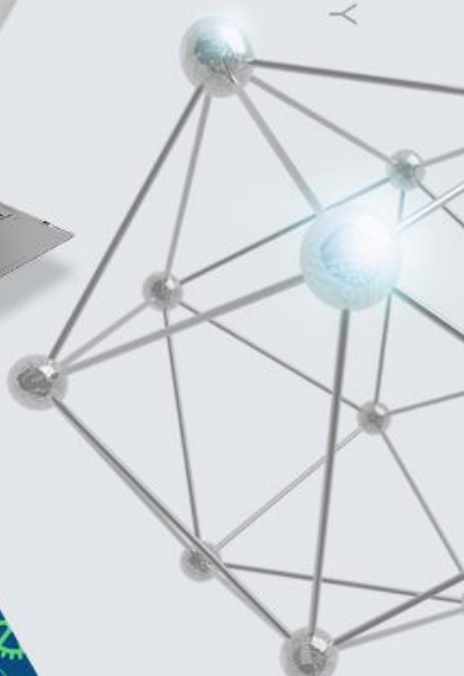
한국기술교육대학교
온라인평생교육원

The 4th Industrial Revolution is characterized by super connectivity and super intelligence, where various products and services are connected to the network, and artificial intelligence and information communication technologies are used in 3D printing, unmanned transportation, robotics, Of the world's most advanced technologies.

TECHNOLOGY

C# 프로그래밍

클래스



The 4th Industrial Revolution is characterized by super connectivity and super intelligence, where various products and services are connected to the network, and artificial intelligence and information communication technologies are used in 3D printing, unmanned transportation, robotics, Of the world's most advanced technologies.

클래스

학/습/목/표

1. 클래스와 객체를 이해하고 설명할 수 있다.
2. 필드, 메소드, 프로퍼티를 이해하고 구현 할 수 있다.
3. 인덱서, 델리게이트, 이벤트를 이해하고 구현 할 수 있다.
4. 연산자 중복과 구조체를 이해하고 구현 할 수 있다.

학/습/내/용

1. 클래스와 객체
2. 필드, 메소드, 프로퍼티
3. 인덱서, 델리게이트, 이벤트
4. 연산자 중복과 구조체

1. 클래스와 객체

1) 클래스

(1) 클래스

- C# 프로그래밍의 기본 단위
  → 재사용성(reusability)
  → 이식성(portability)
  → 유연성(flexibility) 증가

- 객체를 정의하는 탬플릿
  → 객체의 구조와 행위를 정의하는 방법

- 자료 추상화(Data abstraction)의 방법

(2) 클래스의 구조

속성
필드

행위
메소드

이외에 상수와 프로퍼티, 인덱서, 연산자 중복, 이벤트 클래스형이나 델리게이트형과 같은 자료형이 클래스 안에 포함될 수 있음

1. 클래스와 객체

1) 클래스

(3) 클래스의 선언 형태

- 크게 필드 계통과 메소드 계통으로 구분함

public, internal, abstract, static, sealed

```
[class-modifier] class ClassName {
  //member declarations
}
```

필드, 메소드, 프로퍼티, 인덱서, 연산자 중복, 이벤트

(4) 수정자

- 수정자(modifier) : 부가적인 속성을 명시하는 방법
- 클래스 수정자

수정자	설명
public	▪ 다른 프로그램에서 사용 가능
internal	▪ 같은 프로그램에서만 사용 가능 ▪ 수정자가 생략된 경우
static	▪ 정적 클래스, 클래스의 모든 멤버가 정적 멤버 ▪ 객체 단위로 존재하는 것이 아니라 클래스 단위로 존재
partial	▪ 부분 클래스 ▪ 여러 파일에 나누어서 같은 이름의 클래스를 정의
abstract, sealed	** 파생 클래스에서 설명*
protected, private	▪ 친구외에는 참조 불가 (외부에서 참조 불가)
new	▪ 중첩 클래스에서 사용되며 베이스 클래스의 멤버를 숨김

1) 클래스

(4) 수정자

- 클래스 선언의 예

예

```
class Fraction {                        // 분수 클래스
  int numerator;                        // 분자 필드
  int denominator;                      // 분모 필드
  public Fraction Add(Fraction f) {/* ... */}
     // 덧셈 메소드
  public static Fraction operator + (Fraction f1, Fraction f2)
     // 덧셈 연산자
  {/* ... */}
  public void PrintFraction() {/* ... */} // 출력 메소드
}
```

2) 객체

(1) 객체선언

- 클래스형의 변수 선언

객체를 선언하는 방법 클래스 이름을 쓰고 그에 해당하는 객체들을 나열

➡ 일반적으로 프로그래밍 언어에서 변수를 선언하는 것과 같음

예 Fraction f1, f2;

