에 Entertainer 클래스를 상속 받는 FemaleEntertainer 와 같은 클래스가 또 생길 수 있는 가능성을 염두에 두면 가상 메서드의 재정의를 사용하는 것이 좀 더 유연하겠지요.

Person.IncreaseAge 메서드를 가상으로 만들고 Entertainer.IncreaseAge 가 이를 재정의하는 코드는 아래와 같습니다.

```
1 public virtual void IncreaseAge()
2 {
3     int oldAge = _age;
4     _age++;
5
6     if (AgeChanged != null)
7         AgeChanged(this, new AgeChangedEventArgs(oldAge, _age));
8 }
```

코드 16 Person 의 IncreaseAge 메서드

```
1 public override void IncreaseAge()
2 {
3     int oldAge = _age;
4     _age++;
5
6     if (AgeChanged != null)
7         AgeChanged(this, new AgeChangedEventArgs(oldAge, oldAge));
8 }
```

코드 17 Entertainer 의 IncreaseAge 메서드

하지만 가상 메서드를 재정의했음에도 불구하고, 코드 16 과 코드 17 은 여전히 몇가지 문제를 가지고 있습니다.

- _age 변수는 Person 의 private 필드이므로 Entertainer 클래스에서 접근할 수 없습니다.
- 3 ~ 4 번 라인이 두 클래스에서 중복되어 있습니다.
- 파생 클래스에서 부모 클래스에 정의된 이벤트에 대해서 수행할 수 있는 연산은 += 와 -= 밖에 없습니다. 즉 코드 17 의 6 ~ 7 번 라인은 컴파일 에러입니다.

결국 IncreaseAge 메서드를 재정의해서 사용하는 방법으로는 문제를 해결할 수 없을 것 같습니다. 대신 다른 방법을 생각해 보아야 할 것인데요. 위 코드 16 과 코드 17 을 유심히 보면, 두 코드에서 차이가 나는 부분은 7 번 라인 밖에 없습니다. IncreaseAge 메서드는 Person 클래스에서 한 번만 정의하고, 두 코드에서 서로 다르게 동작하는 7 번 라인에 해당하는 부분을 새로운 메서드로 만들어서, IncreaseAge 메서드가 새로운 메서드를 호출하는 로직으로 바꾸면 될 것 같습니다. 이 때, 이 새로운 메서드는 Person 과 Entertainer 에서 서로 다르게 동작을 해야하므로 가상 메서드로 만들어야 합니다. 이러한 패턴을 바로 Template Method 패턴이라고 합니다.

코드를 살펴보면서 이야기를 해보지요. Person 클래스의 코드부터 봅시다.

```
01 protected virtual void OnAgeChanged(AgeChangedEventArgs e)
02 {
03     if (AgeChanged != null)
04         AgeChanged(this, e);
05 }
06
07 public void IncreaseAge()
08 {
09     int oldAge = _age;
10     _age++;
```

```
11
12     OnAgeChanged(new AgeChangedEventArgs(oldAge, _age));
13 }
```

코드 18

이벤트를 호출하는 부분을 따로 빼서 OnAgeChanged 라는 메서드를 새로 만들었습니다. 이 메서드는 Entertainer 에서 재정의할 것이기 때문에 가상 메서드로 선언하였습니다. 그리고 IncreaseAge 메서드는 이 OnAgeChanged 메서드를 호출하고 있습니다. IncreaseAge 가 더 이상 가상 메서드가 아닌 점도 확인하여 주십시오.

Entertainer 클래스의 코드도 보지요.

```
1 protected override void OnAgeChanged(AgeChangedEventArgs e)
2 {
3     base.OnAgeChanged(new AgeChangedEventArgs(e.OldAge, e.OldAge));
4 }
```

코드 19

Entertainer 클래스에서는 이제 IncreaseAge 메서드가 없습니다. 대신 OnAgeChanged 메서드를 재정의하는 코드가 추가되었습니다. 재정의된 OnAgeChanged 메서드는 Person 의 OnAgeChanged 를 호출합니다. 이때 OnAgeChanged 의 매개변수를 적절히 조작하여 전달을 합니다. 우리의 예에서는 새 나이 대신에 예전 나이를 넘기고 있습니다.

