제 6 장 고급 프로그래밍 기법

- 델리게이트
- 이벤트
- 스레드
- 애트리뷰트
- 예외

6장 요약

- C# 언어의 고급 프로그래밍 기법
  - 응용 프로그램을 보다 짜임새 있고 원론적으로 작성 가능
  - 델리게이트
    - 객체지향 특성이 반영된 메소드 포인터
    - 이벤트와 스레드를 처리하는 방법론
  - 애트리뷰트
    - 프로그래밍 단위(어셈블리, 클래스, 메소드 등)에 줄 수 있는 추가적인 정보
    - 언어 시스템에서 실행 시간에 다양하게 활용할 수 있는 방법을 지원
  - 예외 처리
    - 실행 중에 발생하는 에러
    - 언어 시스템에서 에러 처리를 지원

델리게이트

- 델리게이트(delegate)는 메소드 참조 기법
  - 객체지향적 특징이 반영된 메소드 포인터
- 이벤트와 스레드를 처리하기 위한 방법론
- 특징
  - 정적메소드 및 인스턴트 메소드 참조 가능 – 객체지향적
  - 델리게이트의 형태와 참조하고자하는 메소드의 형태는 항상 일치 – 타입안정적
  - 델리게이트 객체를 통하여 메소드를 호출 – 메소드참조
- VS. 함수포인터(C/C++)
  - 메소드 참조 기법면에서 유사
  - 객체지향적이며 타입 안정적

델리게이트의 정의 [1/2]

- 정의 형태

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

- 수정자
  - 접근 수정자
    - public, protected, internal, private
  - new
  - 클래스 밖에서는 public과 internal 만 가능
- 델리게이트 정의 시 주의점
  - 델리게이트 할 메소드의 메소드 반환형 및 매개변수의 개수, 반환형을 일치시켜야 함

델리게이트의 정의 [2/2]

- 델리게이트 정의 예

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

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);
```

델리게이트 객체 호출

- 델리게이트 객체의 호출은 일반 메소드의 호출과 동일
- 델리게이트를 통하여 호출할 메소드가 매개변수를 갖는다면 델리게이트를 호출하면서 ( )안에 매개변수를 기술

[예제 6.1 - **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
```

멀티캐스트

- 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

[예제 6.2 - **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 : 2005-06-11 오후 12:05:30
Date : 2005-06-11 오전 12:00:00
```

이벤트

- **이벤트**(event)
  - 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

이벤트 정의 [1/3]

- 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 [2/3]

- 이벤트 정의 순서
  1. 이벤트 처리기를 작성
  2. 이벤트 처리기의 형태와 일치하는 델리게이트를 정의
     (또는 System.EventHandler 델리게이트를 사용)
  3. 델리게이트를 이용하여 이벤트를 선언
     (미리 정의된 이벤트인 경우에는 생략)
  4. 이벤트에 이벤트 처리기를 등록
  5. 이벤트를 발생
     (미리 정의된 이벤트는 사용자 행동에 의해 이벤트가 발생)

- 이벤트가 발생되면 등록된 메소드가 호출되어 이벤트를 처리
  - 미리 정의된 이벤트 발생은 사용자의 행동에 의해서 발생
  - 사용자 정의 이벤트인 경우에는 명시적으로 델리게이트 객체를 호출함으로써 이벤트 처리기를 작동

[예제 6.4 - **EventHandlingApp.cs**]

```
using System;
public delegate void MyEventHandler();                    // ② 이벤트를 위한 델리게이트 정의
class Button {
   public event MyEventHandler Push;                      // ③ 이벤트 선언
   public void OnPush() {
      if (Push != null)
         Push();                                          // ⑤ 이벤트 발생
   }
}
class EventHandlerClass {
   public void MyMethod() {                               // ① 이벤트 처리기 작성
      Console.WriteLine("In the EventHandlerClass.MyMethod ...");
   }
}
class EventHandlingApp {
   public static void Main() {
      Button button = new Button();
      EventHandlerClass obj = new EventHandlerClass();
      button.Push += new MyEventHandler(obj.MyMethod);    // ④ 등록
      button.OnPush();
   }
}
```

[실행 결과]

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

이벤트 정의 [3/3]

- 이벤트 처리기 등록
  - 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
```

