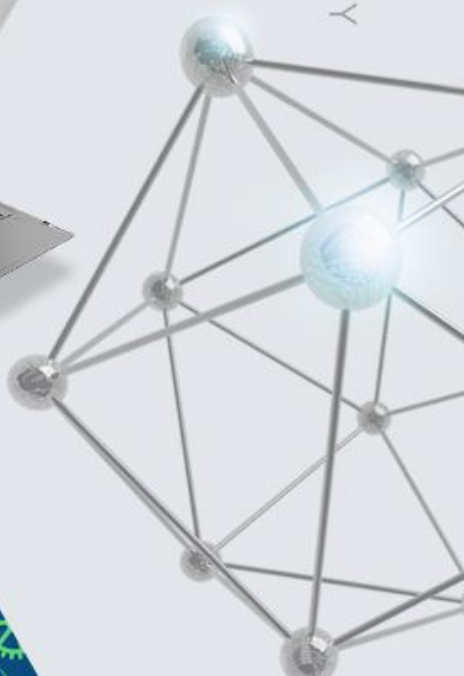
한국기술교육대학교
온라인평생교육원

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

고급 프로그래밍 기법



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.

고급 프로그래밍 기법

학/습/목/표

1. 제네릭을 이해하고 구현할 수 있다.
2. 애트리뷰트를 이해하고 구현할 수 있다.
3. 예외와 예외 처리를 이해하고 구현할 수 있다.
4. 스레드를 이해하고 구현할 수 있다.

학/습/내/용

1. 제네릭
2. 애트리뷰트
3. 예외
4. 스레드

1. 제네릭

1) 제네릭 클래스

(1) 정의

- 제네릭이란?

  → 변수의 형을 매개변수로 하여 클래스나 메소드의 알고리즘을 자료형과 무관하게 기술하는 기법

- 형 매개변수(Type Parameter)

  → 클래스 내의 필드나 메소드 선언시 자료형으로 사용함

  →'<'과'>' 사이에 형 매개변수의 이름을 기술 :

  <T1, T2, T3, ..., Tn>

(2) 장점

- 제네릭의 장점

  → 알고리즘의 재사용성을 높임

  → 자료형에 따른 프로그램의 중복을 줄임

  → 프로그램의 구조를 단순하게 만듦

(3) 형 매개변수를 가지는 클래스

- 형 매개변수(Type Parameter)를 가지는 클래스

  → 형 매개변수를 이용하여 필드나 지역변수에 사용함

  → 실제 형 정보는 객체 생성 시에 전달받음

1. 제네릭

1) 제네릭 클래스

(4) 형태

- 제네릭 클래스의 정의 형태

선언 형태

```
[modifiers] class ClassName<TypeParameters> {
  // ... class body
}
```

(5) 형식 매개변수 형

- 형식 매개변수 형(Formal Parameter Type)

→ 제네릭 클래스를 선언할 때 사용한 형 매개변수

→ `class SimpleGeneric <T> { ... }`

(6) 실제 형 인자

- 실제 형 인자(Actual Type Argument)

→ 제네릭 클래스에 대한 객체를 생성할 때 주는 자료형

→ `new SimpleGeneric <Int32> (10);`

1. 제네릭

1) 제네릭 클래스

(7) 예제

- 제네릭 클래스 사용의 예

```
using System;
class SimpleGeneric<T> {
 private T[] values;
 private int index;
 public SimpleGeneric (int len) { // Constructor
  values = new T[len];
  index = 0;
 }
 public void Add (params T[] args) {
  foreach (T e in args)
  values[ index++ ] = e;
 }
 public void Print() {
  foreach (T e in values)
  Console.Write(e + " ");
  Console.WriteLine();
 }
}
public class GenericClassExample {
 public static void Main() {
  SimpleGeneric<Int32> gInteger = new SimpleGeneric<Int32>(10);
  SimpleGeneric<Double> gDouble = new SimpleGeneric<Double>(10);
  gInteger.Add(1, 2);
  gInteger.Add(1, 2, 3, 4, 5, 6, 7);
  gInteger.Add(0);
  gInteger.Print();
  gDouble.Add(10.0, 20.0, 30.0);
  gDouble.Print();
 }
}
```

실행 결과

```
1 2 1 2 3 4 5 6 7 0
10 20 30 0 0 0 0 0 0 0
```

2) 제네릭 인터페이스

(1) 형 매개변수를 가지는 인터페이스

- 형 매개변수를 가지는 인터페이스

→ 형 매개변수의 선언 외에 일반 인터페이스를 구현하는 과정과 동일

(2) 형태

- 제네릭 인터페이스의 정의 형태

선언 형태

```
[modifiers] interface IName<TypeParameters> {
  // ... interface body
}
```