만일 연예인의 경우, 나이가 더해졌을 때 새 나이 대신에 예전 나이를 알려주는 것이 아니라 아예 나이를 알려주지 않는다고 한다면, 코드 19 에서 3 번 라인을 삭제해버리면 될 것입니다. 이처럼 가상 메서드를 재정의해서 사용하면 대단히 유연한 제어를 할 수가 있습니다.

Person 과 Entertainer 를 사용하는 코드는 아마도 이런 식이겠지요.

```
01 static void Main(string[] args)
02 {
03     Person person = new Person(32, "김과장");
04     person.AgeChanged += Person_AgeChanged;
05     person.IncreaseAge();
06
07     Person entertainer = new Entertainer(32, "김스타");
08     entertainer.AgeChanged += Person_AgeChanged;
09     entertainer.IncreaseAge();
10 }
11
12 private static void Person_AgeChanged(object sender,
AgeChangedEventArgs e)
13 {
14     Person person = sender as Person;
15     if (person != null)
16     {
17         Console.WriteLine(string.Format("{0} : {1} -> {2}", person.Name,
e.OldAge, e.NewAge));
18     }
19 }
```

코드 20

3 번 라인과 7 번 라인에서 각각 Person 과 Entertainer 의 객체를 하나씩 생성하였습니다. 그리고 두 객체의 AgeChanged 이벤트에 모두 Person_AgeChanged 이벤트 핸들러를 등록하였습니다. 또한 14 번 라인에서 확인할 수 있듯이, Person_AgeChanged 이벤트 핸들러의 sender 매개변수에는 이벤트를 발생시킨 객체가 넘어옵니다. 위 예에서는 person 과 entertainer 객체가 각각 넘어오겠네요. 왜냐하면 코드 18 의 4 번 라인의 AgeChanged(this, e); 라는 코드에 의해 Person_AgeChanged 이벤트 핸들러가 실행되는데, 이때 this 객체는 Person 객체(혹은 Person 에서 상속받은 Entertainer 객체)이기 때문입니다.

실행 결과는 다음과 같습니다.

```
김과장 : 32 - > 33
김스타 : 32 - > 32
```

```
계속하려면 아무 키나 누르십시오 . . .
```

예상한대로 연예인인 김스타는 민간인이 김과장과는 달리, 32 살에서 33 살이 되어도 여전히 32 살이라고 외부에 알려주고 있습니다.

여기까지의 Person 과 Entertainer 의 전체 코드는 다음과 같습니다.

```
01 public class AgeChangedEventArgs : EventArgs
02 {
03     private int _oldAge;
04     private int _newAge;
05
06     public int OldAge
07     {
08         get { return _oldAge; }
09         set { _oldAge = value; }
10     }
11
12     public int NewAge
13     {
14         get { return _newAge; }
15         set { _newAge = value; }
16     }
17
18     public AgeChangedEventArgs(int oldAge, int newAge)
19     {
20         _oldAge = oldAge;
21         _newAge = newAge;
22     }
23 }
```

코드 21 AgeChangedEventArgs 클래스

```
01 public class Person
02 {
03     private int _age;
04
05     public int Age
06     {
07         get { return _age; }
08         set { _age = value; }
09     }
10
11     private string _name;
12
```

```
13     public string Name
14     {
15         get { return _name; }
16         set { _name = value; }
17     }
18
19     public Person(int age, string name)
20     {
21         _age = age;
22         _name = name;
23     }
24
25     public event EventHandler<AgeChangedEventArgs> AgeChanged;
26
27     protected virtual void OnAgeChanged(AgeChangedEventArgs e)
28     {
29         if (AgeChanged != null)
30             AgeChanged(this, e);
31     }
32
33     public void IncreaseAge()
34     {
35         int oldAge = _age;
36         _age++;
37
38         OnAgeChanged(new AgeChangedEventArgs(oldAge, _age));
39     }
40 }
```

코드 22 Person 클래스

```
01 public class Entertainer : Person
02 {
03     public Entertainer(int age, string name) : base(age, name)
04     {
05     }
06
07     protected override void OnAgeChanged(AgeChangedEventArgs e)
08     {
09         base.OnAgeChanged(new AgeChangedEventArgs(e.OldAge, e.OldAge));
10     }
11 }
```

코드 23 Entertainer 클래스

2. 상속된 윈폼의 이벤트

상속과 이벤트에 대해서 또 한가지 생각해 볼 문제가 있습니다.