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

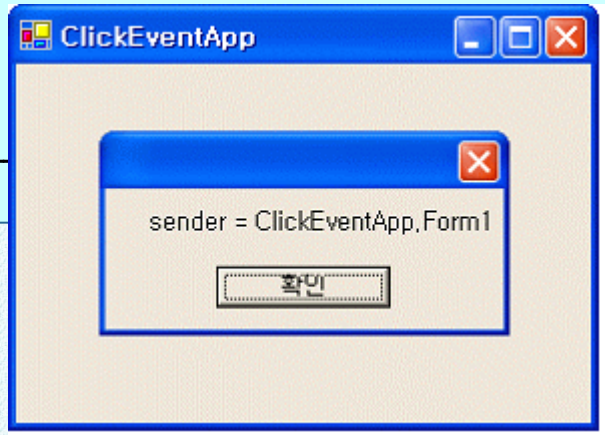
```
delegate void EventHandler(object sender, EventArgs e);
```

- 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 제공한

[예제 6.5 - **ClickEventApp.cs**]

```
using System;
using System.Windows.Forms;
class ClickEventApp : Form {
  public ClickEventApp() {                    // 생성자
    this.Text = "ClickEventApp";
    this.Click += new EventHandler(ClickEvent); // 등록
  }
  // ... 이벤트 처리기 ...
  private void ClickEvent(object sender, EventArgs e) {
    MessageBox.Show(" sender = " + sender.GetType());
  }
  public static void Main() {
    Application.Run(new ClickEventApp());
  }
}
```

[실행결과]



스레드 [1/3]