- 제네릭 인터페이스의 구현의 예

→ 인터페이스 정의, 구현 그리고 사용

2) 제네릭 인터페이스

(3) 예제

- 예제 : GenericInterfaceApp.cs

```
interface IGenericInterface<T> {
  void SetValue(T x);
  string GetValueType();
}
class GenericClass<T> : IGenericInterface<T> {
  private T value;
  public void SetValue(T x) {
    value = x;
  }
  public String GetValueType() {
    return value.GetType().ToString();
  }
}
```

- 예제 : GenericInterfaceApp.cs

```
public class GenericInterfaceApp {
  public static void Main() {
    GenericClass<Int32> gInteger =
      new GenericClass<Int32>();
    GenericClass<String> gString =
      new GenericClass<String>();

    gInteger.SetValue(10);
    gString.SetValue("Text");

    Console.WriteLine(gInteger.GetValueType ());
    Console.WriteLine(gString.GetValueType ());
  }
}
```

3) 제네릭 메소드

(1) 정의

- 제네릭 메소드(Generic Method) : 형 매개변수 (Type Patameter)를 갖는 메소드

(2) 형태

- 제네릭 메소드의 정의 형태

정의 형태

```
void Swap<DataType>
(DataType x, DataType y) {
  DataType temp = x;
  x = y;
  y = temp;
}
```

- 제네릭 메소드의 호출 예

예

```
Swap<int>(a, b);    // a, b: 정수형
Swap<double>(c, d); // c, d: 실수형
```

3) 제네릭 메소드

(3) 형 매개변수의 중첩

- 형 매개변수의 중첩

→ 제네릭 메소드의 형 매개변수의 이름과 제네릭 클래스의 형 매개변수 이름이 같은 경우

→ 서로 독립된 형 매개변수의 개념을 가짐

- 제네릭 클래스는 객체 생성 시에 형 매개변수를 전달받음
- 제네릭 메소드는 호출 시에 유추하여 형 매개변수가 결정됨

3) 제네릭 메소드

(4) 예제

- 예제 : GenericMethodApp.cs

```
using System;
class GenericMethodApp {
  static void Swap<DataType>(ref DataType x, ref DataType y) {
    DataType temp;
    temp = x; x = y; y = temp;
  }
  public static void Main() {
    int a = 1, b = 2; double c = 1.5, d = 2.5;
    Console.WriteLine("Before: a = {0}, b = {1}", a, b);
    Swap<int>(ref a, ref b); // 정수형 변수로 호출
    Console.WriteLine(" After: a = {0}, b = {1}", a, b);
    Console.WriteLine("Before: c = {0}, d = {1}", c, d);
    Swap<double>(ref c, ref d); // 실수형 변수로 호출
    Console.WriteLine(" After: c = {0}, d = {1}", c, d);
  }
}
```

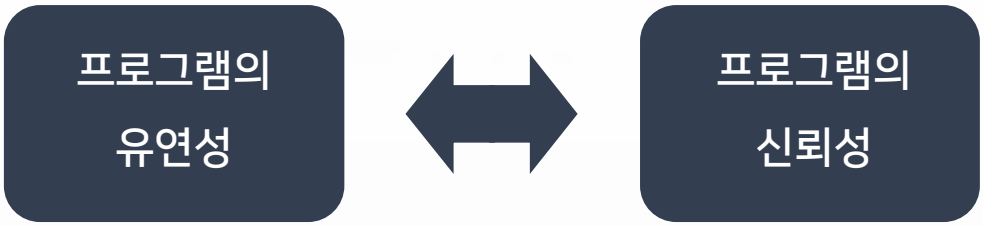
실행 결과

```
Before: a = 1, b = 2
After: a = 2, b = 1
Before: c = 1.5, d = 2.5
After: c = 2.5, d = 1.5
```

4) 형 매개변수의 제한

(1) 형 매개 변수의 제한

- 형 매개 변수의 범위



(2) 형 매개 변수의 제한 조건

- 형 매개변수의 제한 조건

→ 프로그램의 신뢰성을 증진하기 위해 제네릭에 전달 가능한 자료형의 범위를 제한할 필요가 있음

제한 조건	설명
where T : struct	T는 값형이어야 함
where T : class	T는 참조형이어야 함
where T : new ( )	T는 매개변수가 없는 생성자가 있어야 함
where T : MyClass	T는 MyClass의 파생 클래스이어야 함
where T : IMyInterface	T는 IMyInterface를 구현한 클래스이어야 함
where T : U	T는 U로부터 파생된 클래스이어야 함