윈도우 폼 응용프로그램이 BaseForm 과 DerivedForm 이라는 두 개의 윈폼 클래스를 가지고 있다고 합시다. BaseForm 은 System.Windows.Forms.Form 을 상속받으며 DerivedForm 은 이 BaseForm 을 상속받는 형태입니다.

이때, 두 윈폼 클래스의 Load 이벤트가 발생할 때 어떤 작업, 여기서는 간단히 메시지 박스를 띄우는 일을 해보겠습니다. 먼저 BaseForm 의 Load 이벤트에 이벤트 핸들러를 연결합니다.

```
01 public partial class BaseForm : Form
02 {
03     public BaseForm()
04     {
05         InitializeComponent();
06
07         Load += new EventHandler(BaseForm_Load);
08     }
09
10     private void BaseForm_Load(object sender, EventArgs e)
11     {
12         MessageBox.Show("BaseForm Loaded");
13     }
14 }
```

코드 24

그 다음에는 DerivedForm 의 Load 이벤트에 대해서도 동일한 작업을 합니다.

```
01 public partial class DerivedForm : BaseForm
02 {
03     public DerivedForm()
04     {
05         InitializeComponent();
06
07         Load += new EventHandler(DerivedForm_Load);
08     }
09
10     private void DerivedForm_Load(object sender, EventArgs e)
11     {
12         MessageBox.Show("DerivedForm Loaded");
13     }
14 }
```

코드 25

코드 24 와 코드 25 의 7 번 라인을 보면 Form 클래스의 Load 이벤트에 각각 이벤트 핸들러를 등록하고 있습니다. 즉 Load 이벤트에는 BaseForm_Load 와 DerivedForm_Load 라는 두 개의 이벤트 핸들러가 등록되어 있습니다. 따라서 DerivedForm 이 로드되면 BaseForm_Load 이벤트 핸들러와 DerivedForm_Load 이벤트 핸들러가 순서대로 실행됩니다.

그런데, 만일 이 두 이벤트 핸들러를 이 순서대로 실행하는 것이 우리가 의도하는 바가 아니라면 어떻게 할까요? 예컨데 DerivedForm 이 로드될 때 BaseForm_Load 는 실행하지 않고 DerivedForm_Load 만 실행하려고 한다면요?

이러한 문제를 해결하기 위해서는 Load 이벤트가 정의된 클래스(Form)의 자식 클래스들(BaseForm, DerivedForm)에서 Load 이벤트에 바로 이벤트 핸들러를 등록하여서는 안됩니다. 대신 앞에서 이야기한 것 처럼, Form 클래스의 메서드 중 Load 이벤트를 호출하는 메서드를 찾아 이를 BaseForm 과 DerivedForm 에서 재정의하여야 합니다.

Form 클래스는 이미 이러한 용도로 사용하기 위해 OnLoad 라는 가상 메서드를 제공하고 있습니다. OnLoad 메서드의 구현은 아마도 다음과 같은 형태일 것입니다.

```
1 protected virtual void OnLoad(EventArgs e)
2 {
3     if (Load != null)
4         Load(this, EventArgs.Empty);
5 }
```

코드 26

이 OnLoad 메서드를 재정하는 BaseForm 의 코드는 다음과 같습니다.

```
01 public partial class BaseForm : Form
02 {
03     public BaseForm()
04     {
05         InitializeComponent();
06     }
07
08     protected override void OnLoad(EventArgs e)
09     {
10         MessageBox.Show("BaseForm Loaded");
11     }
12 }
```

코드 27

코드에서 드러나지는 않지만, 코드 24 의 7 번 라인과 같이 Load += new EventHandler(BaseForm_Load); 와 같은 코드는 이제 존재하지 않습니다. 즉, BaseForm 은 Form 의 Load 이벤트을 사용하지 않습니다. 하지만 Load 이벤트에 이벤트 핸들러를 등록한 것과 동일하게 동작을 합니다. 어떻게 이것이 가능할까요?