~ f1, f2 – 객체를 참조(reference)하는 변수 선언

(2) 객체생성

- 객체를 생성하는 법 : new 연산자와 함께 생성자를 명시하는 형태로 기술

```
f1 = new Fraction();
Fraction f1 = new Fraction();
```

1. 클래스와 객체

2) 객체

(3) 생성자

- 객체를 생성할 때 객체의 초기화를 위해 자동으로 호출되는 루틴
- 클래스와 동일한 이름을 갖는 메소드
- 객체 생성 시에 필요한 초기 값을 매개변수로 가질 수 있음

(4) 객체의 멤버 참조

- 객체 이름과 멤버 사이에 멤버 접근 연산자인 점 연산자(dot operator) 사용

예

~ 필드 참조: f1.numerator
~ 메소드 참조: f1.Add(f2)

멤버의 참조 형태

```
objectName.MemberName
```

2. 필드, 메소드, 프로퍼티

1) 필드

(1) 필드란

- 필드(field) : 클래스의 형태에서 객체의 구조를 기술하는 자료 부분

변수들의 선언으로 구성

(2) 필드 선언 형태

선언 형태

```
[field-modifier] DataType fieldNames;
```

예

```
int anInterger, anotherInteger;
public string usage;
static long idNum = 0;
public static readonly double earthWeight =
5.97e24;
```

필드의 이름은 소문자로 시작하는 것이 관례

2. 필드, 메소드, 프로퍼티

1) 필드

(3) 접근 수정자

- 다른 클래스에서 필드의 접근 허용 정도를 나타내는 속성

접근 수정자	동일 클래스	파생 클래스	네임스페이스	모든 클래스
private	O	X	X	X
protected	O	O	X	X
internal	O	X	O	X
protected internal	O	O	O	X
public	O	O	O	O

- 접근수정자의 선언 예

예

```
private int privateField;          // private
int noAccessModifier;              // private
protected int protectedField;      // protected
internal int internalField;        // internal
protected internal int piField;    // protected internal
public int publicField;            // public
```

1) 필드

(3) 접근 수정자(Access Notifier)

- Private

→ 정의된 클래스 내에서만 필드 접근 허용
→ 접근 수정자가 생략된 경우

예

```
class PrivateAccess {
  private int iamPrivate;
  int iamAlsoPrivate;
  // ...
}
class AnotherClass {
  void AccessMethod() {
    PrivateAccess pa = new PrivateAccess();
    pa.iamPrivate = 10;     // 에러
    pa.iamAlsoPrivate = 10; // 에러
  }
}
```

2. 필드, 메소드, 프로퍼티

1) 필드

(3) 접근 수정자(Access Notifier)

- public

→ 모든 클래스 및 네임스페이스에서 자유롭게 접근

예

```
class PublicAccess {
  public int iamPublic;
  // ...
}
class AnotherClass {
  void AccessMethod() {
    PublicAccess pa = new PublicAccess();
    pa.iamPublic = 10; // OK
  }
}
```

1) 필드

(3) 접근 수정자(Access Notifier)

- Internal : 같은 네임스페이스 내에서 자유롭게 접근
- Protected : 파생 클래스에서만 참조 가능
- protected internal 혹은 internal protected : 파생 클래스와 동일 네임스페이스 내에서도 자유롭게 접근

(4) new / static 수정자

- new

  → 상속 계층에서 상위 클래스에서 선언된 멤버를 하위 클래스에서 새롭게 재정의하기 위해 사용

- static

  → 정적 필드 (static field)
  → 클래스 단위로 존재
  → 생성 객체가 없는 경우에도 존재하는 변수
  → 정적 필드의 참조 형태

선언 형태

```
ClassName.staticField
```

1) 필드

(5) readonly / const 수정자

- readonly

  → 읽기 전용 필드
  → 값이 변할 수 없는 속성
  → 실행 중에 값이 결정

- const

  → 값이 변할 수 없는 속성
  → 컴파일 시간에 값이 결정
  → 상수 멤버의 선언 형태

선언 형태

```
[const-modifiers] const DataType constNames;
```

2) 메소드

(1) 메소드란?