4) 형 매개변수의 제한

(3) 형태

- 제네릭 클래스를 작성 시 한정 : where 키워드 사용

정의 형태

```
< T > where T : S
// S 형의 서브클래스형으로 제한한다.
```

4) 형 매개변수의 제한

(4) 예제

▪ 형 매개변수 사용 시 한정의 예제 : BoundedGenericApp.cs

```
using System;
class GenericType<T> where T : SystemException {
  private T value;
  GenericType(T v) { value = v; }
  override public String ToString() {
    return "Type is " + value.GetType();
  }
}
public class BoundedGenericApp {
  public static void Main() {
    GenericType<NullReferenceException> gNullEx =
    new GenericType<NullReferenceException>
(new NullReferenceException());
    GenericType<IndexOutOfRangeException> gIndexEx =
    new GenericType<IndexOutOfRangeException>
(new IndexOutOfRangeException());
  // GenericType<String> gString = new GenericType<String>
("Error"); 에러
    Console.WriteLine(gNullEx.ToString ());
    Console.WriteLine(gIndexEx.ToString ());
  }
}
```

실행 결과

```
Type is System.NullReferenceException
Type is System.IndexOutOfRangeException
```

2. 애트리뷰트

1) 애트리뷰트의 개요

(1) 특징

- 애트리뷰트의 특징

→ 어셈블리나 클래스, 필드, 메소드, 프로퍼티 등에 다양한 종류의 속성 정보를 추가하기 위해서 사용함

→ 어셈블리에 메타데이터(Metadata) 형식으로 저장되며, 이를 참조하는 .NET 프레임워크나 C# 또는 다른 언어의 컴파일러에 의해 다양한 용도로 사용함

(2) 형태

- 애트리뷰트의 정의 형태

정의 형태

```
[attribute AttributeName("positional_parameter", named_parameter = value, ...)]
```

(3) 종류

- 애트리뷰트의 종류

→ 표준 애트리뷰트(.NET 프레임 워크에서 제공)

→ 사용자 정의 애트리뷰트

2) 표준 애트리뷰트

(1) Conditional 애트리뷰트

- Conditional 애트리뷰트

→ 조건부 메소드를 작성할 때 사용함

→ C/C++ 언어에서 사용했던 전처리기 지시어를 이용하여 명칭을 정의(#define)함

→ System.Diagnostics를 사용해야 함

(2) Obsolete 애트리뷰트

- Obsolete 애트리뷰트

→ 앞으로 사용되지 않을 메소드를 표시하기 위해서 사용함

→ 해당 애트리뷰트를 가진 메소드를 호출할 경우 컴파일러는 컴파일 과정에서 애트리뷰트에 설정한 내용이 출력하는 경고를 발생함

3) 사용자 정의 애트리뷰트

(1) 특징

- 사용자 정의 애트리뷰트의 특징

→ System.Attribute 클래스에서 파생 : 이름의 형태 : XxxxAttribute

→ 정의한 애트리뷰트를 사용할 때는 이름에서 Attribute가 제외된 부분만을 사용함

→ 사용자 정의 애트리뷰트나 표준 애트리뷰트를 사용하기 위해서는 .NET 프레임워크가 제공하는 리플렉션 기능을 사용함

(2) 예제

- 사용자 정의 애트리뷰트의 예

정의한 예

```
// 사용자 정의 애트리뷰트를 정의한 예
public class AttributeNameAttribute : Attribute {
  // 생성자 정의
}
```

사용한 예

```
// ... 사용자 정의 애트리뷰트를 사용한 예
[AttributeName()]
```

3) 사용자 정의 애트리뷰트

(2) 예제

- 예제 : MyAttributesApp.cs

```
using System;
public class MyAttrAttribute: Attribute { // 속성 클래스
  public MyAttrAttribute(string message) { // 생성자
    this.message = message;
  }
  private string message;
  public string Message { // 프로퍼티
    get { return message; }
  }
}
[MyAttr ("This is Attribute test.")]
class MyAttributeApp {
  public static void Main() {
    Type = typeof (MyAttributeApp);
    object[] arr = type.GetCustomAttributes (typeof
(MyAttrAttribute), true);
    if (arr.Length == 0)
      Console.WriteLine ("This class has no custom attrs.");
    else {
      MyAttrAttribute ma = (MyAttrAttribute)arr[0];
      Console.WriteLine (ma.Message);
    }
  }
}
```

3. 예외

1) 예외의 개요

(1) 개요

- 예외(Exception)

  → 실행 시간에 발생하는 에러 (run-time error)

  → 프로그램의 비정상적인 종료

  → 잘못된 실행 결과

  → 메소드의 호출과 실행, 부정확한 데이터, 그리고 시스템 에러 등 다양한 상황에 의해 야기

- 예외 처리(Exception Handling)

