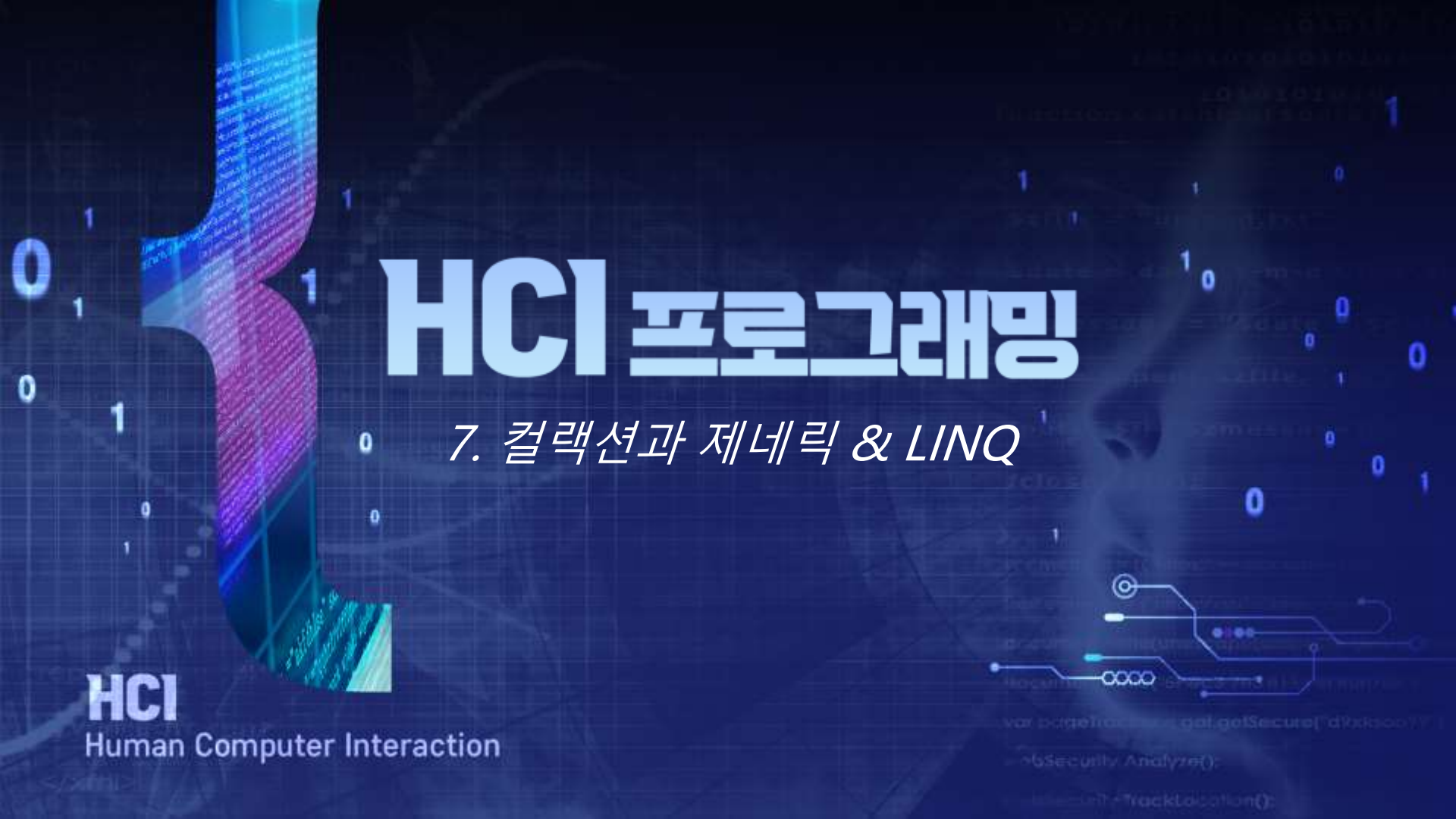

HCI 프로그래밍
7. 컬랙션과 제네릭 & LINQ
HCI
Human Computer Interaction

자료 구조

자료 구조(Data Structure)

- 자료들이 저장되고 사용되는 방식
- 즉, 정해진 방식에 따라 자료를 저장하고, 자료간 관계를 저장하고,
  이를 효과적으로 사용할 수 있는 방법을 정함
- 저장 방식과 사용법이 다양함
  - 자료를 순서대로 저장
  - 키(Key)와 값(Value)으로 저장
  - 계층형으로 저장

- 특히 자료가 많을 때 관리 용이함
- C#에서는 자료 구조를 지원하는 다양한 클래스 제공함
- C#에서 자료 구조 클래스들은 모두 메모리가 허용하는 한 저장할 수 있는 자료의 개수에 제한을 두지 않음

자료 구조

자료구조	설명	명칭	관련 C# 클래스
순서에 따른 구조	• 자료를 순서에 따라 연결하는 구조 • 인덱스(Index)로 자료를 읽을 수 있음	• 배열(Array) • 리스트(List)	배열, ArrayList, Stack, Queue, List<T>, LinkedList<T>, Stack<T>, Queue<T>
키에 따라 읽는 구조	• 고유 Key에 대응하는 자료가 연동되는 구조 • 키가 서로 중복되지 않아야 함 • 키를 인덱스로 삼아 자료 읽기	• 맵(Map) • 딕셔너리(Dictionary)	Hashtable, SortedList, NameValueCollection, Dictionary<T,V>, SortedList<T, V>, KeyValuePair<T, V>
집합구조	• 자료를 순서 없어 저장 • 동일한 자료는 한 개만 존재하고, 중복 허용 않음 • 수학의 집합과 유사함	• 집합(Set)	HashSet<T>, SortedSet<T>
계층에 따라	• 자료가 하위자료를 가지는 계층적 구조	• 트리(Tree)	

자료구조의 필요성

자료 구조(Data Structure)의 필요성

- 자료 구조를 이용하면 관련 있는 자료들을 동일한 변수 이름으로 체계적으로 저장하고 사용할 수 있음
- 배열을 이용해서 같은 변수 이름으로 여러 개의 자료를 저장하고 인덱스를 이용해서 각 자료에 접근 가능
  - 여러 개의 변수 대신 배열과 반복문을 이용하면 코드를 줄일 수 있음

```
int x0 = 1;
int x1 = 2;
int x2 = 3;
int x3 = 4;
int x4 = 5;
```

```
int[] x = new int[5];

for (int i = 0; i < 5; i++) {
   x[i] = i + 1;
}
```