Form 클래스의 코드 중에, 자신이 로드가 되고 나면 OnLoad 메서드를 호출하는 코드가 있을 것입니다. 그런데 이 OnLoad 메서드는 가상 메서드이므로 BaseForm 클래스에서 이를 재정의해버리면, 결국 Form 클래스는 자신이 로드 되고 나면 자신의 OnLoad 메서드를 호출하는 것이 아니라 BaseForm 의 OnLoad 메서드를 호출하게 되는 것입니다.

재미있는 코드를 한번 볼까요. 아래 코드를 보고 결과를 예상해 보십시오.

```
01 public partial class BaseForm : Form
02 {
03     public BaseForm()
04     {
05         InitializeComponent();
06
07         Load += BaseForm_Load;
08     }
09
```

```
10      void BaseForm_Load(object sender, EventArgs e)
11      {
12          MessageBox.Show("BaseForm Loaded By Event");
13      }
14
15      protected override void OnLoad(EventArgs e)
16      {
17          // base.OnLoad(e);
18
19          MessageBox.Show("BaseForm Loaded");
20      }
21 }
```

코드 28

Load 이벤트에 이벤트 핸들러도 등록하고, 동시에 OnLoad 메서드도 재정의 하였습니다. 그렇다면 12 번 라인과 19 번 라인의 메시지 박스는 둘 다 나타날까요?

BaseForm 이 로드가 되었을 때, BaseForm 이 상속 받은 Form 의 어떤 코드에 의해서 OnLoad 메서드가 실행이 됩니다. 그런데 이 OnLoad 메서드는 가상 메서드이기 때문에, Form 의 것이 아니라 BaseForm 의 OnLoad 가 실행됩니다. 그것이 동작의 전부입니다. Load 이벤트에 등록된 이벤트 핸들러들을 실행하는 코드는 Form 의 OnLoad 에 있는데, 이 Form 의 OnLoad 는 BaseForm 의 OnLoad 에 의해 가리워져 실행이 되지 않는 것입니다. 따라서 12 번의 메시지 박스는 나타나지 않습니다.

만일 17 번 라인의 주석을 풀고 실행을 하면 어떤 결과가 나타날까요? '12 번의 메시지 박스가 먼저 나타나고 그 다음에 19 번의 메시지 박스가 나타난다'고 설명을 할 수 있다면, 원래의 목적이었던 BaseForm 과 DerivedForm 의 Load 이벤트들을 제어하는 것은 쉽게 하실 수 있을 것입니다. 여기서는 확인 삼아 DerivedForm 의 메시지 박스를 먼저 띄우고 그 다음에 BaseForm 의 메시지 박스를 띄우는 코드를 적겠습니다. 생각한 코드가 맞는지 확인하시기 바랍니다.

```
01 public partial class BaseForm : Form
02 {
03     public BaseForm()
04     {
05         InitializeComponent();
06     }
07
08     protected override void OnLoad(EventArgs e)
09     {
10         MessageBox.Show("BaseForm Loaded");
11     }
12 }
```

코드 29 BaseForm

```
01 public partial class DerivedForm : BaseForm
02 {
03     public DerivedForm()
04     {
05         InitializeComponent();
06     }
07
08     protected override void OnLoad(EventArgs e)
09     {
10         MessageBox.Show("DerivedForm Loaded");
11
12         base.OnLoad(e);
13     }
14 }
```

코드 30 DerivedForm

E. 이벤트 코드 조각 생성기

이때까지의 이야기를 종합하여 이벤트를 추가하는 과정을 정리하면 다음과 같습니다.

1. 이벤트에서 사용할 대리자를 작성한다. 또는 제네릭 EventHandler 를 사용한다.
2. EventArgs 를 상속하는 이벤트 인자 클래스를 작성한다.
3. 이벤트를 선언한다.
4. 이벤트를 호출하는 가상 메서드를 작성한다.