  → 기대되지 않은 상황에 대해 예외를 발생함

  → 야기된 예외를 적절히 처리 (exception handler)

- 예외 처리를 위한 방법을 언어 시스템에서 제공

  → 응용프로그램의 신뢰성(reliability) 향상

  → 예외 검사와 처리를 위한 프로그램 코드를 소스에 깔끔하게 삽입

2) 예외의 정의

(1) 정의

- 예외 정의

→ 예외도 하나의 객체로 취급함

- 따라서, 먼저 예외를 위한 클래스를 정의하여야 함

→ 예외를 명시적으로 발생시키려면 예외를 처리하는 예외 처리기가 반드시 필요함

→ 예외에 관련된 메시지를 스트링 형태로 예외 객체에 담아 전달 가능함

(2) 예외 클래스

- 예외 클래스

→ 모든 예외는 형(type)이 Exception 클래스 또는 그의 파생 클래스들 중에 하나로부터 확장된 클래스의 객체

→ 일반적으로 프로그래머는 Exception 클래스의 파생 클래스인 ApplicationException 클래스를 확장하여 새로운 예외 클래스를 정의하여 사용

2) 예외의 정의

(3) 예제

- 예제 : UserExceptionApp.cs

```
using System;
class UserErrException: ApplicationException {
  public UserErrException (string s): base (s) {}
}
class UserException {
  public static void Main () {
    try {
      throw new UserErrException
        ("throw a exception with a message");
    } catch (UserErrException e) {
      Console.WriteLine (e.Message);
    }
  }
}
```

실행 결과

```
throw a exception with a message
```

2) 예외의 정의

(4) 시스템 정의 예외

- 시스템 정의 예외(System-defined Exception)

→ 프로그램의 부당한 실행으로 인하여 시스템에 의해 묵시적으로 일어나는 예외

→ SystemException클래스나 IOException 클래스로부터 확장된 예외

→ CLR에 의해 자동적으로 생성

→ 야기된 예외에 대한 예외 처리기의 유무를 컴파일러가 검사하지 않음 (Unchecked Exception)

(5) 프로그래머 정의 예외

- 프로그래머 정의 예외 (Programmer-defined Exception)

→ 프로그래머에 의해 의도적으로 야기되는 프로그래머 정의 예외

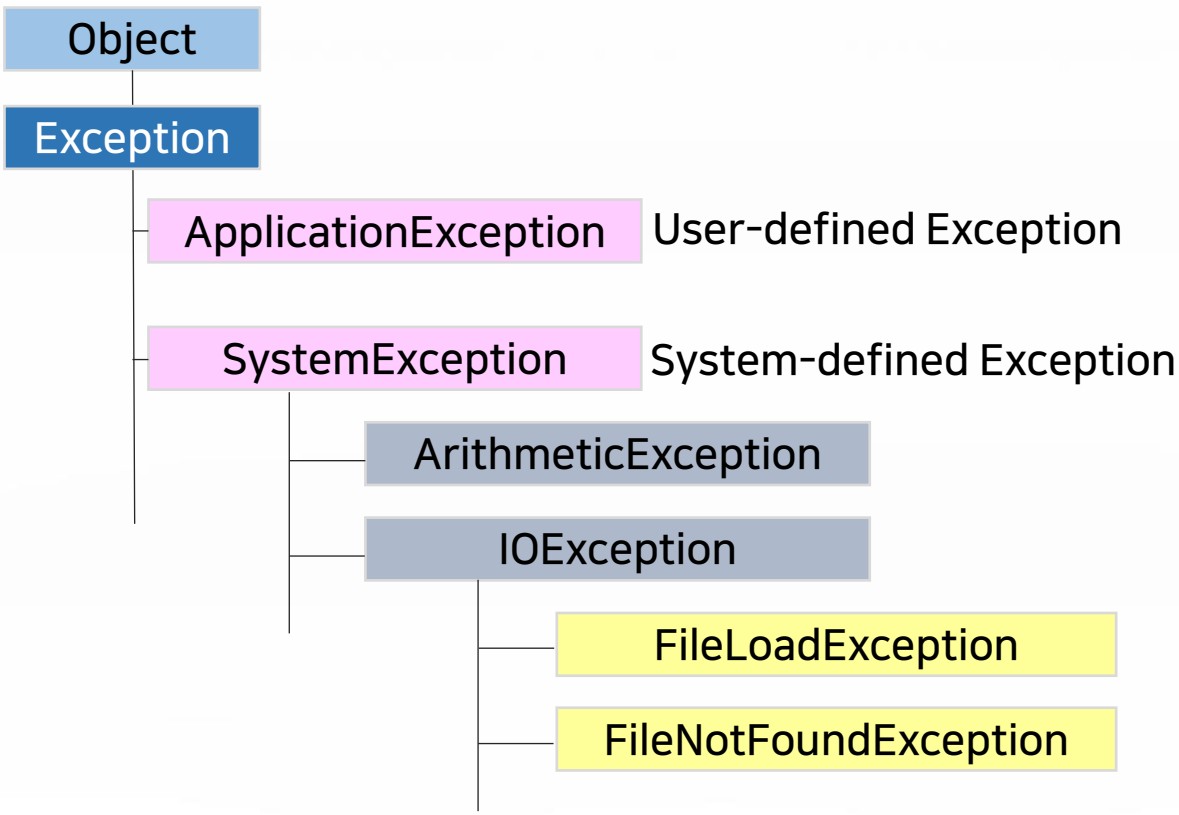
→ 프로그래머 정의 예외는 발생한 예외에 대한 예외 처리기가 존재하는지 컴파일러에 의해 검사(Checked Exception)