자료구조의 필요성

또 다른 예

- 공중파 TV 시청 프로그램을 작성한다고 가정

방송국	채널 번호
"SBS"	6
"KBS2"	7
"KBS1"	9
"EBS1"	10
"MBC"	11

- 변수로 관리

```
int SBS = 6;
int KBS2 = 7;
int KBS1 = 9;
int EBS1 = 10;
int MBC = 11;
```

자료구조의 필요성

- 방송국 이름과 채널 번호를 나타내는 변수를 프로그래머가 연관시켜야 함
- 방송국 중 한 개를 이름과 채널 번호로 출력

```
Console.WriteLine("television networks: {0}, channel number: {1}", "SBS", SBS);
```

- 전체 출력

```
Console.WriteLine("television networks: {0}, channel number: {1}", "SBS", SBS);
Console.WriteLine("television networks: {0}, channel number: {1}", "KBS2", KBS2);
...
Console.WriteLine("television networks: {0}, channel number: {1}", "MBC", MBC);
```

자료구조의 필요성

- 차라리 표를 프로그램에 저장해 놓고, 방송국의 이름을 이용해서 채널 번호를 찾을 수 있도록 구현
- 방송국 이름을 인덱스처럼 사용할 수 있는 배열

```
// 자료구조에 방속국 이름과 채널 번호를 연관시켜 저장
channels["SBS"] = 6;

// 방송국 이름을 이용해서 채널 번호를 찾아내서 화면에 출력
Console.WriteLine("television networks: {0}, channel number: {1}", "SBS", channels["SBS"]);
```

자료구조의 필요성

자료 구조를 이용할 때의 장점

- 서로 자료가 연관되어 있을 때, 동일한 변수 이름 사용
  - 변수를 여러 개 사용할 때보다 가독성이 높고 코드 이해의 용이성
- 자료 항목 각각에 동일한 연산 또는 비슷한 연산을 처리하는 경우 반복문을 사용해서 코드를 줄이기 쉬움

Collections

- 컬랙션 (Collections)은 객체를 쉽게 다룰 수 있도록 여러가지 클래스와 인터페이스를 미리 정의한 자료 구조
  - 자료 구조란 데이터를 다루는 방식을 정의한 것
  - 프로그래머가 일일이 자료 구조를 만드는 불편함 감소
  - 컬랙션으로 `ArrayList, SortedList, Hashtable, Stack, Queue, NameValueCollection` 등이 있음

Collections

컬랙션 특징

- 데이터를 보관할 수 있으며 수정, 삭제, 삽입, 검색 등의 기능
- 컬랙션은 클래스마다 구현되어지는 알고리즘이 다를 뿐 같은 부류임
  - LinkedList, Hash, Stack, Queue 등
- 동적으로 메모리 확장 가능

C#Collections

- .NET 은 다양한 컬랙션을 제공
  - `System.Collections` 클래스
  - `System.Collections.Generic` 클래스
  - `System.Collections.Concurrent` 클래스

System.Collections

Object 객체 컬렉션 클래스

클래스	설명
ArrayList	필요에 따라 크기가 동적으로 증가하는 개체 배열
Hashtable	키의 해시코드에 따라 구성된 키/값 쌍의 컬렉션
Queue	FIFO 방식의 개체 컬렉션
Stack	LIFO 방식의 개체 컬렉션
SortedList	키에 따라 정렬된 키/값 쌍의 컬렉션

System.Collections.Generic

제네릭 컬랙션 클래스

클래스	설명
Dictionary<TKey, TValue>	키에 따라 구성된 키/값 쌍의 컬랙션
HashSet<T>	값 집합
LinkedList<T>	이중 연결 목록
List<T>	순서가 있어서 인덱스로 액세스할 수 있는 개체 목록
Queue<T>	FIFO 방식의 개체 컬랙션
SortedList<TKey, TValue>	연관된 IComparer<T> 구현을 기반으로 키에 따라 정렬된 키/값 쌍의 컬랙션
Stack<T>	LIFO 방식의 개체 컬랙션

System.Collections.Concurrent

.NET 4.0이후 소개된 스레드에 안전한 컬랙션 클래스

클래스	설명
BlockingCollection<T>	IProducerConsumerCollection<T>를 구현하는 스레드로부터 안전한 컬랙션
ConcurrentBag<T>	스레드로부터 안전한 정렬되지 않은 개체 컬랙션
ConcurrentDictionary<TKey, TValue>	여러 스레드에서 동시에 액세스할 수 있는 키/값 쌍의 컬랙션
ConcurrentQueue<T>	스레드로부터 안전한 FIFO 방식의 개체 컬랙션
ConcurrentStack<T>	스레드로부터 안전한 LIFO 방식의 개체 컬랙션