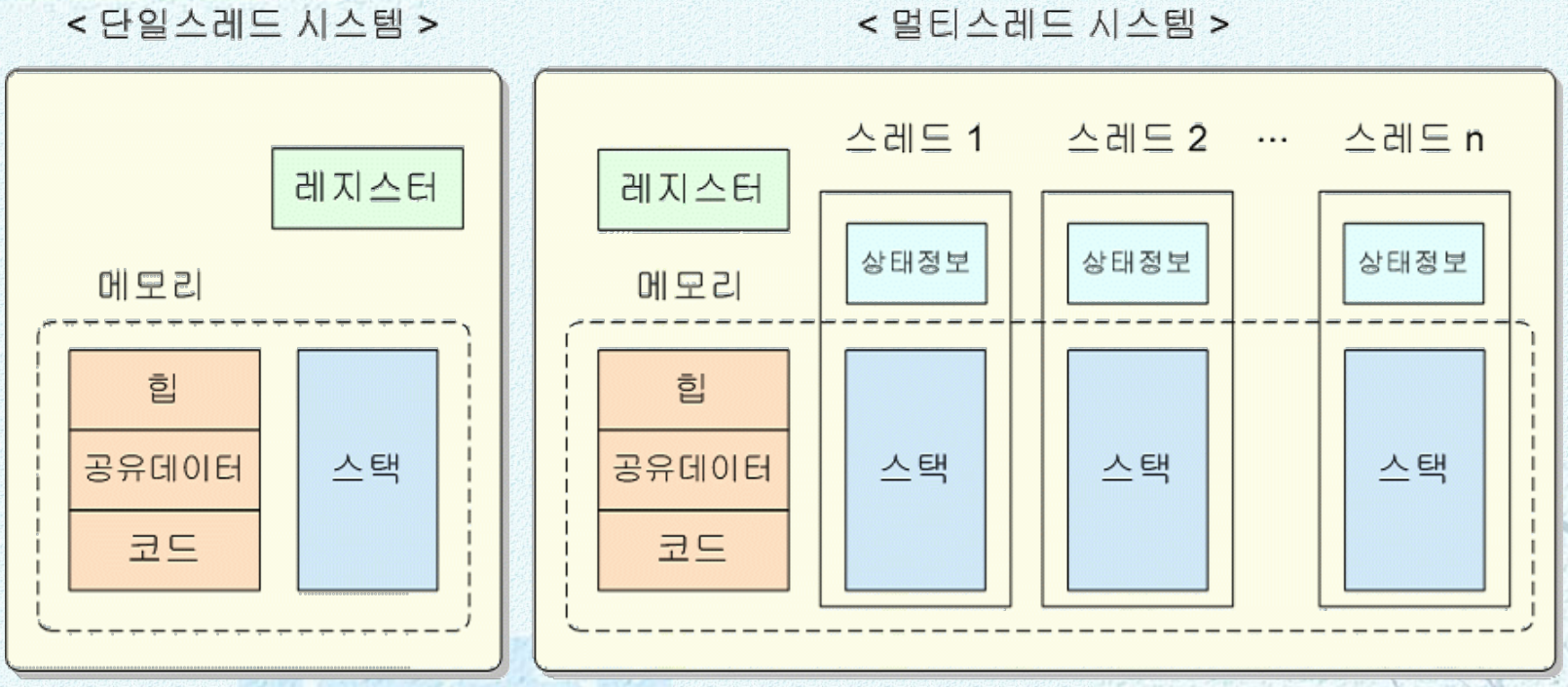
- 순차 프로그램(sequential program)
  - 각 프로그램별로 시작, 순차적 실행 그리고 종료를 가짐
  - 프로그램의 실행 과정 중 오직 하나의 실행 점(execution point)을 갖고 있음
- 스레드
  - 순차 프로그램과 유사하게 시작, 실행, 종료의 순서를 가짐
  - 실행되는 동안에 한 시점에서 단일 실행 점을 가짐
  - 프로그램 내에서만 실행 가능
    - 스레드는 프로그램 내부에 있는 제어의 단일 순차 흐름(single sequential flow of control)
    - 단일 스레드 개념은 순차 프로그램과 유사

스레드 [2/3]

- 멀티스레드(multithread) 시스템
  - 스레드가 하나의 프로그램 내에 여러 개 존재
  - 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

스레드 [3/3]

- 단일스테드 시스템과 멀티스레드 시스템의 구조 비교



스레드 프로그래밍 [1/2]

- C#에서의 스레드 개념
  - 객체이며 스레드가 실행하는 단위는 메소드
  - 스레드 객체를 위해 Thread 클래스를 제공
  - 메소드 연결을 위해 ThreadStart 델리게이트를 제공 (System.Threading)
- 스레드 프로그래밍의 순서
  1. 스레드 몸체에 해당하는 메소드를 작성
  2. 작성된 메소드를 ThreadStart 델리게이트에 연결
  3. 생성된 델리게이트를 이용하여 스레드 객체를 생성
  4. 스레드의 실행을 시작(Start() 메소드를 호출)