3. 예외

2) 예외의 정의

(6) 예외의 계층도

- 예외의 계층도



3. 예외

3) 예외 발생

(1) 묵시적 예외 발생

- 묵시적 예외 발생

→ 시스템 정의 예외로 CLR에 의해 발생

→ 시스템에 의해 발생되므로 프로그램 어디서나 발생 가능

→ 프로그래머가 처리하지 않으면 디폴트 예외 처리기

→ (Default Exception Handler)에 의해 처리

(2) 명시적 예외 발생

- 명시적 예외 발생

→ throw 문을 이용하여 프로그래머가 의도적으로 발생

정의 형태

```
throw ApplicationExceptionObject;
```

→ 프로그래머 정의 예외는 생성된 메소드 내부에 예외를 처리하는 코드 부분인 예외 처리기를 두어 직접 처리해야 함

3) 예외 발생

(3) 예제

- 예제 : UserExThrowApp.cs

```
using System;
class UserException : ApplicationException {}
class UserExThrowApp {
  static void Method() {
    throw new UserException();
  }
  public static void Main() {
    try {
      Console.WriteLine("Here: 1");
      Method();
      Console.WriteLine("Here: 2");
    } catch (UserException) {
      Console.WriteLine("User-defined Exception");
    }
  }
}
```

실행 결과

```
Here: 1
User-defined Exception
```

3. 예외

4) 예외 처리

(1) 예외 처리 구분

- 에러 처리 구문(try-catch-finally 구문)

  → 예외를 검사하고 처리해주는 문장

정의 형태

```
try {
  // ... "try 블록"
} catch (ExceptionType identifier) {
  // ... "catch 블록"
} catch (ExceptionType identifier) {
  // ... "catch 블록"
} finally {
  // ... "finally 블록"
}
```

try 블록 : 예외 검사

catch 블록 : 예외 처리

finally 블록 : 종결 작업, 예외 발생과 무관하게 반드시 실행

4) 예외 처리

(2) 예외 처리기 실행 순서

- 에러 처리기의 실행 순서

① try 블록 내에서 예외가 검사되고 또는 명시적으로 예외가 발생하면,

② 해당하는 catch 블록을 찾아 처리하고,

③ 마지막으로 finally 블록을 실행한다.

(3) Default 예외 처리기

- Default 예외 처리기

→ 시스템 정의 예외가 발생됐는데도 불구하고 프로그래머가 처리하지 않을 때 작동

→ 단순히 에러에 대한 메시지를 출력하고 프로그램을 종료하는 기능

4) 예외 처리

(4) 예제

- 예제 : FinallyClauseApp.cs

```
using System;
class FinallyClauseApp {
  static int count = 0;
  public static void Main() {
    while (true) {
      try {
        if (++count == 1) throw new Exception ();
        if (count == 3) break;
        Console.WriteLine(count + ") No exception");
     } catch (Exception) {
       Console.WriteLine(count + ") Exception thrown");
     } finally {
       Console.WriteLine(count + ") in finally clause");
     }
    } // end while
    Console.WriteLine("Main program ends");
  }
}
```

실행 결과

```
1) Exception thrown
1) in finally clause
2) No exception
2) in finally clause
3) in finally clause
Main program ends
```