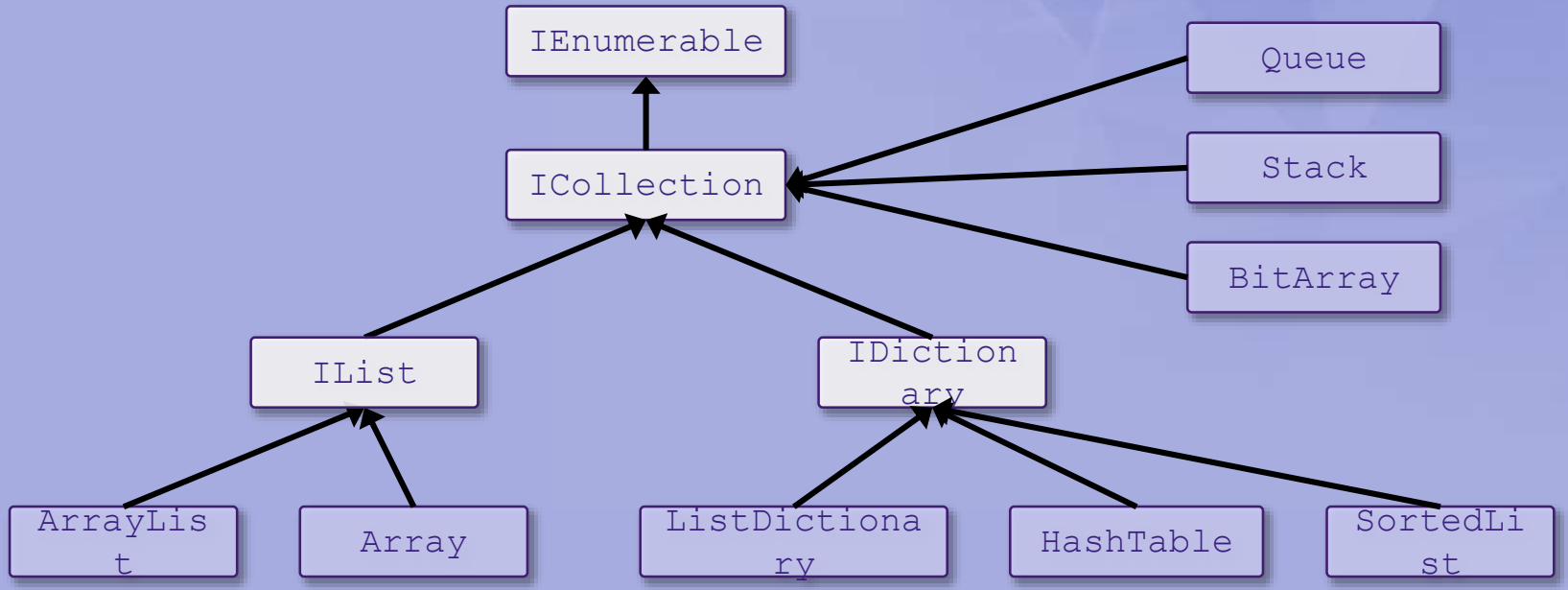
Collections

System.Collections의 클래스와 인터페이스

- ArrayList
- Hashtable
- SortedList
- Stack, Queue
- BitArray
- NameValueCollections
- IEnumerable, IEnumerator
- ICollection
- IList
- IDictionary, IDictionaryEnumerator
- ICloneable
- ISerializable

Collections

- System.Collections의 클래스와 인터페이스



SortedList Class

- SortedList는 Hashtable과 ArrayList의 혼합형
  - 내부의 데이터는 키(Key)와 값(Value)로 이루어져 있으며 키를 기준으로 정렬되고 키와 인덱스로 접근가능
  - 내부적으로 정렬된 컬랙션을 유지하고 있는 특징을 가짐

```
public class SortedList: IDictionary, ICollection, IEnumerable, ICloneable
```

SortedList Class

SortedList 특징

- 키의 목록 또는 값의 목록만 반환하는 메소드 제공
- 내부적으로 두 개의 배열, 즉 키에 대한 배열과 값에 대한 배열을 유지하여 요소를 목록에 저장
- SortedList는 각 요소에 대해 키, 값 또는 인덱스의 세가지 방법으로 접근
- 요소가 삽입되면, 지정된 키가 이미 존재하는 검사 (중복키 허용 안함)

SortedList Class

SortedList 메소드

- `Add()` – 키와 값으로 데이터를 삽입
- `Clear()` – 모든 요소를 제거
- `Contains()` – 특정 키가 들어 있는지 여부를 확인
- `ContainsKey()`, `ContainsValue()` – 특정 키/값이 들어있는지 여부 확인
- `GetByIndex()`, `GetKey()` – 지정한 인덱스에서 값/키를 가져옴
- `GetKeyList()` – 키 리스트를 가져옴
- `Remove()`, `RemoveAt()` – 지정한 키/인덱스로 요소를 제거
- `GetEnumerator()` – `IDictionaryEnumerator`를 반환

Queue Class

- Queue는 **FIFO(First-In, First-Out)** 컬랙션
  - FIFO – 먼저 들어간 데이터가 제일 먼저 나오는 메모리 구조를 클래스화

```
public class Queue: ICollection, IEnumerable, ICloneable
```

- Queue 메소드
  - Enqueue() 메소드는 큐의 첫 위치에 요소를 삽입
  - Dequeue() 메소드는 큐의 마지막 위치의 요소를 반환하고 삭제 (반환되는 데이터형은 object형)
  - Peek() 메소드는 마지막 위치의 요소를 제거하지 않고 반환 (반환되는 데이터형은 object형)

Queue 예시

```
Queue queue = new Queue(new object[] {10, 20, 30});
queue.Enqueue(1); // 가장 마지막에 추가
queue.Enqueue("abc"); // 가장 마지막에 추가
queue.Enqueue(3.4); // 가장 마지막에 추가

foreach(object obj in queue)
    Console.WriteLine(obj);
Console.WriteLine();

while (queue.Count > 0) {
    Console.WriteLine("Dequeue: {0} Count: {1}",
                      queue.Dequeue(), queue.Count);
}
```

```
10
20
30
1
abc
3.4

Dequeue: 10 Count: 5
Dequeue: 20 Count: 4
Dequeue: 30 Count: 3
Dequeue: 1 Count: 2
Dequeue: abc Count:
1
Dequeue: 3.4 Count:
0
```

Stack Class

- Stack는 **LIFO(Last-In, First-Out)** 컬랙션
  - LIFO – 제일 마지막에 들어간 데이터가 제일 먼저 나오는 메모리 구조를 클래스화

```
public class Stack: ICollection, IEnumerable, ICloneable
```

- Stack 메소드
  - Push() 메소드는 스택의 맨 위에 요소를 삽입
  - Pop() 메소드는 스택의 맨 위에 있는 요소를 삭제하고 데이터 반환 (반환되는 데이터형은 object형)
  - Peek() 메소드는 스택의 맨 위에 있는 요소를 제거하지 않고 반환 (반환되는 데이터형은 object형)

Stack 예시

```
Stack stack = new Stack(new object[] {10, 20, 30});
stack.Push(1); // 가장 마지막에 추가
stack.Push("abc"); // 가장 마지막에 추가
stack.Push(3.4); // 가장 마지막에 추가

foreach(object obj in stack)
    Console.WriteLine(obj);
Console.WriteLine();
while (stack.Count > 0) {

    Console.WriteLine("Pop: {0} Count: {1}",
                      stack.Pop(), stack.Count);
}
```

```
3.4
abc
1
30
20
10

Pop: 3.4 Count: 5
Pop: abc Count: 4
Pop: 1 Count: 3
Pop: 30 Count: 2
Pop: 20 Count: 1
Pop: 10 Count: 0
0
```

ArrayList Class

ArrayList는 **ILIST**를 구현한 대표적인 클래스

- ArrayList는 데이터를 삽입했을 때 순서대로 삽입되며 중간삽입이나 제거 또한 가능

```
public class ArrayList: IList, ICollection, IEnumerable, ICloneable
```

ArrayList 메소드

- Add(), AddRange() 메소드는 데이터/데이터 리스트 삽입
- Insert() 메소드는 중간에 데이터 삽입
- Remove(), RemoveAt(), RemoveRange() 메소드는 해당 요소 제거 또는 인덱스로 요소 제거 또는 범위만큼 요소 제거
- Sort() 메소드는 요소 정렬
- GetEnumerator() 메소드는 IEnumerator를 반환