각 과정에 따라 이벤트 관련 코드를 작성하다보면 반복적인 타이핑 작업이 일어남을 알 수 있습니다. 그렇다면 이런 기계적이고 반복적인 코드를 자동으로 생성할 수도 있을 것입니다. 그래서 간단한 툴을 만들어 보았습니다. 특별한 이름을 짓지 못해 그냥 이벤트 코드 조각 생성기라고 부르겠습니다.

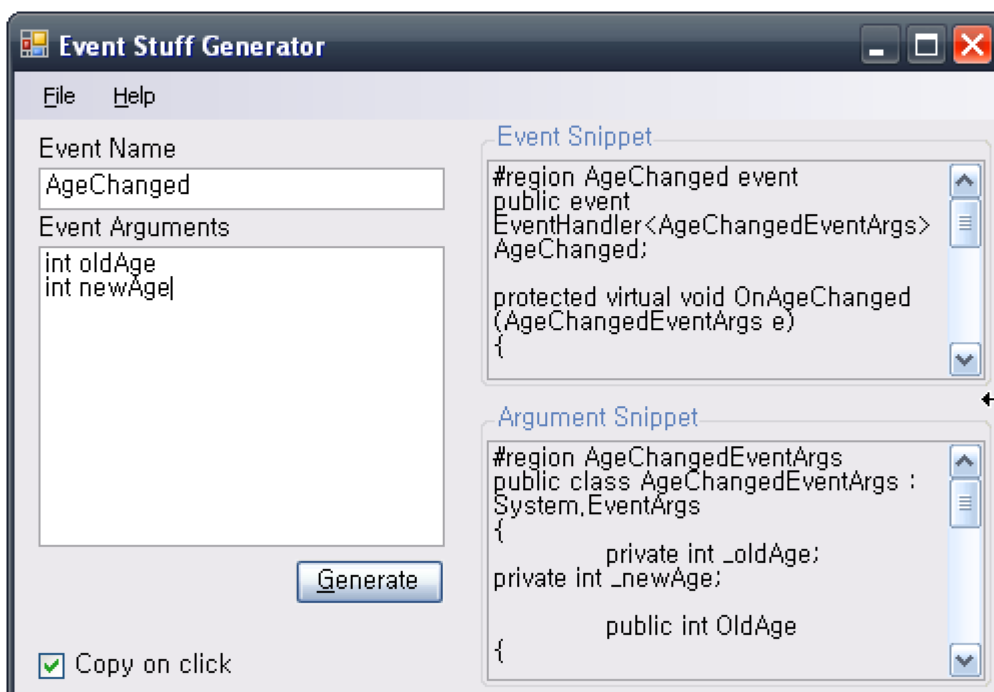

그림 1 이벤트 코드 조각 생성기

1. 사용방법

사용 방법은 간단합니다. Event Name 에 생성할 이벤트의 이름을 입력하고, Event Arguments 에 매개변수들을 입력하면 됩니다. 그림과 같이 각 매개변수는 개행으로 구분하고 매개변수의 형과 이름은 띄어쓰기로 구분을 합니다. Generate 버튼을 누르면 오른편에 있는 두 개의 텍스트 박스에 생성된 코드들이 출력됩니다. 이를 복사하여 사용하면 되겠습니다.

생성된 코드가 출력되는 텍스트 박스를 두 개로 나눈 이유는 생성되는 두 개의 코드 조각이 복사될 위치가 다르기 때문입니다. 위쪽 텍스트 박스의 경우 이벤트의 선언과 이벤트를 호출하는 가상 메서드가 생성되는데, 이는 클래스 내부에 들어갈 코드 조각입니다. 반면에 아래쪽 텍스트 박스에는 이벤트 인자 클래스가 생성되는데, 이는 이벤트 클래스의 외부에 두는 것이 권장되는 구조입니다.

또한 Copy on click 체크박스가 클릭되어 있는 상태에서 오른 쪽의 생성된 코드를 클릭하면 생성된 코드가 클립보드로 바로 복사가 됩니다.