- 메소드: 객체의 행위를 기술하는 방법

  → 객체의 상태를 검색하고 변경하는 작업

  → 특정한 행동을 처리하는 프로그램 코드를 포함하고 있는 함수의 형태

- 메소드 선언 예

예

```
class MethodExample {
  int SimpleMethod() {
    //...
  }
  public void EmptyMethod() {}
}
```

2) 메소드

(2) 메소드 수정자

- 메소드수정자 : 총 11개
- 접근수정자:

public | protected | internal | private

- Static

→ 정적 메소드
→ 전역 함수와 같은 역할
→ 정적 메소드는 해당 클래스의 정적 필드 또는 정적 메소드만 참조 가능
→ 정적 메소드 호출 형태 객체의 상태를 검색하고 변경하는 작업
→ 특정한 행동을 처리하는 프로그램 코드를 포함하고 있는 함수의 형태

선언 형태

```
ClassName.MethodName();
```

2) 메소드

(2) 메소드 수정자

- abstract/extern

  → 메소드 몸체 대신에 세미콜론(;)이 나옴

abstract	extern
메소드가 하위 클래스에 정의	메소드가 외부에 정의

- 이외의 메소드 수정자

  → new
  → virtual
  → override
  → sealed

2) 메소드

(3) 매개변수

- 매개변수 : 메소드 내에서만 참조될 수 있는 지역 변수

- 매개변수의 종류

  → 형식 매개변수 (formal parameter):
  메소드를 정의할 때 사용하는 매개변수

  → 실 매개변수 (actual parameter) :
  메소드를 호출할 때 사용하는 매개변수

- 매개변수의 자료형

  기본형 | 참조형

예

```
void parameterPass(int i, Fraction f) {
  // ...
}
```

2. 필드, 메소드, 프로퍼티

2) 메소드

(3) 매개변수

- this 지정어 : 자기 자신의 객체를 가리키는 특별한 포인터

클래스 필드와 매개변수를 구별하기 위함

- this 지정어 예

예

```
class Fraction {
  int numerator,
  denominator;

public Fraction(int numerator, int denominator) {
  this.numerator = numerator;
  this.denominator = denominator;
  }
}
```

필드로 선언된 numerator와 denumerator에 값을 할당함

메소드 안에서 사용되는 모든 필드와 메소드 이름 앞에는 내부적으로 this가 붙어 있음!

2) 메소드

(4) 매개변수 전달

- 매개변수 전달 : 메소드 호출 시에 실 매개변수가 형식 매개변수로 전달되는 것

  → 전달된 실 매개변수가 호출된 메소드 내에서 사용

- C#에서의 매개변수 전달 방법

  → 값 호출 (Call by value) : 실 매개변수의 값이 형식 매개변수로 전달

  → 참조 호출 (Call by reference)

  - 주소 호출(call by address)
  - 실 매개변수의 주소가 형식 매개변수로 전달
  - C#에서 제공하는 방법
    - 매개변수 수정자 이용
    - 객체 참조를 매개변수로 사용

2) 메소드

(4) 매개변수 전달

- 매개변수수정자

  → ref: 매개변수가 전달될 때 반드시 초기화
  → out: 매개변수가 전달될 때 초기화하지 않아도 됨

- 예제 : CallByValueApp.cs

```
using System;
class CallByReferenceApp {
  static void Swap(int x, int y) {
    int temp;
    temp = x; x = y; y = temp;
    Console.WriteLine("  Swap: x = {0}, y = {1}", x, y);
  }
  public static void Main() {
    int x = 1, y = 2;
    Console.WriteLine("Before: x = {0}, y = {1}", x, y);
    Swap(x, y);
    Console.WriteLine(" After: x = {0}, y = {1}", x, y);
  }
}
```

실행 결과

```
Before: x = 1, y = 2
Swap: x = 2, y = 1
After: x = 1, y = 2
```

2) 메소드

(5) 매개변수 배열

- 매개변수배열(Parameter array)

  → 실 매개변수의 개수가 상황에 따라 가변적인 경우
  → 메소드를 정의할 때 형식 매개변수를 결정할 수 없음

- 배열, 정의, 호출 예:

매개변수 배열 정의 예

```
void ParameterArray1(params int[] args)  { /* ... */ }
void ParameterArray2(params object[] obj) { /* ... */ }
```

호출 예

```
ParameterArray1();
ParameterArray1(1);
ParameterArray1(1, 2, 3);
ParameterArray1(new int[] {1, 2, 3, 4});
```

2) 메소드

(5) 매개변수 배열

- 예제 : ParameterArrayApp.cs

```
using System;
class ParameterArrayApp {
  static void ParameterArray(params object[] obj) {
    for (int i = 0; i < obj.Length; i++)
    Console.WriteLine(obj[i]);
   }
  public static void Main() {
    ParameterArray(123, "Hello", true, 'A');
  }
}
```

실행 결과

```
123
Hello
True
A
```

2) 메소드

(6) 메인 메소드

- 메인메소드:C# 응용 프로그램의 시작점

실행 결과

```
public static void Main(string[] args) {
  // ...
}
```

→ 매개변수 - 명령어 라인으로부터 스트링 전달

→ 명령어 라인으로부터 스트링 전달 방법

```
c:\>실행 파일명  인수1  인수2  ...  인수n
```

→ args[0] = 인수1, args[1] = 인수2, args[n-1] = 인수n

2) 메소드

(6) 메인 메소드

- 예제 : CommandLineArgsApp.cs

```
using System;
class CommandLineArgsApp {
  public static void Main(string[] args) {
    for (int i = 0; i < args.Length; ++i)
      Console.WriteLine("Argument[{0}] = {1}", i, args[i]);
  }
}
```