ArrayList 예시

```
ArrayList list = new ArrayList();
list.Add(10); // 가장 마지막에 추가
list.Add(20); // 가장 마지막에 추가
list.Add(30); // 가장 마지막에 추가
foreach(object obj in list)
    Console.WriteLine(obj);
Console.WriteLine();

list.RemoveAt(1); // 1번 인덱스의 요소를 제거
foreach(object obj in list)
    Console.WriteLine(obj);

Console.WriteLine();
list.Insert(1, 25); // 1번 인덱스에 새 요소를 추가
foreach(object obj in list)
    Console.WriteLine(obj);
Console.WriteLine();
```

```
10
20
30

10
30

10
25
30
```

Hashtable Class

- Hashtable은 **IDictionary**를 구현한 대표적인 클래스
  - 내부의 데이터는 키(Key)와 값(Value)을 이용

```
public class Hashtable: IDictionary, ICollection, IEnumerable,
                        ISerializable, IDeserializationCallback, ICloneable
```

Hashtable Class

Hashtable 메소드

- Add() 메소드는 (키, 변수)로 된 데이터 삽입
- Clear() 메소드는 모든 요소 제거
- Remove() 메소드는 키를 확인하여 요소 삭제
- ContainsKey(), ContainsValue() 메소드는 특정 키/값을 포함하는지 확인
- CopyTo() 메소드는 해쉬테이블에 있는 원소를 1차원 배열로 복사
- Keys, Values 속성은 ICollection으로 반환
- GetEnumerator() 메소드는 IDictionaryEnumerator 를 반환

Hashtable 예시

```
Hashtable ht = new Hashtable();
ht.Add(1, “Dooly”);
ht.Add(3, “Heedong”);
ht.Add(2, “Gildong”);
ht[4] = “Tochi”;
if (!ht.ContainsKey(5))
    ht.Add(5, “Douner”);
foreach(DictionaryEntry de in ht)
    Console.WriteLine(“Key={0} Value={1}”, de.Key, de.Value.ToString());

Console.WriteLine(“After remove Dooly”);
ht.Remove(1); // Key=1인 둘리 삭제
foreach(DictionaryEntry de in ht)
    Console.WriteLine(“Key={0} Value={1}”, de.Key, de.Value.ToString());
Console.WriteLine();
```

```
Key=5 Value=Douner
Key=4 Value=Tochi
Key=3 Value=Heedong
Key=2 Value=Gildong
Key=1 Value=Dooly
After remove Dooly
Key=5 Value=Douner
Key=4 Value=Tochi
Key=3 Value=Heedong
Key=2 Value=Gildong
```

Collections Interface

IEnumerable 인터페이스

- `GetEnumerator()` – `IEnumerator` 개체를 반환

IEnumerator 인터페이스

- 내부에서 `IEnumerable`을 사용하여 데이터 검색 기능 제공
- `Current` 속성 – 컬렉션에서 현재 객체에 대한 참조를 반환
- `MoveNext()` – 다음 요소로 이동
- `Reset()` – `Current` 포인터를 컬렉션의 처음 앞으로 설정

Collections Interface

ICollection 인터페이스

- `Count` 속성 – 컬랙션의 객체 수를 반환
- `IsSynchronized` 속성 – 다중 스레드된 액세스를 위해 컬랙션에 대한 액세스를 동기화한 경우 `true` 반환
- `SyncRoot` 속성 – 하나 이상의 코드 문장이 동시에 한 스레드에만 실행되는 것을 확실하게 하기 위해 잠그거나 해제
- `CopyTo()` – 지정한 배열 위치부터 컬랙션 요소를 배열로 복사

Collections Interface

IList 인터페이스

- ICollection 인터페이스에서 파생된 것으로 IEnumerable과 ICollection 기능을 모두 포함
- IsFixedSize 속성 – 리스트가 고정 길이 리스트인지 확인
- IsReadOnly 속성 – 리스트가 읽기전용인지 확인
- 인덱서 속성 – 인덱스 값으로 데이터를 얻거나 추가
- Add() – 리스트 끝에 데이터를 추가
- Clear() – 리스트 내의 모든 데이터를 제거
- Contains() – 어떤 데이터가 리스트 내에 존재하는지 여부 확인
- IndexOf() – 리스트 내의 특정 데이터의 위치를 반환
- Insert() – 리스트 내의 특정 위치에 데이터를 삽입
- Remove() – 매개변수로 입력된 객체를 리스트 내에서 제거
- RemoveAt() – 지정한 인덱스의 데이터를 제거

Collections Interface

IDictionary 인터페이스

- 순서에 의존하는 IList와 달리 키와 값으로 대응시켜 데이터를 추출
- IsFixedSize 속성 – 컬랙션의 크기가 정해져 있는지 검사
- IsReadOnly 속성 – 컬랙션이 읽기전용인지 확인
- Keys 속성 – 컬랙션 내의 모든 키를 나열
- Values 속성 – 컬랙션 내의 모든 값을 나열
- Add() – 키와 값을 전달하여 데이터를 컬랙션에 추가
- Clear() – 컬랙션의 모든 데이터를 제거
- Contains() – 특정 키가 데이터와 연관되어 있는지 검사
- GetEnumerator() – IDictionaryEnumerator 반환
- Remove() – 삭제할 값의 키를 전달하여 데이터를 컬랙션에서 제거

Collections Interface

IDictionaryEnumerator 인터페이스

- `DictionaryEntry` 속성 – 열거 요소 내의 키와 값을 가져옴
- `Key` 속성 – 열거 요소 내의 키를 가져옴
- `Value` 속성 – 열거 요소 내의 값을 가져옴

Collection과 Autoboxing Autounboxing

- C# Collections 클래스(ArrayList, Hashtable, Queue, Stack)에 데이터 입출력이 있으면 그때마다 자동으로 boxing과 unboxing이 계속해서 발생
  - Collections 클래스(ArrayList, Hashtable, Queue, Stack)는 전부 object 타입으로 저장하기 때문에 데이터에 접근할 때마다 본래 타입으로의 형식변화가 일어나기 때문임
  - 데이터가 많아질 수록 컴퓨터 성능에 상당한 부하가 발생
  - 성능 상의 이슈가 문제가 된다면, Collections 클래스보다 Generic Collections을 사용해야 함