2. 템플릿

이벤트 코드 조각 생성기의 핵심 로직은 간단합니다. 생성될 코드 조각의 뼈대가 되는 템플릿을 사용자가 입력한 이벤트 이름과 이벤트 인자를 사용하여 치환하는 것이거든요. 예를 들어 Event Snippet 텍스트 박스에서 사용되는 템플릿의 내용은 다음과 같습니다.

```
매크로      의미                 예
{0}         이벤트 이름          AgeChanged
{1}         이벤트 인자 클래스   AgeChangedEventArgs
{2}         매개변수 형과 이름 목록  int oldAge, int newAge
{3}         매개변수 이름 목록   oldAge, newAge

### 이하 템플릿 시작
#region {0} event
public event EventHandler<{1}> {0};

protected virtual void On{0}({1} e)
{{
	if({0} != null)
		{0}(this, e);
}}

protected virtual void On{0}({2})
{{
	if({0} != null)
```

```
                {0}(this, new {1}({3}));
}}
#endregion
```

필요한 경우 이 템플릿 파일을 수정하면 생성되는 코드를 제어할 수 있습니다.

이벤트 코드 조각 생성기는 두 개의 템플릿을 사용합니다. (EventSnippet.txt, ArgumentSnippet.txt) 자세한 사항은 이벤트 코드 조각 생성기의 소스를 참고하시기 바랍니다.

3. 데모

이벤트 코드 조각 생성기를 사용하여 Person 클래스에 AgeChanged 이벤트를 추가하는 과정을 살펴보겠습니다. 먼저 이벤트 코드 조각 생성기를 실행시키고, 이벤트 이름과 이벤트 인자를 각각 다음과 같이 입력합니다.

이벤트 이름
```
AgeChanged
```

이벤트 인자
```
int oldAge
int newAge
```

Generate 버턴을 누르면 이벤트 코드 조각이 생성됩니다. 먼저 Event Snippet 의 내용을 클릭하여 복사한 후 Person 클래스에 붙여 넣습니다.

```
01 public class Person
02 {
03     ...
04     #region AgeChanged event
05     public event EventHandler<AgeChangedEventArgs> AgeChanged;
06
07     protected virtual void OnAgeChanged(AgeChangedEventArgs e)
08     {
09         if (AgeChanged != null)
10             AgeChanged(this, e);
11     }
```

```
12
13     protected virtual void OnAgeChanged(int oldAge, int newAge)
14     {
15         if (AgeChanged != null)
16             AgeChanged(this, new AgeChangedEventArgs(oldAge, newAge));
17     }
18     #endregion
19
20     public void IncreaseAge()
21     {
22         int oldAge = _age;
23         _age++;
24
25         OnAgeChanged(oldAge, _age);
26     }
27 }
```

코드 31

13 번 라인의 메서드는 이전 예제에서는 등장하지 않았던 메서드입니다. 이는 단지 코딩의 편리함을 위해 원래의 OnAgeChanged 메서드를 래퍼하는 것에 불과합니다. 25 번 라인 역시 이 새로운 OnAgeChanged 메서드를 호출하는 형태로 변경되었습니다.

이번에는 Arguments Snippet 의 코드를 복사하여 Person 클래스의 외부에 붙여넣습니다. 물론 AgeChangedEventArgs 클래스를 Person 클래스 내부에 포함된 클래스로 위치시킬 수도 있겠지만 권장되는 설계는 아닙니다. AgeChangedEventArgs 의 코드는 코드 21 과 동일하므로 생략합니다.

F. 사용자 정의 컨트롤의 이벤트

이벤트를 작성하는 좀 더 실용적인 예를 생각해 봅시다. 윈도우 응용 프로그램을 개발하는 중 사용자 정의 컨트롤을 만든다고 가정을 하지요. 대략 다음과 같은 컨트롤을 만들었습니다. (이하 이 사용자 정의 컨트롤을 MemoWriter 라고 하겠습니다.)