실행 방법

```
C:\> CommandLineArgsApp 12 Medusa 5.26
```

실행 결과

```
Argument[0] = 12
Argument[1] = Medusa
Argument[2] = 5.26
```

2) 메소드

(7) 메소드 중복

- 시그너처(Signature) : 메소드를 구별하는데 쓰이는 정보

메소드 이름	매개변수의 개수	매개변수의 자료형	메소드 반환형 제외

- 메소드 중복(Method overloading)

  → 메소드의 이름은 같은데 매개변수의 개수와 형이 다른 경우

  → 호출 시 컴파일러에 의해 메소드 구별

예

```
void SameNameMethod(int i) {/* ... */}        // 첫 번째 형태
void SameNameMethod(int i, int j) {/* ... */} // 두 번째 형태
```

2) 메소드

(7) 메소드 중복

- 예제 : MethodOverloadingApp.cs

```
using System;
class MethodOverloadingApp {
  void SomeThing() {
    Console.WriteLine("SomeThing() is called.");
  }
  void SomeThing(int i) {
    Console.WriteLine("SomeThing(int) is called.");
  }
  void SomeThing(int i, int j) {
    Console.WriteLine("SomeThing(int,int) is called.");
  }
  void SomeThing(double d) {
    Console.WriteLine("SomeThing(double) is called.");
  }
  public static void Main() {
    MethodOverloadingApp obj = new MethodOverloadingApp();
    obj.SomeThing();              obj.SomeThing(526);
    obj.SomeThing(54, 526);       obj.SomeThing(5.26);
  }
}
```

실행 결과

```
SomeThing() is called.
SomeThing(int) is called.
SomeThing(int,int) is called.
SomeThing(double) is called.
```

2) 메소드

(8) 생성자(Constructor)

- 생성자(Constructor)

  → 객체가 생성될 때 자동으로 호출되는 메소드
  → 클래스 이름과 동일하며 반환형을 갖지 않음
  → 주로 객체를 초기화하는 작업에 사용
  → 생성자 중복 가능

- 생성자예

```
class Fraction {
  // ....
  public Fraction(int a, int b) { // 생성자
    numerator = a;
    denominator = b;
  }
}
// ...
Fraction f = new Fraction(1, 2);
```

2) 메소드

(8) 생성자(Constructor)

- 정적생성자(Static constructor)

  → 수정자가 static으로 선언된 생성자
  → 매개변수와 접근 수정자를 가질 수 없음
  → 클래스의 정적 필드를 초기화할 때 사용
  → Main() 메소드보다 먼저 실행

- 정적필드초기화방법

  → 정적 필드 선언과 동시에 초기화
  → 정적 생성자 이용

2) 메소드

(8) 생성자(Constructor)

- 예제 : StaticConstructorApp.cs

```
using System;
class StaticConstructor {
  static int staticWithInitializer = 100;
  static int staticWithNoInitializer;
  static Constructor() { // 매개변수와 접근 수정자를 가질 수 없다.
    staticWithNoInitializer = staticWithInitializer + 100;
  }
  public static void PrintStaticVariable() {
    Console.WriteLine(
      "field 1 = {0}, field 2 = {1}",
      staticWithInitializer,
      staticWithNoInitializer
    );
  }
}
class StaticConstructorApp {
  public static void Main() {
    StaticConstructor.PrintStaticVariable();
  }
}
```

실행 결과

```
field 1 = 100, field 2 = 200
```

2) 메소드

(9) 소멸자(Destructor)

- 소멸자(Destructor) 란? : 클래스의 객체가 소멸될 때 필요한 행위를 기술한 메소드

> 소멸자의 이름은 생성자와 동일하나 이름 앞에 ~(tilde)를 붙임

- Finalize() 메소드

→ 컴파일 시 소멸자를 Finalize() 메소드로 변환해서 컴파일
→ Finalize() 메소드 재정의할 수 없음
→ 객체가 더 이상 참조되지 않을 때 GC(Garbage Collection)에 의해 호출

- Dispose() 메소드

→ CLR에서 관리되지 않은 자원을 직접 해제할 때 사용
→ 자원이 스코프를 벗어나면 즉시 시스템에 의해 호출

3) 프로퍼티

(1) 프로퍼티란?