[예제 6.6 – **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
```

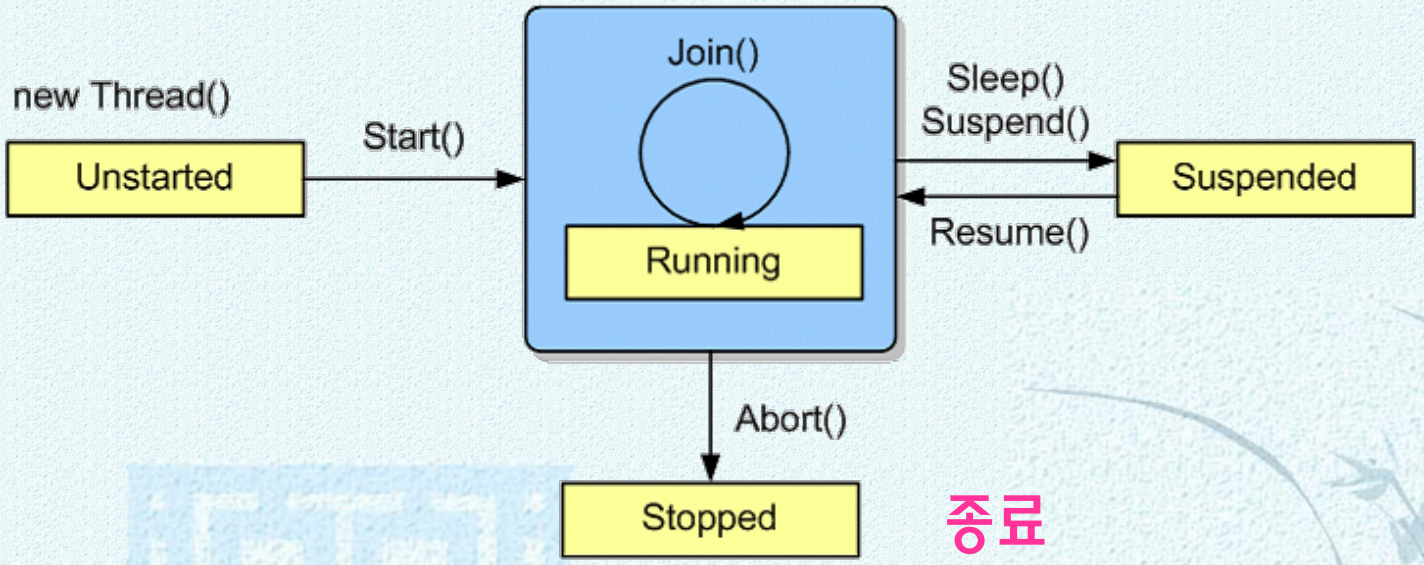
스레드 프로그래밍 [2/2]

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.

스레드의 상태 [1/2]

- 스레드의 상태도
  - **상태** : Unstarted, Running, Suspended, Stopped
  - **시작메소드** : Start()
  - **종료조건** : **메소드의 종료**, Abort() **실행**



스레드의 상태 [2/2]

- 스레드의 상태 변경 메소드

메소드	기 능
Thread.Start()	해당 스레드를 실행한다.
Thread.Abort()	해당 스레드를 종료시킨다.
Thread.Join()	해당 스레드의 실행이 종료될 때까지 기다린다.
Thread.Suspend()	해당 스레드를 대기 상태로 만든다.
Thread.Resume()	대기 상태 스레드를 실행 상태로 만든다.
Thread.Sleep()	지정한 시간동안 실행을 멈추고 대기상태로 간다.

[예제 6.8 – **ThreadStateApp.cs**]

```
using System;
using System.Threading;
class ThreadState {   public void ThreadBody() {   while (true) {   // ... infinite loop ...   }   }   }
class ThreadStateApp {
   public static void Main() {
      ThreadState obj = new ThreadState();
      ThreadStart ts = new ThreadStart(obj.ThreadBody);
      Thread t = new Thread(ts);
      Console.WriteLine("Step 1: " + t.ThreadState);
      t.Start();              Thread.Sleep(100);
      Console.WriteLine("Step 2: " + t.ThreadState);
      t.Suspend();         Thread.Sleep(100);
      Console.WriteLine("Step 3: " + t.ThreadState);
      t.Resume();          Thread.Sleep(100);
      Console.WriteLine("Step 4: ” + t.ThreadState);
      t.Abort();             Thread.Sleep(100);
      Console.WriteLine("Step 5: " + t.ThreadState);
   }
}
```

[**실행 결과**]