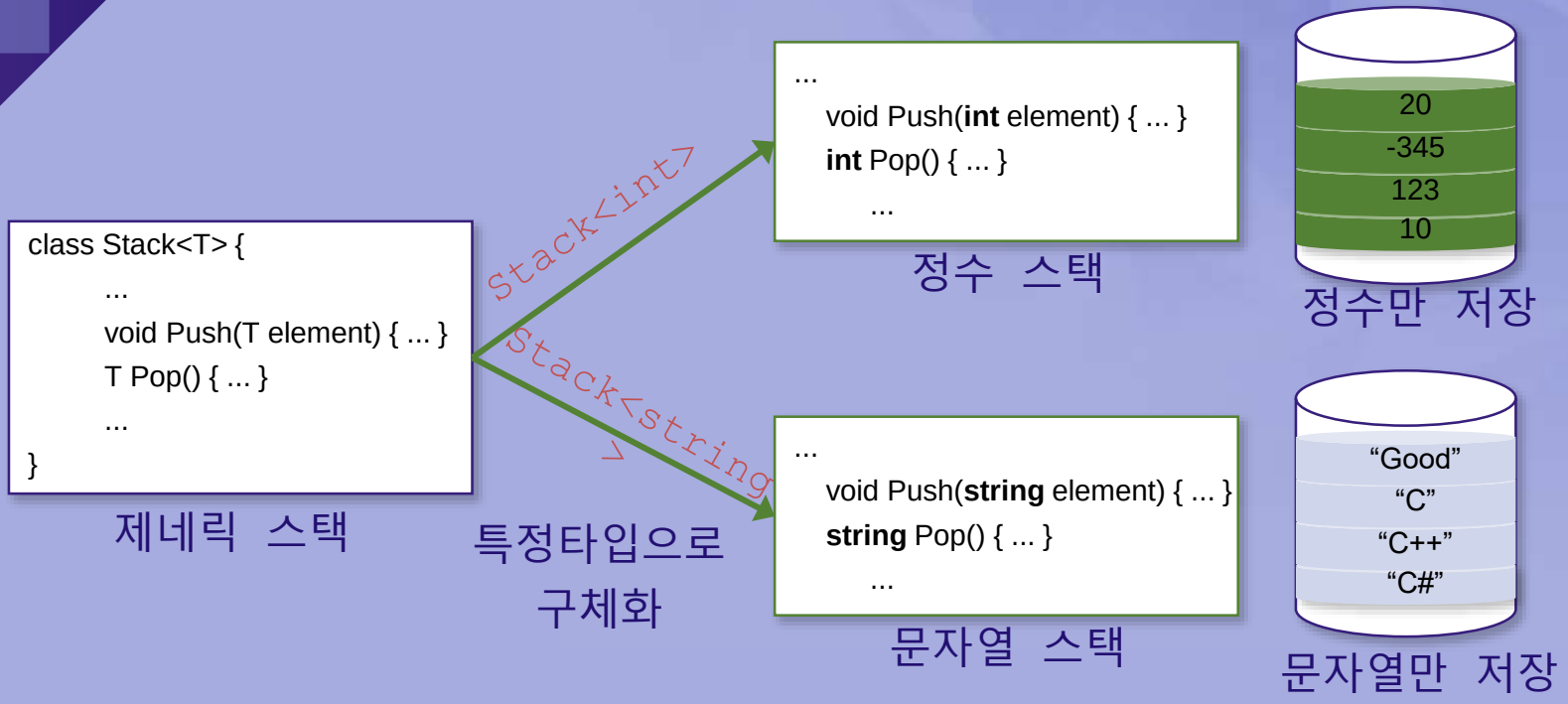
Collection과 Generics

- 제네릭 컬렉션은 제네릭(Generics) 기법으로 구현됨
- 제네릭
  - 특정 타입만 다루지 않고, 여러 종류의 타입으로 변신할 수 있도록 클래스나 메소드를 일반화시키는 기법
    - ✓<T>, <K>, <V> : 타입 매개 변수 요소 타입을 일반화한 타입
  - 제네릭 클래스 사례
    - ✓제네릭 리스트 : List<T>
    - ✓T에 특정 타입으로 구체화
    - ✓정수만 다루는 리스트 : List<int>
    - ✓문자열만 다루는 리스트 : List<string>
    - ✓Person만 다루는 리스트 : List<Person>

Generics

- 제네릭은 다양한 자료형에 적용될 수 있는 일반화된 타입 매개변수로 클래스나 메소드를 작성하는 기법
  - C++의 템플릿(template)과 동일

Generics


class Stack<T> {
...
void Push(T element) { ... }
T Pop() { ... }
...
}
제네릭 스택
Stack<int>
Stack<string>
특정타입으로
구체화
...
void Push(int element) { ... }
int Pop() { ... }
...
정수 스택
20
-345
123
10
정수만 저장
...
void Push(string element) { ... }
string Pop() { ... }
...
문자열 스택
“Good”
“C”
“C++”
“C#”
문자열만 저장

Generics

제네릭은 사용할 자료형을 매개변수로 전달받음

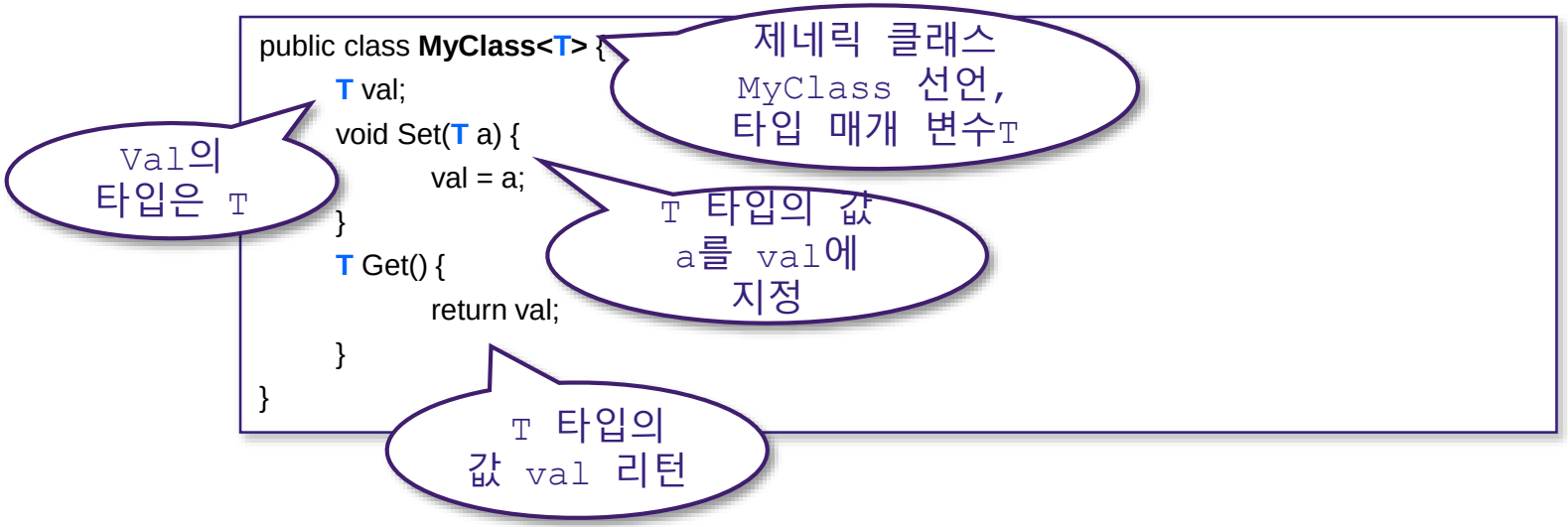
```
public class IntStack {
    int[] items;
    int count;
    public void Push(int item) { … }
    public int Pop() { … }
}

public class StringStack {
    string[] items;
    int count;
    public void Push(string item) { … }
    public string Pop() { … }
}
```

```
public class Stack<T> {
    T[] items;
    int count;
    public void Push(T item) { … }
    public T Pop() { … }
}
….
Stack<int> istack = new Stack<int>();
istack.Push(3);
int x = istack.Pop();
Stack<string> sstack = new Stack<string>();
sstack.Push("C#");
string s = sstack.Pop();
```