5) 예외 전파

(1) 예외의 전파

- 예외의 전파

→ 호출한 메소드로 예외를 전파 (Propagation)하여 특정 메소드에서 모아 처리

- 예외 처리 코드의 분산을 막을 수 있음

→ 예외 전파 순서

- 예외를 처리하는 catch 블록이 없으면 호출한 메소드로 예외를 전파함
- 예외처리기를 찾을 때까지의 모든 실행은 무시

5) 예외 전파

(1) 예외의 전파

- 예외 전파 과정

```
using System;
namespace PropagateApp {
  class Propagate {
    void Orange() {
      int i = 25, j = 0;
      i = i / j;
    }
    void Apple() {
      Orange();
      Console.WriteLine("The End of Orange method");
    }
  }
  public static void Main() {
    Propagate p = new Propagate();
    p.Apple();
  }
}
```

DivideByZeroException

Default Exception Handler

5) 예외 전파

(1) 예외의 전파

- 예외 전파 과정

실행 결과

```
System.DivideByZeroException:
   at PropagateApp.Orange
() in c:\Csharp\chapter6\PropagateApp.cs: line 5
   at PropagateApp.Apple
() in c:\Csharp\chapter6\PropagateApp.cs: line 8
   at PropagateApp.Main
() in c:\Charp\chapter6\PropagateApp.cs: line 12
```

5) 예외 전파

(2) 예제

- 예제 : PropagateApp.cs

```
using System;
namespace PropagateApp {
  class Propagate {
    void Orange() {
      int i = 25, j = 0;
      i = i / j;
      Console.WriteLine("End of Orange method");
    }
    void Apple() {
      Orange();
      Console.WriteLine("End of Apple method");
    }
  }
  public static void Main() {
    Propagate p = new Propagate();
    try {
      p.Apple();
    } catch (ArithmeticException) {
      Console.WriteLine("ArithmeticException is processed");
    }
    Console.WriteLine("End of Main");
  }
}
```

3. 예외

5) 예외 전파

(3) 예외 전파 순서

- 예외 전파 순서

→ 예외를 처리하는 catch 블록이 없으면 호출한 메소드로 예외를 전파함

→ 예외처리기를 찾을 때까지의 모든 실행은 무시함

4. 스레드

1) 스레드의 개요

(1) 개요

- 순차 프로그램(Sequential Program)

  → 각 프로그램별로 시작, 순차적 실행 그리고 종료를 가짐

  → 프로그램의 실행 과정 중 오직 하나의 실행 점(Execution Point)을 갖고 있음

- 스레드

  → 순차 프로그램과 유사하게 시작, 실행, 종료의 순서를 가짐

  → 실행되는 동안에 한 시점에서 단일 실행 점을 가짐

  → 프로그램 내에서만 실행 가능함

  - 스레드는 프로그램 내부에 있는 제어의 단일 순차 흐름임(single sequential flow of control)
  - 단일 스레드 개념은 순차 프로그램과 유사함

1) 스레드의 개요

(1) 개요

- 멀티스레드 (Multithread) 시스템

→ 스레드가 하나의 프로그램 내에 여러 개 존재임

→ 공유 힙 (Shared Heap)과 공유 데이터 (Shared Data), 그리고 코드를 공유함으로써 문맥 전환 (Context Switching) 시 적은 부담을 가짐

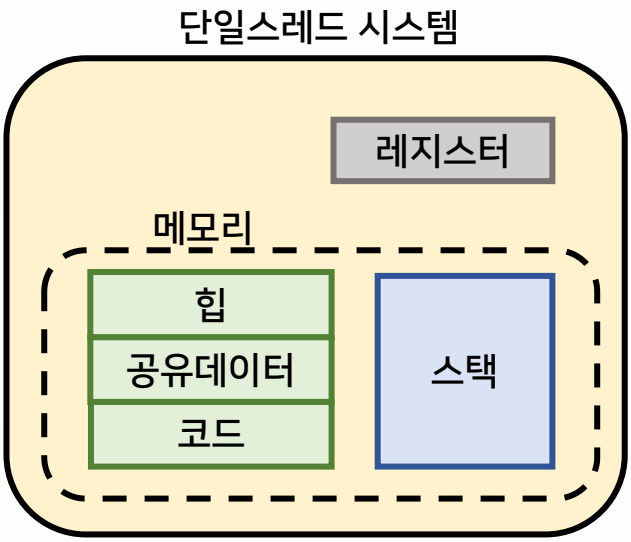
→ 한 개의 프로그램 내에서 동일 시점에 각각 다른 작업을 수행하는 여러 개의 스레드가 존재하므로 복잡한 문제들이 야기될 수 있음