```
Step 1: Unstarted
Step 2: Running
Step 3: Suspended
Step 4: Runnable
Step 5: Stopped
```

스레드의 스케줄링

- 정의
  - 실행 가능한 상태에 있는 여러 스레드의 실행 순서를 제어하는 것
- 특징
  - 스레드의 스케줄링에 따라 스레드의 작업 순서가 달라지며, 스레드의 우선순위(priority)에 따라 실행 순서가 결정
  - 스레드가 생성될 때 그 스레드를 만든 스레드의 우선순위가 상속되며 Thread.Priority 프로퍼티를 통해서 참조하거나 변경 가능
    - ThreadPriority 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

[예제 6.10 – **ThreadPriorityApp.cs**]

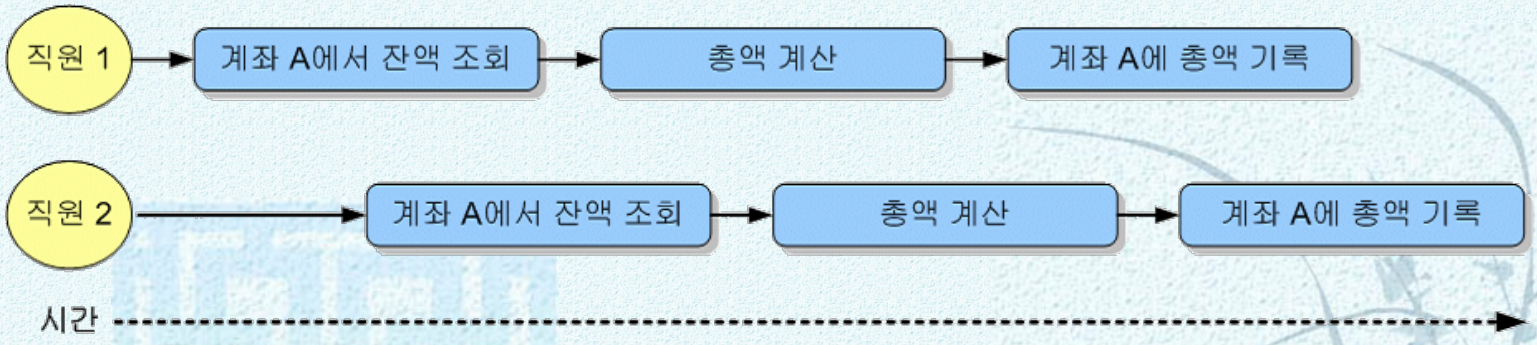
```
using System;
using System.Threading;
class ThreadPriorityApp {
   static void ThreadBody() {
      Thread.Sleep(1000);
   }
   public static void Main() {
      Thread t = new Thread(new ThreadStart(ThreadBody));
      t.Start();
      Console.WriteLine("Current Priority : " + t.Priority);
    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

}
```

[**실행 결과**]

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

동기화 [1/3]

- 비동기 스레드
  - 각각의 스레드는 그의 실행에 필요한 모든 자료와 메소드를 포함
  - 병행으로 실행 중인 다른 스레드의 상태 또는 행위에 관계되지 않는 자신만의 공간에서 실행
- 동기 스레드
  - 동시에 실행되는 스레드들이 자료를 공유
  - 다른 스레드의 상태와 행위를 고려
  - 동기화 문제



동기화 [2/3]

- lock 문
  - 동일한 객체에 대하여 여러 스레드의 중첩 실행을 방지
  - lock 문은 아토믹 루틴(atomic routine)이 되어 실행 순서를 제어
  - lock 문의 형태

```
lock (object obj)
   <문장>
```

[예제 6.12 – **LockStApp.cs**]

```
public void ThreadBody() {
    Thread myself = Thread.CurrentThread;
    lock (this) {
        for (int i = 1; i <= 3; i++) {
            Console.WriteLine("{0} is activated => {1}",
                        myself.Name, i);
            Thread.Sleep(1000);
        }
    }
}
```

[**실행 결과**]