제네릭 만들기

제네릭 클래스

- 클래스나 인터페이스 선언부에 일반화된 타입 추가



제네릭 개체 생성
– 구체화(Specialization)

구체화

- 제네릭 타입의 클래스에 구체적인 타입을 대입하여 객체 생성
- 컴파일러에 의해 이루어짐

```
// 제네릭 타입 T에 string 지정
MyClass<string> s = new MyClass<string>();
s.Set("hello");
Console.WriteLine(s.Get()); // "hello" 출력

// 제네릭 타입 T에 int지정
MyClass<int> n = new MyClass<int>();
n.Set(5);
Console.WriteLine(n.Get()); // 숫자 5 출력
```

제네릭 개체 생성
– 구체화(Specialization)

구체화

- 구체화된 `MyClass<string>`의 소스 코드

```
public class MyClass<T> {
        T val;
        void Set(T a) { val = a; }
        T Get() { return val; }
}
```

T가 string으로 구체화

```
public class MyClass<string> {
        string val; // 변수 val의 타입은 string
        void Set(string a) {
                val = a; // string 타입의 값 a를 val에 지정
        }
        string Get() {
                return val; // string 타입의 값 val을 리턴
        }
}
```

제네릭 타입 제약

- 제네릭타입을 선언할 때, 인자 타입을 지정 가능함
  - Value Type인지 Reference Type인지 또는 어떤 특정 Base 클래스로부터 파생된 타입인지 또는 어떤 인터페이스를 구현한 타입인지 등
  - **where T : 제약조건**

제네릭 타입 제약

예시

```
// T는 Value Type
class MyClass<T> where T : struct

// T는 Reference Type
class MyClass<T> where T : class

// T는 default constructor가 있어야 함
class MyClass<T> where T : new()
```

제네릭 타입 제약

예시

```
// T는 MyBaseClass의 파생클래스이어야 함
class MyClass<T> where T : MyBaseClass

// T는 IComparable 인터페이스를 가져야 함
class MyClass<T> where T : IComparable

// 복수 타입 파라미터 제약
class MyClass<T, V> where T : struct
                    where V: class

// 여러 개 제약
class PersonManager<T> where T : Person, IComparable<T>, new()
```

제네릭 불변성/가변성

- C# 3.0까지 제네릭은 불변성 (Invariance)만 지원
  - 불변성(Invariance)는 동일 타입만 사용
- C# 4.0부터 제네릭은 불변성 (invariance)와 가변성 (variance)를 모두 지원
  - 가변성(variance)이란 암묵적인 레퍼런스 변환(implicit reference conversion)을 가능하게 하는 기능
  - 가변성은 공변성(covariance)와 반공변성(contravariance)로 나뉨
  - 공변성(covariance)은 하위타입에서 상위타입으로 레퍼런스를

```
IEnumerable<Derived> d = new List<Derived>();
IEnumerable<Base> b = d;
```

제네릭 불변성/가변성

- C# 4.0부터 제네릭은 불변성 (invariance)와 가변성 (variance)를 모두 지원
  - 반공변성(contravariance)는 반대로 상위타입에서 하위타입으로 레퍼런스를 변환할 수 있는 것

```
Action<Base> b = t => { Console.WriteLine(t.GetType().Name); }
Action<Derived> d = b;
```

제네릭
불변성/가변성

성질	설명	예시

불변성 (Invariance)	• 동일 타입만 사용가능
	• 상/하위 타입으로 레퍼런스 변환 불가
		interface IDepositor<T> { void Store(T t); }
		interface IRetriever<T> { IEnumerable<T> Retrieve(); }
		class Storage<T> : IDepositor<T>, IRetriever<T>
		Storage<Dog> d = new Storage<Dog>();
		//Storage<Animal> a = d; // Dog 타입만 가능
		//d.Store(new Animal()); // Dog 타입만 가능
		IRetriever<Dog> dogs = d;
		//IRetriever<Animal> dogs = d; // Dog 타입만 가능

가변성 (Variance)	• 공변성 (covariance) • out 키워드로 지정 • 출력/리턴 타입에 T 사용 • Dog보다 상위타입에 assign 가능	interface IRetriever<out T> { IEnumerable<T> Retrieve(); } Storage<Dog> d = new Storage<Dog>(); IRetriever<Animal> dogs = d; // Dog 상위타입 가능
	• 반공변성 (contravariance) • in 키워드로 지정 • 입력/메소드 인자에 T	interface IDepositor<in T> { void Store(T t); } IDepositor<Animal> a = new Storage<Animal>();

공변성/반공병성 제네릭 인터페이스

공변성(covariance)을 지원하는 제네릭 인터페이스

- `IEnumerable<out T>`
- `IEnumerator<out T>`
- `IQueryable<out T>`
- `IGrouping<out TKey, out TValue>`

반공변성(contravariance)을 지원하는 제네릭 인터페이스

- `IComparer<in T>`
- `IComparable<in T>`
- `IEqualityCompararer<in T>`

Generic Collection

System.Collections.Generic의 클래스와 인터페이스

- List<T>
- Dictionary<TKey, TValue>
- HashSet<T>
- LinkedList<T>
- Stack<T>
- Queue<T>
- SortedList<TKey, TValue>