- 클래스의 private 필드를 형식적으로 다루는 일종의 메소드
- 셋-접근자 : 값을 지정
- 겟-접근자로 : 값을 참조
- 겟-접근자 혹은 셋 : 접근자만 정의할 수 있음

(2) 프로퍼티의 정의 형태

정의 형태

```
[property modifiers] returnType PropertyName {
  get {
    // get-accessor body
  }
  set {
    // set-accessor body
  }
}
```

3) 프로퍼티

(3) 프로퍼티 수정자

- 수정자의 종류와 의미는 메소드와 모두 동일
- 총 11개 → 접근 수정자(4개)
  - → new
  - → static
  - → virtual
  - → sealed
  - → override
  - → abstract
  - → extern

(4) 프로퍼티의 동작

- 필드처럼 사용되지만, 메소드처럼 동작
- 배정문의 왼쪽에서 사용되면 셋-접근자 호출
- 배정문의 오른쪽에서 사용되면 겟-접근자 호출

3) 프로퍼티

(5) 예제 : PropertyApp.cs

```
using System;
class Fraction {
  private int numerator;
  private int denominator;
  public int Numerator {
    get { return numerator; }
    set { numerator = value; }
  }
  public int Denominator {
    get { return denominator; }
    set { denominator = value;}
  }
  override public string ToString() {
    return (numerator + "/" + denominator);
  }
}
```

```
class PropertyApp {
  public static void Main() {
    Fraction f = new Fraction();
    int i;
    f.Numerator = 1; // invoke set-accessor in Numerator
    i = f.Numerator + 1; // invoke get-accessor in Numerator
    f.Denominator = i; // invoke set-accessor in Denominator
    Console.WriteLine(f.ToString());
  }
}
```

1) 인덱서

(1) 인덱서 (Indexer)

- 배열 연산자인 '[]'를 통해서 객체를 다룰 수 있도록 함
- 지정어 this를 사용하고, '[]'안에 인덱스로 사용되는 매개 변수 선언
- 겟-접근자 혹은 셋-접근자만 정의할 수 있음

(2) 인덱서의 수정자

- static만 사용할 수 없으며, 의미는 메소드와 모두 같음
- 총 10개
  → 접근 수정자(4개)
  → new
  → override
  → abstract
  → virtual
  → sealed
  → extern

1) 인덱서

(3) 인덱서의 정의 형태

정의 형태

```
[indexer-
modifiers] returnType this[parameterList]{
  set {
    // indexer body
  }
  get {
    // indexer body
  }
}
```

1) 인덱서

(4) 예제 : IndexerApp.cs

```
using System;
class Color {
  private string[] color = new string[5];
  public string this[int index]{
    get { return color[index]; }
    set { color[index] = value; }
  }
}
class IndexerApp {
  public static void Main() {
    Color c = new Color();
    c[0] = "WHITE"; c[1] = "RED";
    c[2] = "YELLOW"; c[3] = "BLUE";
    c[4] = "BLACK";
    for (int i = 0; i < 5; i++)
      Console.WriteLine("Color is " + c[i]);
    }
  }
```

실행 결과

```
Color is WHITE
Color is RED
Color is YELLOW
Color is BLUE
Color is BLACK
```

2) 델리게이트(Delegate)

(1) 개요

- 델리게이트(Delegate)

  → 메소드 참조 기법

  객체지향적 특징이 반영된 메소드 포인터

  이벤트와 스레드를 처리하기 위한 방법론

- 특징

  → 객체지향적-정적메소드 및 인스턴트 메소드 참조 가능

  → 타입안정적-델리게이트의 형태와 참조하고자하는 메소드의 형태는 항상 일치

  → 메소드참조-델리게이트 객체를 통하여 메소드를 호출

- 함수포인터(C/C++)

  → 메소드 참조 기법면에서 유리

  → 객체지향적이며 타입 안정적

2) 델리게이트(Delegate)

(2) 델리게이트의 정의

- 정의형태

정의 형태

```
[modifiers] delegate returnType DelegateName(parameterList);
```

- 수정자

→ 접근 수정자 : public, protected, internal, private

→ new

→ 클래스 밖에서는 public과 internal만 가능

(2) 델리게이트의 정의

- 델리게이트정의시주의점

→ 델리게이트 할 메소드의 메소드 반환형 및 매개변수의 개수, 반환형을 일치시켜야 함

- 델리게이트정의의예

```
delegate void SampleDelegate(int param); // 델리게이트 정의
class DelegateClass {
  public void DelegateMethod(int param) { // 델리게이트할 메소드
    // ...
  }
}
```

2) 델리게이트(Delegate)

(3) 델리게이트 객체 생성

- 델리게이트를 사용하기 위해서는 델리게이트 객체를 생성하고 대상 메소드를 연결해야 함

→ 해당 델리게이트의 매개변수로 메소드의 이름을 명시
→ 델리게이트 객체에 연결할 수 있는 메소드는 형태가 동일하면 인스턴스 메소드뿐만 아니라 정적 메소드도 가능

- 델리게이트 생성(인스턴스 메소드)