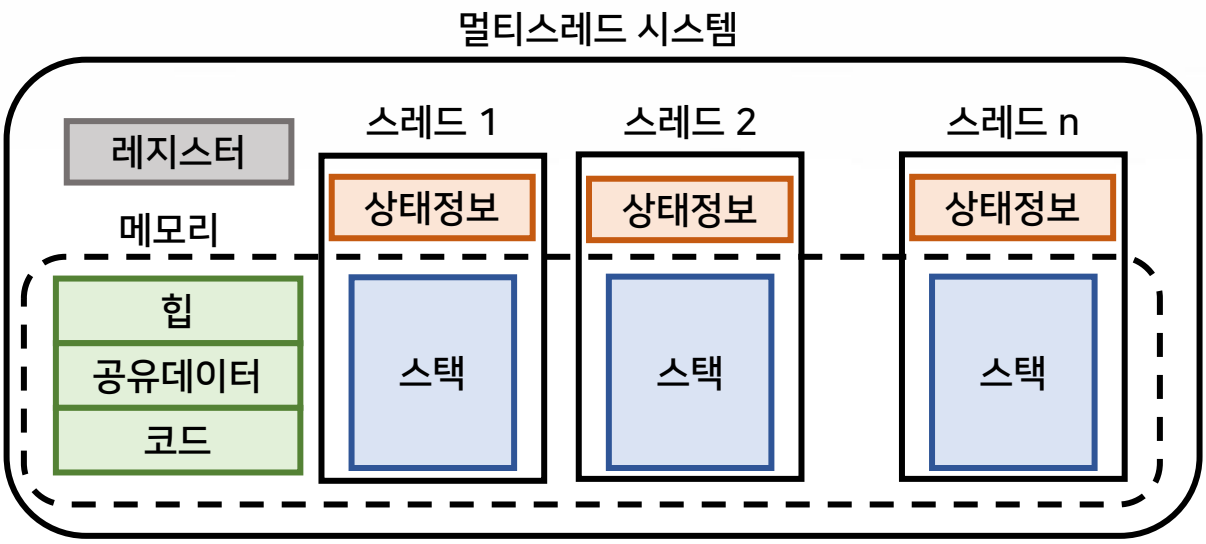
- C#에서는 언어 수준에서 스레드를 지원

1) 스레드의 개요

(1) 개요

- 단일스레드 시스템과 멀티스레드 시스템의 구조 비교





2) 스레드 프로그래밍

(1) C# 스레드 개념

- C#에서의 스레드 개념

  → 객체이며 스레드가 실행하는 단위는 메소드임

  → 스레드 객체를 위해 Thread 클래스를 제공함

  → 메소드 연결을 위해 ThreadStart 델리게이트를 제공함 (System.Threading)

(2) 스레드 프로그래밍 순서

- 스레드 프로그래밍의 순서

  → 스레드 몸체에 해당하는 메소드를 작성

  → 작성된 메소드를 ThreadStart 델리게이트에 연결

  → 생성된 델리게이트를 이용하여 스레드 객체를 생성

  → 스레드의 실행을 시작 (Start ( ) 메소드를 호출)

2) 스레드 프로그래밍

(3) 예제

- 예제 : SimpleThreadApp.cs

```
using System;
using System.Threading; // 반드시 포함 !!!
class SimpleThreadApp {
  static void ThreadBody() { // --- ①
    for (int i = 0; i < 5; i++) {
      Console.WriteLine(DateTime.Now.Second + " : " + i);
      Thread.Sleep(1000);
    }
  }
  public static void Main() {
    ThreadStart ts = new ThreadStart(ThreadBody); // --- ②
    Thread t = new Thread (ts); // --- ③
    Console.WriteLine("*** Start of Main");
    t.Start(); // --- ④
    Console.WriteLine("*** End of Main");
  }
}
```

실행 결과

```
*** Start of Main
*** End of Main
15 : 0
16 : 1
17 : 2
18 : 3
19 : 4
```

2) 스레드 프로그래밍

(4) 스레드 프로퍼티

- 스레드 프로퍼티

프로퍼티	설 명
Thread.CurrentThread	현재 실행 중인 스레드 객체를 반환한다.
Thread.IsAlive	스레드의 실행 여부를 반환한다.
Thread.Name	스레드의 이름을 지정하거나 반환한다.
Thread.IsBackground	스레드가 백그라운드 스레드인지의 여부를 반환한다.
Thread.ThreadState	스레드의 상태를 반환한다.
Thread.Priority	해당 스레드의 우선순위를 지정하거나 반환한다.