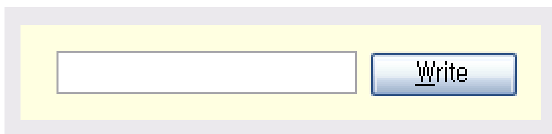

그림 2 MemoWriter 사용자 정의 컨트롤

MemoWriter 는 사용자로부터 메모를 입력 받는 간단한 기능을 하는 컨트롤입니다. 이 컨트롤을 가지고 있는 폼 역시 아주 단순합니다. (이하 이 윈폼을 Form1 이라고 하겠습니다.)

그림 3 Form1 윈폼 클래스

사용자가 텍스트 박스에 메모를 입력하고 Write 버튼을 클릭하면 입력한 메모를 메시지 박스로 띄우는 기능을 구현하고자 합니다. 어떤 방법이 있을까요?

만일 버튼이 Form1 의 자식 컨트롤이라면 단순히 버튼의 클릭 이벤트 핸들러를 만드는 것으로 가능합니다. 하지만 문제는 버튼은 Form1 의 자식 컨트롤이 아니라 MemoWriter 의 자식 컨트롤입니다. 따라서 Form1 은 자신의 자식 컨트롤이 아닌 버튼의 이벤트에 이벤트 핸들러를 등록할 수가 없습니다.

이 문제를 해결하는 정석은 MemoWriter 에 Written 이라는 이벤트를 추가하고, Form1 이 이 이벤트에 대한 이벤트 핸들러를 등록하는 것입니다. MemoWriter 안에 있는 버튼이 클릭되었을 때, MemoWriter 가 Written 이벤트를 발생시키면, Form1 에 있는 Written 의 이벤트 핸들러가 실행되는 로직이지요.

직접 코드를 보면서 이야기 하면 이해가 더 쉽겠지요. 먼저 MemoWriter 에 Written 이벤트 관련 코드를 추가합니다.

```
01 public partial class MemoWriter : UserControl
02 {
03     public MemoWriter()
04     {
05         InitializeComponent();
06     }
07
08     private void btnWrite_Click(object sender, EventArgs e)
09     {
10         OnWritten(txtMemo.Text);
11     }
12
13     #region Written event
14     public event EventHandler<WrittenEventArgs> Written;
15
16     protected virtual void OnWritten(WrittenEventArgs e)
17     {
18         if (Written != null)
19             Written(this, e);
20     }
21
22     protected virtual void OnWritten(string memo)
23     {
24         if (Written != null)
25             Written(this, new WrittenEventArgs(memo));
26     }
27     #endregion
28 }
29
30 #region WrittenEventArgs
31 public class WrittenEventArgs : EventArgs
32 {
33     private string _memo;
34
35     public string Memo
36     {
37         get { return _memo; }
38         set { _memo = value; }
39     }
40
41     public WrittenEventArgs(string memo)
42     {
43         _memo = memo;
44     }
```

```
45 }
46 #endregion
```

코드 32

8 ~ 11 번 라인을 주의해서 볼 필요가 있습니다. 버튼에 대한 이벤트 핸들러 내에서 Written 이벤트를 다시 발생시키고 있습니다. (물론 btnWrite.Clicked 이벤트에 btnWrite_Click 이벤트 핸들러를 등록하는 코드는 생략되어 있습니다.)

Form1 의 코드는 아래와 같습니다.

```
01 public partial class Form1 : Form
02 {
03     public Form1()
04     {
05         InitializeComponent();
06     }
07
08     private void uscMemoWriter_Written(object sender, WrittenEventArgs
e)
09     {
10         MessageBox.Show(e.Memo);
11     }
12 }
```

코드 33

MemoWriter 의 Written 이벤트에 이벤트 핸들러를 등록하였네요. 또한 10 번 라인에서 WrittenEventArgs 의 Memo 프로퍼티를 통해 사용자가 텍스트 박스에 입력한 메모 내용을 읽을 수 있는 것도 알 수 있습니다. (물론 코드 33 에서도 MemoWriter 의 인스턴스인 usc MemoWriter 의 Written 이벤트에 uscMemoWriter_Written 이벤트 핸들러를 등록하는 코드는 생략되었습니다.)