→ 델리게이트할 메소드가 포함된 클래스의 객체를 먼저 생성
→ 정의된 델리게이트 형식으로 델리게이트 객체를 생성
→ 생성된 델리게이트를 통하여 연결된 메소드의 호출

- 델리게이트 객체 생성 예

```
DelegateClass obj = new DelegateClass();
SampleDelegate sd = new SampleDelegate(obj.DelegateMethod);
```

(4) 델리게이트 객체 호출

- 델리게이트 객체 호출

→ 델리게이트 객체의 호출은 일반 메소드의 호출과 동일

→ 델리게이트를 통하여 호출할 메소드가 매개변수를 갖는다면 델리게이트를 호출하면서 ( )안에 매개변수를 기술

2) 델리게이트(Delegate)

(4) 델리게이트 객체 호출

- 예제 : DelegateCallApp.cs

```
using System;
delegate void DelegateOne(); // delegate with no params
delegate void DelegateTwo(int i); // delegate with 1 param
class DelegateClass {
  public void MethodA() {
    Console.WriteLine("In the DelegateClass.MethodA ...");
  }
  public void MethodB(int i) {
    Console.WriteLine("DelegateClass.MethodB, i = " + i);
  }
}
class DelegateCallApp {
  public static void Main() {
    DelegateClass obj = new DelegateClass();
    DelegateOne d1 = new DelegateOne(obj.MethodA);
    DelegateTwo d2 = new DelegateTwo(obj.MethodB);
    d1(); // invoke MethodA() in DelegateClass
    d2(10); // invoke MethodB(10) in DelegateClass
  }
}
```

실행 결과

```
In the DelegateClass.MethodA ...
DelegateClass.MethodB, i = 10
```

2) 델리게이트(Delegate)

(5) 멀티캐스트(Multicast)

- 멀티캐스트(Multicast)

→ 하나의 델리게이트 객체에 형태가 동일한 여러 개의 메소드를 연결하여 사용 가능

→ C# 언어는 델리게이트를 위한 +와 – 연산자(메소드 추가/제거)를 제공

멀티캐스트 델리게이션 (multicast delegation)

- 델리게이트 연산을 통해 하나의 델리게이트 객체에 여러 개의 메소드가 연결되어 있는 경우, 델리게이트 호출을 통해 연결된 모든 메소드를 한번에 호출
- 델리게이트를 통하여 호출되는 순서는 등록된 순서와 동일

2) 델리게이트(Delegate)

(5) 멀티캐스트(Multicast)

- 예제 : MultiCastApp.cs

```
using System;
delegate void MultiCastDelegate();
class Schedule {
  public void Now() {
    Console.WriteLine("Time : " + DateTime.Now.ToString());
  }
  public static void Today() {
    Console.WriteLine("Date : " + DateTime.Today.ToString());
  }
}
class MultiCastApp {
  public static void Main() {
    Schedule obj = new Schedule();
    MultiCastDelegate mcd = new MultiCastDelegate(obj.Now);
    mcd += new MultiCastDelegate(Schedule.Today);
    mcd();
  }
}
```

실행 결과

```
Time : 2020-06-11 오후 12:05:30
Date : 2020-06-11 오전 12:00:00
```

3) 이벤트

(1) 이벤트란?

- 이벤트

  → 사용자 행동에 의해 발생하는 사건
  → 어떤 사건이 발생한 것을 알리기 위해 보내는 메시지
  → C#에서는 이벤트 개념을 프로그래밍 언어 수준에서 지원

- 이벤트 처리기(event handler)

  → 발생한 이벤트를 처리하기 위한 메소드

- 이벤트-주도 프로그래밍(event-driven programming)

  → 이벤트와 이벤트 처리기를 통하여 객체에 발생한 사건을 다른 객체에 통지하고 그에 대한 행위를 처리하도록 시키는 구조를 가짐

  → 각 이벤트에 따른 작업을 독립적으로 기술

  → 프로그램의 구조가 체계적/구조적이며 복잡도를 줄일 수 있음

3) 이벤트

(2) 이벤트의 정의

- 정의형태

정의 형태

```
[event-modifier] event DelegateType EventName;
```

- 수정자

→ 접근 수정자
→ new, static, virtual, sealed, override, abstract, extern
→ 이벤트 처리기는 메소드로 배정되기 때문에 메소드 수정자와 종류/의미가 같음

- 이벤트 정의 순서

→ 이벤트 처리기의 형태와 일치하는 델리게이트를 정의
(또는 System.EventHandler 델리게이트를 사용)
→ 델리게이트를 이용하여 이벤트를 선언
(미리 정의된 이벤트인 경우에는 생략)
→ 이벤트 처리기를 작성
→ 이벤트에 이벤트 처리기를 등록
→ 이벤트를 발생 (미리 정의된 이벤트는 사용자 행동에 의해 이벤트가 발생)

