

01. 일반화 프로그래밍이란?

- ▶ 일반화 프로그래밍(Generic Programming)
데이터 형식(Data Type) 일반화를 이용하는 프로그래밍 패러다임
- ▶ 일반화(Generalization) : 특수한 개념으로부터 공통된 개념을 찾아 묶는 것
 - 타입 인수를 사용하여 일반화된 클래스나 메서드를 정의하는 기법이다.
 - 내부 구조나 알고리즘은 동일하게 하되, 취급하는 자료형이 다른 클래스나 메서드가 필요할 때 사용한다.
 - 일반화된 클래스나 메서드를 정의하는 기법이다.
 - 코드량 감소, 캐스팅 하지 않음으로 인한 속도 증가 효과

01. 일반화 프로그래밍이란?

.Net Framework에서 제공하는 Generic

- Dictionary<TKey, TValue> : Hashtable의 제네릭 버전. Key & Value가 한쌍. Key는 유일
- List<T> : ArrayList의 제네릭 버전. 배열의 크기를 동적으로 구성
- SortedList<TKey, TValue> : Dictionary + List. Key & Value가 한쌍. 동적 배열. Key값으로 정렬됨.
- LinkedList<T> : 새로생김.
- Queue<T> : Queue의 제네릭 버전
- Stack<T> : Stack의 제네릭 버전

타입	제네릭	비제네릭	제네릭 네임스페이스
클래스	List<T>	ArrayList	System.Collections.Generic
	Dictionary<TKey, TValue>	HashTable	
	SortedList<TKey, TValue>	SortedList	
	Stack<T>	Stack	
	Queue<T>	Queue	
	LinkedList<T>	-	
	ReadOnlyCollection<T>	-	System.Collections.ObjectModel
	KeyedCollection<TKey, TValue>	-	

02. 제너릭(일반화) 클래스(1/2)

- ▶ 일반화 클래스
 - ▶ (데이터 형식을) 일반화한 클래스
- ▶ 멤버변수 타입을 미리 결정하지 않고 클래스를 생성하며, 이는 데이터형식을 일반화 하는 클래스가 된다. 즉 여러 자료형을 동일 클래스 구조로 개체를 생성한다.

```
class Array_Int
{
    private int[] array;
    public int GetElement( int index )
    { return array[index]; }
}
```

```
class Array_Double
{
    private double[] array;
    public double GetElement( int index )
    { return array[index]; }
}
```

```
class Array_Generic<T>
{
    private T[] array;
    // ...
    public T GetElement( int index ) { return array[index]; }
}
```

02. 일반화 클래스 (2/2)

```
class GenericClass<T> //클래스 제너릭(일반화)
```

```
{
```

```
    private T value;
```

```
    public GenericClass()
```

```
{
```

```
}
```

```
    public GenericClass(T _value)
```

```
{
```

```
        this.value = _value;
```

```
}
```

```
    public T Value
```

```
{
```

```
        get { return this.value; }
```

```
        set { this.value = value; }
```

```
}
```

```
}
```

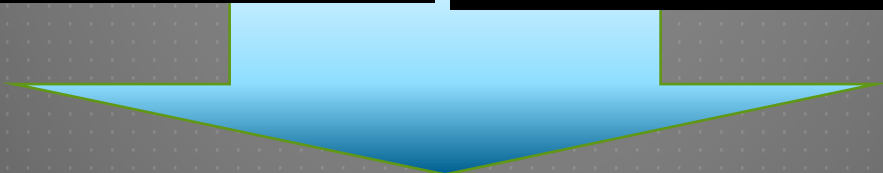
```
GenericClass<int> glnt = new GenericClass<int>(200);  
Console.Write("int : ");  
Console.WriteLine(glnt.Value + 250);
```

03. 일반화 메소드 (1/2)

- ▶ 일반화 메소드(Generic Method)
 - ▶ (데이터 형식)을 일반화한 메소드
- ▶ 일반화 메소드 선언 예

```
void CopyArray( int[ ] source, int [] target )  
{  
    for( int i = 0; i < source.Length; i++ )  
        target[i] = source[i];  
}
```

```
void CopyArray( string[ ] source, string [] target )  
{  
    for( int i = 0; i < source.Length; i++ )  
        target[i] = source[i];  
}
```



```
void CopyArray<T> ( T[ ] source, T[] target )  
{  
    for( int i = 0; i < source.Length; i++ )  
        target[i] = source[i];  
}
```

03. 일반화 메소드 (2/2)

▶ 일반화 메소드 사용 예

```
int[] source = { 1, 2, 3, 4, 5 };  
int[] target = new int[source.Length];
```

형식 매개 변수 T에 int를 대입

```
CopyArray<int>(source, target);
```

```
foreach (int element in target)  
    Console.WriteLine(element);
```

04. 형식 매개 변수 제약시키기 (1/2)

- ▶ 모든 형식에 대응할 수 있는 형식 매개 변수가 필요한 때 도 있지만, 종종 특정 조건을 갖춘 형식에만 대응하는 형식 매개 변수가 필요 할 때 도 있음.
- ▶ 이런 경우, 우리는 형식 매개 변수의 조건에 제약을 줄 수 있음.
- ▶ 제약은 일반화 클래스와 일반화 메소드 모두에 적용 가능
- ▶ 제약을 거는 구문은 다음과 같음

where 형식매개변수 : 제약조건

04. 형식 매개 변수 제약시키기 (2/2)