- IEnumerable<T>, IEnumerator<T>
- ICollection<T>
- IList<T>
- IDictionary<TKey, TValue>, IDictionaryEnumerator<TKey, TValue>
- ISet<T>

List Class

- 인덱스로 액세스할 수 있는 개체 목록
- 목록의 검색(Find), 정렬(Sort) 및 조작에 사용

```
public class List<T>: ICollection<T>, IEnumerable<T>, IList<T>,
                      IReadOnlyCollection<T>, IReadOnlyList<T>, IList
```

List Class

List<T> 메소드

- `public void Sort()`
- `public void Sort(Comparison<T> comparison)`
- `public void Sort(IComparer<T> comparer)`
- `public void Sort(int index, int count, IComparer<T> comparer)`
- `public List<TOutput> ConvertAll<TOutput>(Converter<T,TOutput> converter)`
  - 리스트 객체의 각 원소를 `TOutput` 형으로 변환하여 리스트로 반환
- `public bool Exists(Predicate<T> match)`
  - 리스트에 있는 모든 원소 중 `match` 조건을 만족하는 원소가 있는지 여부를 반환

List Class

List<T> 메소드

- `public T Find(Predicate<T> match)`
  - 리스트에 있는 모든 원소 중 `match` 조건을 만족하는 첫번째 원소를 반환
- `public List<T> FindAll(Predicate<T> match)`
  - 리스트에 있는 모든 원소 중 `match` 조건을 만족하는 모든 원소를 리스트로 반환
- `public int FindIndex(Predicate<T> match)`
  - 리스트에 있는 모든 원소 중 `match` 조건을 만족하는 첫번째 원소의 인덱스를 반환
- `public int FindLastIndex(Predicate<T> match)`
  - 리스트에 있는 모든 원소 중 `match` 조건을 만족하는 마지막 원소의 인덱스를 반환

List Class

List<T> 메소드

- `public void ForEach(Action<T> action)`
  - 리스트에 있는 모든 원소에 대해 `action`을 수행
- `public bool TrueForAll(Predicate<T> match)`
  - 리스트에 있는 모든 원소가 `match` 조건을 만족하는 지 여부를 반환

List 예시

```
List<Part> parts = new List<Part>();
parts.Add(new Part { Name = “arm”, Id=1234 });
parts.Add(new Part { Name = “leg”, Id=567 });
parts.Add(new Part { Name = “foot”, Id=12345 });
parts.Add(new Part { Name = “hand”, Id=7890 });
foreach(Part p in parts)
  Console.WriteLine(p);
Console.WriteLine(
      parts.Contains(new Part { Name = “leg”, Id=567 }));
parts.Insert(2, new Part { Name = “knee”, Id=78 });
foreach(Part p in parts)
  Console.WriteLine(p);
parts.RemoveAt(3);
foreach(Part p in parts)
  Console.WriteLine(p);
Console.WriteLine();
```

```
ID: 1234 Name: arm
ID: 567 Name: leg
ID: 12345 Name: foot
ID: 7890 Name: hand

True

ID: 1234 Name: arm
ID: 567 Name: leg
ID: 78 Name: knee
ID: 12345 Name: foot
ID: 7890 Name: hand

ID: 1234 Name: arm
ID: 567 Name: leg
ID: 78 Name: knee
ID: 7890 Name: hand
```

List 예시

```
class Part {
        public string Name { get; set; }
        public int Id { get; set; }
        public override string ToString() {
            return "ID: " + Id + "   Name: " + Name;
        }
        public override bool Equals(object obj) {
            if (obj == null) return false;
            Part objAsPart = obj as Part;
            if (objAsPart == null) return false;
            else return Equals(objAsPart);
        }
        public override int GetHashCode() {
            return Id;
        }
        public bool Equals(Part other) {
            if (other == null) return false;
            return (this.Id.Equals(other.Id));
        }
}
```

Remove elements from a list while iterating

- List<T>는 RemoveAt(int index) 또는 Remove(T item) 메소드를 제공함
- List에서 모든 요소를 지워야 할 경우, for-loop에서 **역순 Remove** 하거나 **list.ToList()** 사용한 Remove

```
var list = new List<int>(Enumerable.Range(1, 10));
foreach (int i in list)
    Console.WriteLine(i);
for (int i = list.Count - 1; i >= 0; i--)
   list.RemoveAt(i);
foreach (int i in list)
    Console.WriteLine(i);
```

```
var list = new List<int>(Enumerable.Range(1, 10));
foreach (int i in list)
    Console.WriteLine(i);
foreach (int item in list.ToList())
    list.Remove(item);
foreach (int i in list)
    Console.WriteLine(i);
```

Remove elements from a list while iterating

```
var list = new List<int>(Enumerable.Range(1, 10));
for (int i = 0; i < list.Count; i++) {
   list.RemoveAt(i); // 원소삭제될 때 list 사이즈가 줄면서 다른 원소 index도 바뀜
}
foreach (int i in list) Console.WriteLine(i);
```

```
var list = new List<int>(Enumerable.Range(1, 10));
foreach (int item in list) {
   list.Remove(item);  // throws `InvalidOperationException 발생
}
foreach (int i in list) Console.WriteLine(i);
```

```
var list = new List<int>(Enumerable.Range(1, 10));
IEnumerator<int> iter = ilist.GetEnumerator();
while (iter.MoveNext()) { // throws `InvalidOperationException 발생
   int item = iter.Current;
   list.Remove(item);
}
foreach (int i in list) Console.WriteLine(i);
```

Dictionary Class

- Dictionary<TKey, TValue>은 Hashtable 제네릭 버전
- DictionaryEntry 대신 제네릭 구조체인 KeyValuePair<TKey, TValue>를 열거형에 사용

```
public class Dictionary<TKey, TValue>: ICollection<KeyValuePair<TKey, TValue>>,
                                       IDictionary<TKey, TValue>,
                                       IEnumerable<KeyValuePair<TKey, TValue>>,
                                       IReadOnlyCollection<KeyValuePair<TKey, TValue>>,
                                       IReadOnlyDictionary<TKey, TValue>,
                                       IDeserializationCallback, ISerializable
```