3) 이벤트

(2) 이벤트의 정의

- 이벤트의정의순서

이벤트가발생되면등록된메소드가호출되어
이벤트를처리

→ 미리 정의된 이벤트 발생은 사용자의 행동에 의해서 발생

→ 사용자 정의 이벤트인 경우에는 명시적으로 델리게이트 객체를 호출함으로써 이벤트 처리기를 작동

- 예제 : EventHandlingApp.cs

```
using System;
public delegate void MyEventHandler() // ① 이벤트를 위한 델리게이트 정의
class Button {
  public event MyEventHandler Push; // ② 이벤트 선언
  public void OnPush() {
    if (Push != null)
      Push(); // ⑤ 이벤트 발생
    }
  }
class EventHandlerClass {
  public void MyMethod() { // ③ 이벤트 처리기 작성
    Console.WriteLine("In the EventHandlerClass.MyMethod ...");
  }
}
```

3) 이벤트

(2) 이벤트의 정의

- 예제 : EventHandlingApp.cs

```
class EventHandlingApp {
  public static void Main() {
    Button = new Button();
    EventHandlerClass obj = new EventHandlerClass();
    button.Push += new MyEventHandler(obj.MyMethod); // ④ 등록
    button.OnPush();
  }
}
```

실행 결과

```
In the EventHandlerClass.MyMethod ...
```

3) 이벤트

(2) 이벤트의 정의

- 이벤트처리기등록

→ 델리게이트 객체에 메소드를 추가/삭제하는 방법과 동일
→ 사용 연산자

=	이벤트 처리기 등록
+	이벤트 처리기 추가
-	이벤트 처리기 제거

```
Event = new DelegateType(Method); // 이벤트 처리기 등록
Event += new DelegateType(Method); // 이벤트 처리기 추가
Event -= new DelegateType(Method); // 이벤트 처리기 제거
```

3) 이벤트

(3) 이벤트의 활용

- C# 언어에서의 이벤트 사용

→ 프로그래머가 임의의 형식으로 델리게이트를 정의하고 이벤트를 선언할 수 있도록 허용
→.NET 프레임워크는 이미 정의된 System.EventHandler 델리게이트를 이벤트에 사용하는 것을 권고
→ System.EventHandler

선언 형태

```
delegate void EventHandler(object sender, EventArgs e);
```

- 이벤트와 윈도우 환경

→ 이벤트는 사용자와 상호작용을 위해 주로 사용
→ 윈도우 프로그래밍 환경에서 사용하는 폼과 수많은 컴포넌트와 컨트롤에는 다양한 종류의 이벤트가 존재
→ 프로그래머로 하여금 적절히 사용할 수 있도록 방법론을 제공

4. 연산자 중복과 구조체

1) 연산자 중복

(1) 연산자 중복 (Operator overloading)이란?

- 연산자중복의의미

→ 시스템에서 제공한 연산자를 재정의 하는 것
→ 클래스만을 위한 연산자로서 자료 추상화가 가능
→ 문법적인 규칙은 변경 불가 (연산 순위나 결합 법칙 등)

- 연산자중복이가능한연산자

종류	연산자
단항	+, -, !, ~, ++, --, true, false
이항	+, -, *, /, %, &, |, ^, <<, >>, ==, !=, <, >, <=, >=
형 변환	변환하려는 자료형 이름

- 연산자중복방법

→ 수정자는 반드시 public static
→ 반환형은 연산자가 계산된 결과의 자료형
→ 지정어 operator 사용, 연산기호로는 특수 문자 사용

- 연산자중복정의형태

선언 형태

```
public static[extern] returnType operator op(par
ameter1[, parameter2]) {
  // ... operator overloading body ...
}
```

1) 연산자 중복

(2) 연산자 중복의 정의 규칙

- 매개변수 형과 반환형 규칙

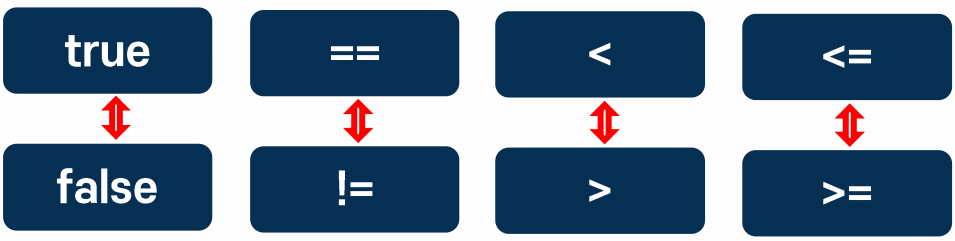
연산자	매개변수 형과 반환형 규칙
단항 +, -, !, ~	매개변수의 형은 자신의 클래스, 복귀형은 모든 자료형이 가능함
++ / --	매개변수의 형은 자신의 클래스, 복귀형은 자신의 클래스이거나 파생 클래스이어야 함
true / false	매개변수의 형은 자신의 클래스, 복귀형은 bool 형이어야 함
shift	첫 번째 매개변수의 형은 클래스, 두 번째 매개변수의 형은 int 형, 복귀형은 모든 자료형이 가능함
이항	shift 연산자를 제외한 이항 연산자인 경우, 두 개의 매개변수 중 하나는 자신의 클래스이며, 복귀형은 모든 자료형이 가능함