```
Apple is activated => 1
Apple is activated => 2
Apple is activated => 3
Orange is activated => 1
Orange is activated => 2
Orange is activated => 3
```

동기화 [3/3]

- Monitor 클래스
  - 임의의 객체에 대한 특정 작업을 동기화를 하기 위해 사용
  - Monitor.Enter()와 Monitor.Exit() 사이에 실행되는 작업을 동기적으로 실행
  - Monitor 클래스의 메소드

메소드	설 명
Enter(Object obj)	매개변수로 전달된 스레드 객체가 특정 문장을 선점할 수 있도록 지정한다.
Exit(Object obj)	Enter() 메소드를 통하여 특정 문장의 실행을 선점한 스레드를 다른 스레드가 접근할 수 있도록 해지한다.

[예제 6.13 – **MonitorApp.cs**]

```
public void ThreadBody() {
    Thread myself = Thread.CurrentThread;
    Monitor.Enter(this);
    for (int i = 1; i <= 3; i++) {
        Console.WriteLine("{0} is activated => {1}",
                          myself.Name, i);
        Thread.Sleep(1000);
    }
    Monitor.Exit(this);
}
```

[**실행 결과**]

```
Apple is activated => 1
Apple is activated => 2
Apple is activated => 3
Orange is activated => 1
Orange is activated => 2
Orange is activated => 3
```

애트리뷰트

- 애트리뷰트의 특징
  - 어셈블리나 클래스, 필드, 메소드, 프로퍼티 등에 다양한 종류의 속성 정보를 추가하기 위해서 사용
  - 어셈블리에 메타데이터(metadata) 형식으로 저장되며, 이를 참조하는 .NET 프레임워크나 C# 또는 다른 언어의 컴파일러에 의해 다양한 용도로 사용
- 애트리뷰트의 정의 형태

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

- 애트리뷰트의 종류
  - 표준 애트리뷰트(.NET 프레임워크에서 제공)
  - 사용자 정의 애트리뷰트

표준 애트리뷰트

- Conditional 애트리뷰트
  - 조건부 메소드를 작성할 때 사용
  - C/C++ 언어에서 사용했던 전처리기 지시어를 이용하여 명칭을 정의 (#define)
  - System.Diagnostics를 사용해야 함
- Obsolete 애트리뷰트
  - 앞으로 사용되지 않을 메소드를 표시하기 위해서 사용
  - 해당 애트리뷰트를 가진 메소드를 호출할 경우 컴파일러는 컴파일 과정에서 애트리뷰트에 설정한 내용이 출력하는 경고를 발생

[예제 6.15 – **ObsoleteApp.cs**]

```
using System;
class ObsoleteAttrApp {
   [Obsolete("경고, Obsolete Method입니다.")]
   public static void OldMethod() {
      Console.WriteLine("In the Old Method ...");
   }
   public static void NormalMethod() {
      Console.WriteLine("In the Normal Method ...");
   }
   public static void Main() {
      ObsoleteAttrApp.OldMethod();
      ObsoleteAttrApp.NormalMethod();
   }
}
```

[**실행 방법**]

```
C:\> csc ObsoleteAttrApp.cs
ObsoleteAttrApp.cs(11,13): warning CS0618: 'ObsoleteAttrApp.OldMethod()'은(는) 사용되지 않습니다.
 '경고, Obsolete Method입니다.'
```

[**실행 결과**]

```
In the Old Method ...
In the Normal Method ...
```

사용자 정의 애트리뷰트

- 특징
  - System.Attribute 클래스에서 파생
    이름의 형태 : XxxxAttribute
  - 정의한 애트리뷰트를 사용할 때는 이름에서 Attribute가 제외된 부분만을 사용
  - 사용자 정의 애트리뷰트나 표준 애트리뷰트를 사용하기 위해서는 .NET 프레임워크가 제공하는 리플렉션 기능을 사용