- ▶ 형식매개 변수 T를 Class의 참조형식으로 한정시키는 예

```
class MyList<T> where T : class
{
    // ...
}
```

- ▶ 형식 매개 변수 T를 값 형식으로 한정시키는 예

```
void CopyArray<T>( T[ ] source, T[] target ) where T : struct
{
    for( int i = 0; i < source.Length; i++ )
        target[i] = source[i];
}
```


05. 일반화 컬렉션 (1/6)

- ▶ **Generic**(범용) : 한가지 type으로 고정하여 속도를 빠르게한다. 컴파일때 Type을 고정한다.
- ▶ `Systems.Collections.Generic` 네임스페이스는 다양한 일반화 컬렉션을 제공함. 대표적인 컬렉션 클래스

```
List<string> : AddList() 를 하나의 자료형으로 처리  
Stack<int>  : Stack()을 하나의 자료형으로 처리  
Queue<int>  : Queue를 하나의 자료형으로 처리  
Dictionary<char, string> : HashTable을 하나의 자료형으로 처리
```

07. 컬렉션(COLLECTION)

- ▶ 컬렉션이란? (**Collection(Data)**를 모아서 처리하는 것)

- ▶ 같은 성격을 띄는 데이터의 모음을 담는 자료 구조
- ▶ 배열도 .NET 프레임워크가 제공하는 컬렉션 자료구조 중 하나

- ▶ ArrayList

추가,제거가 자유롭고 자료형이 달라도 된다. 갯수의 제한이없다.
ObjectType으로 변환하여 사용하므로 속도 느리다.

Array 은 고정된 크기를 갖는다.

```
ArrayList list = new ArrayList();  
list.Add( 10 );  
list.Add( 20 );  
list.Add( 30 );
```

```
list.RemoveAt( 1 ); // 20을 삭제
```

```
list.Insert( 25, 1 ); // 25를 1번 인덱스에 삽입. 즉, 10 과 30 사이에 25를 삽입
```

05. 일반화 컬렉션 (2/6)

▶ List<T>

ArrayList의 구조를 한가지 자료형으로 고정하여 속도를 빠르게 사용한다.

```
List<int> list = new List<int>();  
for (int i = 0; i < 5; i++)  
    list.Add(i);  
  
foreach (int element in list)  
    Console.Write("{0} ", element);  
Console.WriteLine();  
  
list.RemoveAt(2);
```

05. 일반화 컬렉션 (3/6)

▶ Stack<T> 사용 예

```
Stack<int> stack = new Stack<int>();
```

```
stack.Push(1);
```

```
stack.Push(2);
```

```
stack.Push(3);
```

```
stack.Push(4);
```

```
stack.Push(5);
```

```
while (stack.Count > 0)
```

```
    Console.WriteLine(stack.Pop());
```

05. 일반화 컬렉션 (4/6)

▶ Queue<T> 사용 예

```
Queue<int> queue = new Queue<int>();
```

```
queue.Enqueue(1);
```

```
queue.Enqueue(2);
```

```
queue.Enqueue(3);
```

```
queue.Enqueue(4);
```

```
queue.Enqueue(5);
```

```
while (queue.Count > 0)
```

```
    Console.WriteLine(queue.Dequeue());
```

07. 컬렉션 맛보기 (5/5), 10장에서

▶ Hashtable

- ▶ 키(Key)와 값(Value)으로 이루어진 데이터를 다룰 때 사용.
- ▶ 키를 해싱(Hashing)을 통해 테이블 내의 주소를 계산.
- ▶ 다루기 간편하고 탐색속도도 빠름

```
Hashtable nameHT = new Hashtable();
```

```
//해쉬테이블에 Key, Value 추가  
nameHT.Add("홍길동", 3500000);  
nameHT.Add("김몽룡", 2750000)
```

05. 일반화 컬렉션 (5/6)

▶ Dictionary<TKey,TValue>

- **Dictionary** 제네릭 클래스는 키 집합에서 값 집합으로의 매핑을 제공하고 값과 관련 키는 항상 사전에 함께 추가 된다.
- **Dictionary** 클래스가 해시 테이블로 구현되므로 키를 사용하여 값을 검색하면 매우 빠른 속도로 검색 작업이 수행된다.
- **Dictionary**의 모든 키는 고유해야 한다.(유일성)
- **Dictionary**에 추가될 때 필요하면 내부 배열의 재할당을 통해 용량이 자동으로 증가된다.
- C++ STL의 컨테이너중 Map과 동일하다.
- **Dictionary**는 KeyValuePair로 foreach문을 이용한 순차접근 한다.

05. 일반화 컬렉션 (6/6)

▶ Dictionary<TKey,TValue>

```
Dictionary<string, string> dic = new Dictionary<string, string>();
```

```
dic["하나"] = "one";  
dic["둘"] = "two";  
dic["셋"] = "three";  
dic["넷"] = "four";  
dic["다섯"] = "five";
```

```
Console.WriteLine(dic["하나"]);  
Console.WriteLine(dic["둘"]);  
Console.WriteLine(dic["셋"]);  
Console.WriteLine(dic["넷"]);  
Console.WriteLine(dic["다섯"]);
```