4. 연산자 중복과 구조체

1) 연산자 중복

(2) 연산자 중복의 정의 규칙

- 대칭적방식으로정의



- 형 변환연산자(type-conversion operator)

→ 클래스 객체나 구조체를 다른 클래스나 구조체 또는 C# 기본 자료형으로 변환
→ 사용자 정의형 변환(user-defined type conversion)

- 형 변환연산자문법구조

선언 형태

```
public static[extern] explicit operator type-name(parameter1)
또는
public static[extern] implicit operator type-name(parameter1)
```

1) 연산자 중복

(3) 연산자 중복의 예

- 예제 : OperatorOverloadingApp.cs

```
using System;
class Complex {
  private double realPart;  // 실수부
  private double imagePart; // 허수부
  public Complex(double rVal, double iVal) {
    realPart = rVal;
    imagePart = iVal;
  }
  public static Complex operator + (Complex x1, Complex x2) {
    Complex x = new Complex(0, 0);
    x.realPart = x1.realPart + x2.realPart;
    x.imagePart = x1.imagePart + x2.imagePart;
    return x;
  }
  override public string ToString() {
    return "(" + realPart + "," + imagePart + "i)";
  }
}
class OperatorOverloadingApp {
  public static void Main() {
  Complex c, c1, c2;
  c1 = new Complex(1, 2);
  c2 = new Complex(3, 4);
  c = c1 + c2;
  Console.WriteLine(c1 + " + " + c2 +
    " = " + c);
  }
}
```

실행 결과

```
(1,2i) + (3,4i) = (4,6i)
```

2) 구조체

(1) 구조체의 형태

- 구조체(struct)

→ 클래스와동일하게객체의구조와행위를정의하는방법

클래스 : 참조형

구조체: 값형

- 구조체의형태

선언 형태

```
[struct-modifiers] struct StructName {
  // member declarations
}
```

- 구조체의수정자

public | protected | internal | private | new

2) 구조체

(2) 구조체와 클래스의 차이점

- 클래스는 참조형이고 구조체는 값형이다.
- 클래스 객체는 힙에 저장되고 구조체 객체는 스택에 저장된다.
- 배정 연산에서 클래스는 참조가 복사되고 구조체는 내용이 복사된다.
- 구조체는 상속이 불가능하다.
- 구조체는 소멸자를 가질 수 없다.
- 구조체의 멤버는 초기값을 가질 수 없다.

Summary

1. 클래스와 객체

- C#의 클래스는 C# 프로그램을 구성하는 기본 단위이며 구조를 나타내는 필드와 행위를 나타내는 메소드로 구성됨
- 일단 클래스가 정의되면 그 클래스로부터 객체를 선언할 수 있으며 객체를 선언하는 방법은 클래스 이름을 쓰고 그에 해당하는 객체들을 나열함

2. 필드, 메소드, 프로퍼티

- 클래스의 형태에서 객체의 구조를 기술하는 자료 부분은 변수들의 선언으로 구성되며 각각을 클래스의 필드 부름
- 메소드란 객체의 행위를 기술하는 방법으로 객체의 상태를 검색하고 변경하는 작업, 그리고 특정한 행동을 처리하는 프로그램 코드를 포함하고 있는 함수의 형태임
- 프로퍼티란 클래스의 private 필드를 형식적으로 다루는 일종의 메소드로 간주함

3. 인덱서, 델리게이트, 이벤트

- 인덱서(Indexer)란 배열 연산자인 '[ ]'를 통해서 객체를 다룰 수 있도록 해주는 특별한 형태의 프로퍼티임
- 델리게이트는 메소드를 참조하기 위한 기법으로 이벤트와 스레드를 처리하는데 주로 사용됨
- 이벤트란 사용자 행동에 의해 발생하는 사건을 의미하며 어떤 사건이 발생한 것을 알리기 위해 보내는 메시지로 간주함

4. 연산자 중복과 구조체

- 연산자 중복이란 시스템에서 제공한 연산자를 특정 클래스만을 위한 연산 의미를 갖도록 재정의함
- 구조체란 클래스와 동일하게 객체의 구조와 행위를 정의하는 방법이며 클래스와의 차이점은 클래스는 참조형이고 구조체는 값형임