```
// 사용자 정의 애트리뷰트를 정의한 예
public class AttributeNameAttribute: Attribute {
    // 생성자 정의
}
// ...
// 사용자 정의 애트리뷰트를 사용한 예
[AttributeName()]
```

[예제 6.16 – **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 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);
    }
  }
}
```

[**실행 결과**]

```
This is Attribute test.
```

예외

- 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 발생
  - 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

예외 정의 [1/4]

- 예외도 하나의 객체로 취급
  - 따라서, 먼저 예외를 위한 클래스를 정의하여야 함
- 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
- 예외를 명시적으로 발생시키면 예외를 처리하는 예외 처리기가 반드시 필요함
- 예외에 관련된 메시지를 스트링 형태로 예외 객체에 담아 전달 가능

[예제 6.17 – **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**]

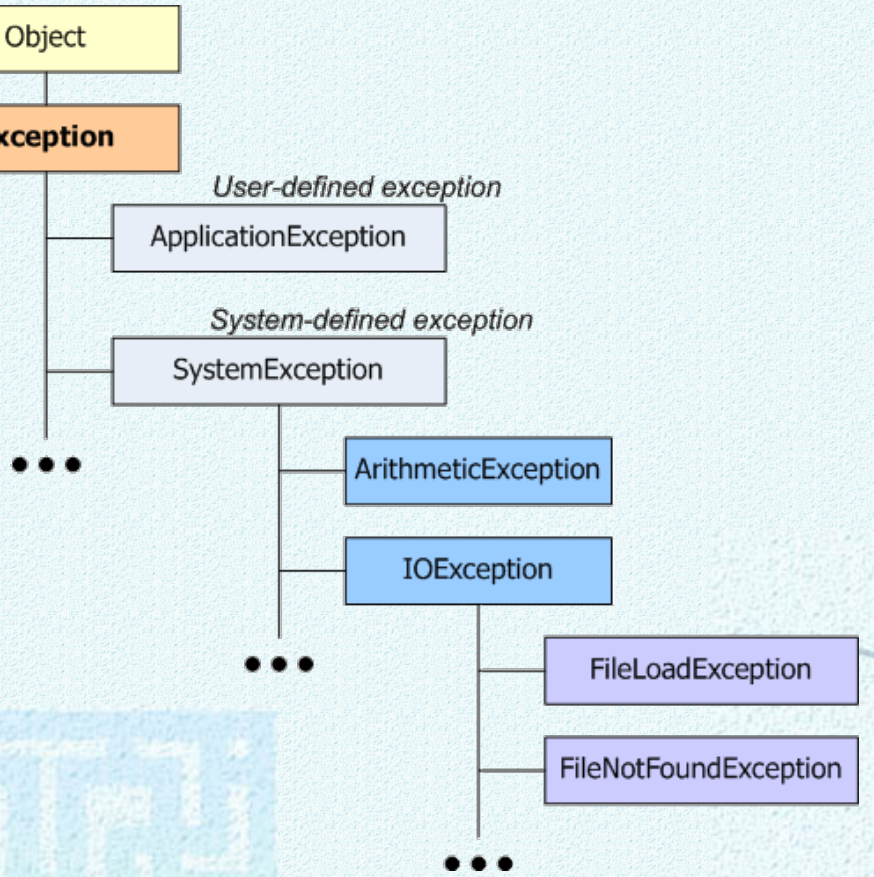
```
throw a exception with a message
```

예외 정의 [2/4]

- **시스템 정의 예외** (system-defined exception)
  - **프로그램의 부당한 실행으로 인하여 시스템에 의해 묵시적으로 일어나는 예외**
  - SystemException**클래스나** IOException **클래스로부터 확장된 예외**
  - CLR**에 의해 자동적으로 생성**
  - **야기된 예외에 대한 예외 처리기의 유무를 컴파일러가 검사하지 않음** (unchecked exception)
- **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)

예외 정의 [3/4]

- 예외 클래스의 계층도



예외 정의 [4/4]