3) 스레드의 상태

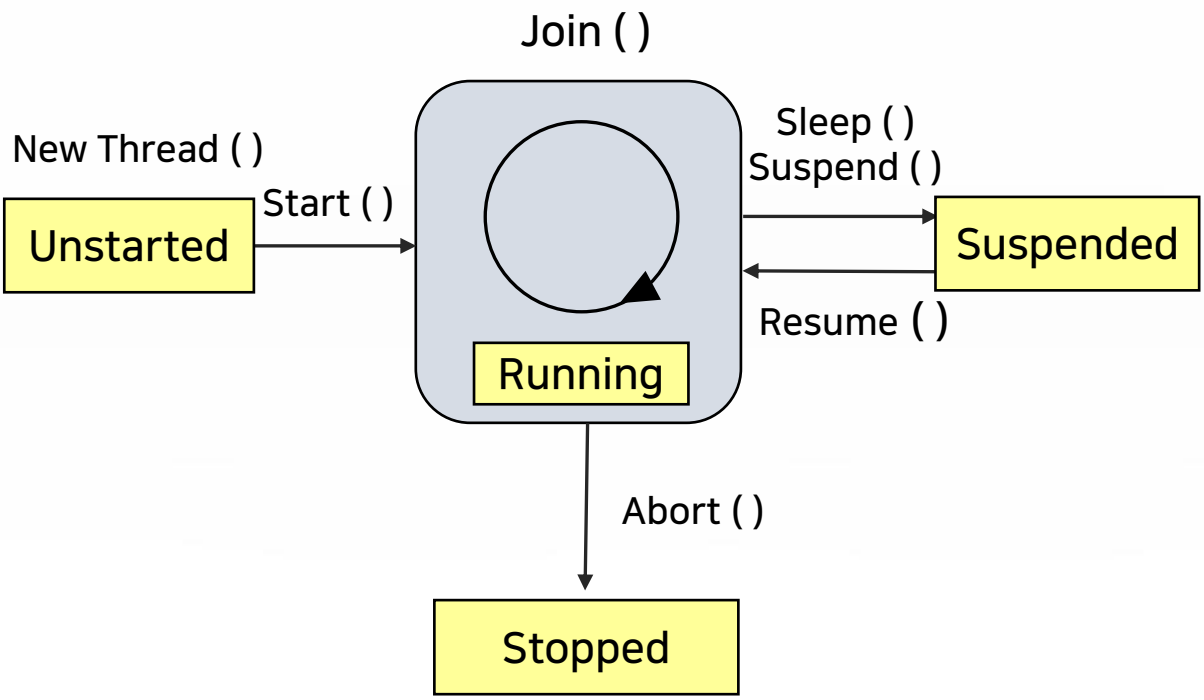
(1) 스레드의 상태

- 스레드의 상태도

→ 상태 : Unstarted, Running, Suspended, Stopped

→ 시작메소드 : Start ()

→ 종료조건 : 메소드의 종료, Abort ( ) 실행



4) 스레드의 스케쥴링

(1) 정의

- 스레드 스케쥴링의 정의

→ 실행 가능한 상태에 있는 여러 스레드의 실행 순서를 제어하는 것

(2) 특징

- 스레드 스케쥴링의 특징

→ 스레드의 스케쥴링에 따라 스레드의 작업 순서가 달라지며, 스레드의 우선순위 (priority)에 따라 실행 순서가 결정됨

→ 스레드가 생성될 때 그 스레드를 만든 스레드의 우선 순위가 상속되며 Thread.Priority 프로퍼티를 통해서 참조하거나 변경 가능함

[Thread.Priority 열거형 원소]

멤버	설 명
Highest	가장 높은 우선순위를 나타낸다.
AboveNormal	높은 우선순위를 나타낸다.
Normal	표준 우선순위를 나타낸다.
BelowNormal	낮은 우선순위를 나타낸다.
Lowest	가장 낮은 우선순위를 나타낸다.

4) 스레드의 스케쥴링

(2) 예제

- 예제 : ThreadPriorityApp.cs

```
using System;
using System.Threading;
class ThreadPriorityApp {
  static void ThreadBody() {
    Thread.Sleep(1000);
  }
  public static void Main() {
    Thread t = new Thread(new ThreadStart(ThreadBody));
    t.Start();
    Console.WriteLine("Current Priority : " + t.Priority);
    ++t.Priority;
    Console.WriteLine("Higher Priority : " + t.Priority);
  }
}
```

실행 결과

```
Current Priority : Normal Higher Priority : AboveNormal
```

Summary

1. 제네릭

- 변수의 형을 매개변수로 하여 클래스나 메소드의 알고리즘을 자료형과 무관하게 기술하는 기법

2. 애트리뷰트

- 어셈블리나 클래스, 필드, 메소드, 프로퍼티 등에 다양한 종류의 속성 정보를 추가하기 위해서 사용

3. 예외

- 실행 시간에 발생하는 에러 (run-time error)

4. 스레드

- 프로그램 내부에 있는 제어의 단일 순차 흐름 (single sequential flow of control)