- 시스템 정의 예외의 종류
  - ArithmeticException
    - 산술 연산시 발생하는 예외
  - IndexOutOfRangeException
    - 배열, 스트링, 벡터 등과 같이 인덱스를 사용하는 객체에서 인덱스의 범위가 벗어날 때 발생하는 예외
  - ArrayTypeMismatchException
    - 배열의 원소에 잘못된 형의 객체를 배정하였을 때 발생하는 예외
  - InvalidCastException
    - 명시적 형 변환이 실패할 때 발생하는 예외
  - NullReferenceException
    - null을 사용하여 객체를 참조할 때 발생하는 예외
  - OutOfMemoryException
    - 메모리 할당(new)이 실패하였을 때 발생하는 예외

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
- 명시적 예외 발생
  - throw 문을 이용하여 프로그래머가 의도적으로 발생

```
throw ApplicationExceptionObject;
```

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

[예제 6.19 – **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
```

예외 처리 [1/2]

- 에러 처리 구문(try-catch-finally 구문)
  - 예외를 검사하고 처리해주는 문장
  - 구문 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

예외 처리 [2/2]

- 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.
- Default 예외 처리기
  - 시스템 정의 예외가 발생됐는데도 불구하고 프로그래머가 처리하지 않을 때 작동
  - 단순히 에러에 대한 메시지를 출력하고 프로그램을 종료하는 기능

[예제 6.19 – **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
```

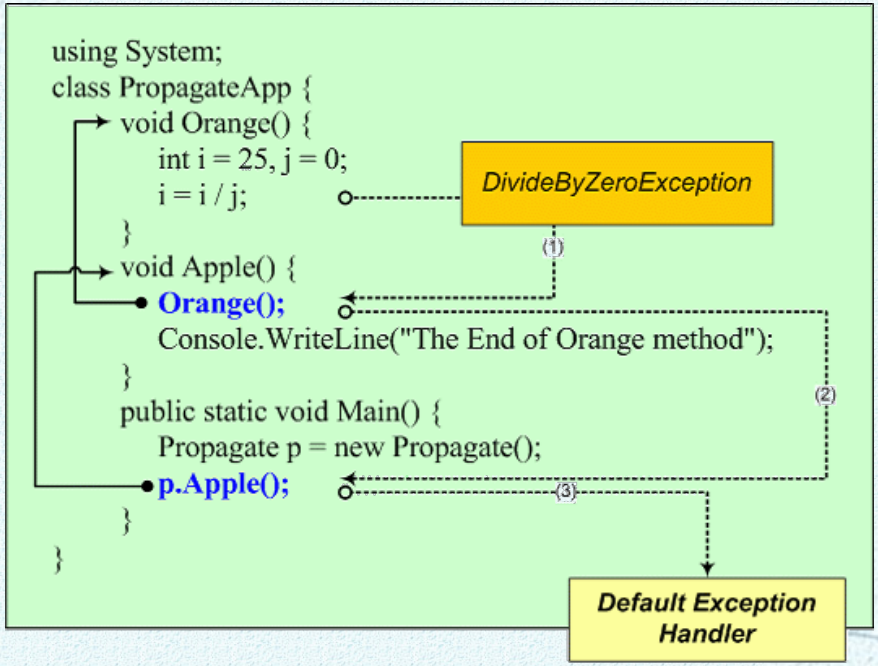
예외 전파 [1/2]

- 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

예외 전파 [2/2]

- 예외 전파 과정




```
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
```

[예제 6.25 – **PropagateApp.cs**]

```
using System;
class PropagateApp {
  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() {
    PropagateApp p = new PropagateApp();
    try {
      p.Apple();
    } catch (ArithmeticException) {
      Console.WriteLine("ArithmeticException is processed");
    }
    Console.WriteLine("End of Main");
  }
}
```

[**실행 결과**]

```
ArithmeticException is processed
End of Main
```