Dictionary 예시

```
Dictionary<int, string> dict = new Dictionary<int, string>();
dict.Add(1, “Dooly”);
dict.Add(3, “Heedong”);
dict.Add(2, “Gildong”);
dict[4] = “Tochi”;
if (!dict.ContainsKey(5))
    dict.Add(5, “Douner”);
foreach(KeyValuePair<int, string> d in dict)
    Console.WriteLine(“Key={0} Value={1}”, d.Key, d.Value.ToString());
Console.WriteLine(“After remove Dooly”);
dict.Remove(1); // Key=1인 둘리 삭제
foreach(KeyValuePair<int, string> d in dict)
    Console.WriteLine(“Key={0} Value={1}”, d.Key, d.Value.ToString());
Console.WriteLine();
```

```
Key=1 Value=Dooly
Key=3 Value=Heedong
Key=2 Value=Gildong
Key=4 Value=Tochi
Key=5 Value=Douner
After remove Dooly
Key=3 Value=Heedong
Key=2 Value=Gildong
Key=4 Value=Tochi
Key=5 Value=Douner
```

HashSet Class

HashSet<T>은 중복 요소를 포함하지 않는(고유한) 요소의 정렬되지 않은 컬랙션

```
public class HashSet<T>: ICollection<T>, IEnumerable<T>, ISet<T>,
                         IReadOnlyCollection<T>, IReadOnlyList<T>,
                         IDeserializationCallback, ISerializable
```

HashSet 예시

```
HashSet<int> even = new HashSet<int>(new int[] { 2, 4, 6});
HashSet<int> odd = new HashSet<int>(new int[] {1, 3, 5, 7});
foreach(int i in even)
    Console.WriteLine(i); // 2 4 6
foreach(int i in odd)
    Console.WriteLine(i); // 1 3 5 7

HashSet<int> all = new HashSet<int>(even);
all.UnionWith(odd);
foreach(int i in all)
    Console.WriteLine(i); // 2 4 6 1 3 5 7

all.IntersectWith(odd);
foreach(int i in all)
    Console.WriteLine(i); // 1 3 5 7
```

yield

- C# yield 키워드는 호출자에게 컬랙션 데이터를 하나씩 리턴할 때 사용
- yield return 또는 yield break의 2가지 방식으로 사용
  - `yield return`은 컬랙션 데이터를 하나씩 리턴하는 데 사용
  - `yield break`는 리턴을 중지하고 `iteration` 루프를 빠져 나올 때 사용

```
public class YieldTest {
   static IEnumerable<int> GetNumber() {
      yield return 10; // 첫번째 루프에서 리턴되는 값
      yield return 20; // 두번째 루프에서 리턴되는 값
      yield return 30; // 세번째 루프에서 리턴되는 값
   }
   public static void Main(string[] args) {
      foreach(int num in GetNumber())
         Console.WriteLine(num);
      Console.WriteLine();
   }
}
```

```
10
20
30
```

yield

- yield return 또는 yield break의 2가지 방식으로 사용
  - `yield return`은 컬랙션 데이터를 하나씩 리턴하는 데 사용
  - `yield break`는 리턴을 중지하고 `iteration` 루프를 빠져

```
public class YieldTest {
   static IEnumerable<int> GetNumber() {
      int num = 0;
      while (true) {
         num++;
         yield return num; // 루프에서 리턴되는 값
         if (num >= 100) {
            yield break; // 루프를 벗어남
         }
      }
   }
   public static void Main(string[] args) {
      foreach(int num in GetNumber())
         Console.WriteLine(num);
      Console.WriteLine();
   }
}
```

```
1
2
3
...
100
```

Custom 클래스에 대한 Sort 사용

개인적으로 만든 클래스에 대해서 컬렉션에 추가하고, Sort를 이용해서 정렬하고 싶다면 IComparable 인터페이스를 구현해주어야 함

```
interface IComparable<T> {
   int CompareTo(T o);
}
```

Custom 클래스에 대한 Sort 사용

- CompareTo(T o) 메소드는 현 객체를 인자로 주어진 o와 비교해서 순서를 정한 후에 정수(int) 값을 반환함
  - 만약 현 객체가 주어진 인자보다 작다면 음수를 반환
  - 만약 현 객체가 주어진 인자와 동일하다면 0을 반환
  - 만약 현 객체가 주어진 인자보다 크다면 양수를 반환

Custom 클래스에 대한 Sort 사용

```
class A : IComparable<A> {
    public string str { get; set; }
    public int num { get; set; }

    public A(String str, int num) {
        this.str = str;
        this.num = num;
    }

    public override string ToString() {
        return this.str + “,” + this.num;
    }
```

Custom 클래스에 대한 Sort 사용

```
public int CompareTo(A a) {
        if (this.str.CompareTo(a.str) == 0) {
            if (this.num > a.num)
                return 1;
            else if (this.num < a.num)
                return -1;
            else
                return 0;
        }
        else {
            return this.str.CompareTo(a.str);
        }
    }
}
```

Custom 클래스에 대한 Sort 사용

```
List<A> list = new List<A>();
list.Add(new A("Kim", 30));
list.Add(new A("Cho", 25));
list.Add(new A("Cho", 15));
list.Add(new A("Lee", 20));
foreach (A a in list)
    Console.WriteLine(a);

Console.WriteLine("After sorting");
list.Sort();
foreach (A a in list)
    Console.WriteLine(a);
```

```
After sorting
Cho, 15
Cho, 25
Kim, 30
Lee, 20
```

== & Equals & Hash code

Equals는 두 객체의 **내용이 같은지** 동등성(Equality)을 비교하는 연산자

```
Person p1 = new Person("Jason", 10);
Person p2 = new Person("Jason", 10);
Person p3 = p1;

// ==
if (p1 == p2) Console.WriteLine("p1 == p2");
else Console.WriteLine("p1 != p2"); //동일한 ref 아니므로 p1 != p2
if (p1 == p3) Console.WriteLine("p1 == p3"); // 동일한 ref므로 p1 == p3
else Console.WriteLine("p1 != p3");

// Equals
// Equals override 안되어 있으면 false
if (p1.Equals(p2)) Console.WriteLine("p1 equals p2");
else Console.WriteLine("p1 is not equal to p2");
```

== & Equals & Hash code

```
class Person : IEquatable<Person> {
        public string Name { get; set; }
        public int Age { get; set; }

        public override string ToString() {
            return Name + ", " + Age;
        }

        public override bool Equals(object obj) {
            if (obj == null) return false;
            Person p = obj as Person;
            if (p == null) return false;
            else return Equals(p);
        }

        public bool Equals(Person other) {
            if (other == null) return false;
            return (this.Name.Equals(other.Name) && (this.Age == other.Age));
        }
```

== & Equals & Hash code

```
// ==
        public static bool operator == (Person p, Person q) {
            return p.Equals(q);
        }

        // !=
        public static bool operator != (Person p, Person q) {
            return !p.Equals(q);
        }

        public override int GetHashCode() {
            return base.GetHashCode();
        }
```

== & Equals & Hash code

IEqutable & == & != operator overriding 하고 나면 true

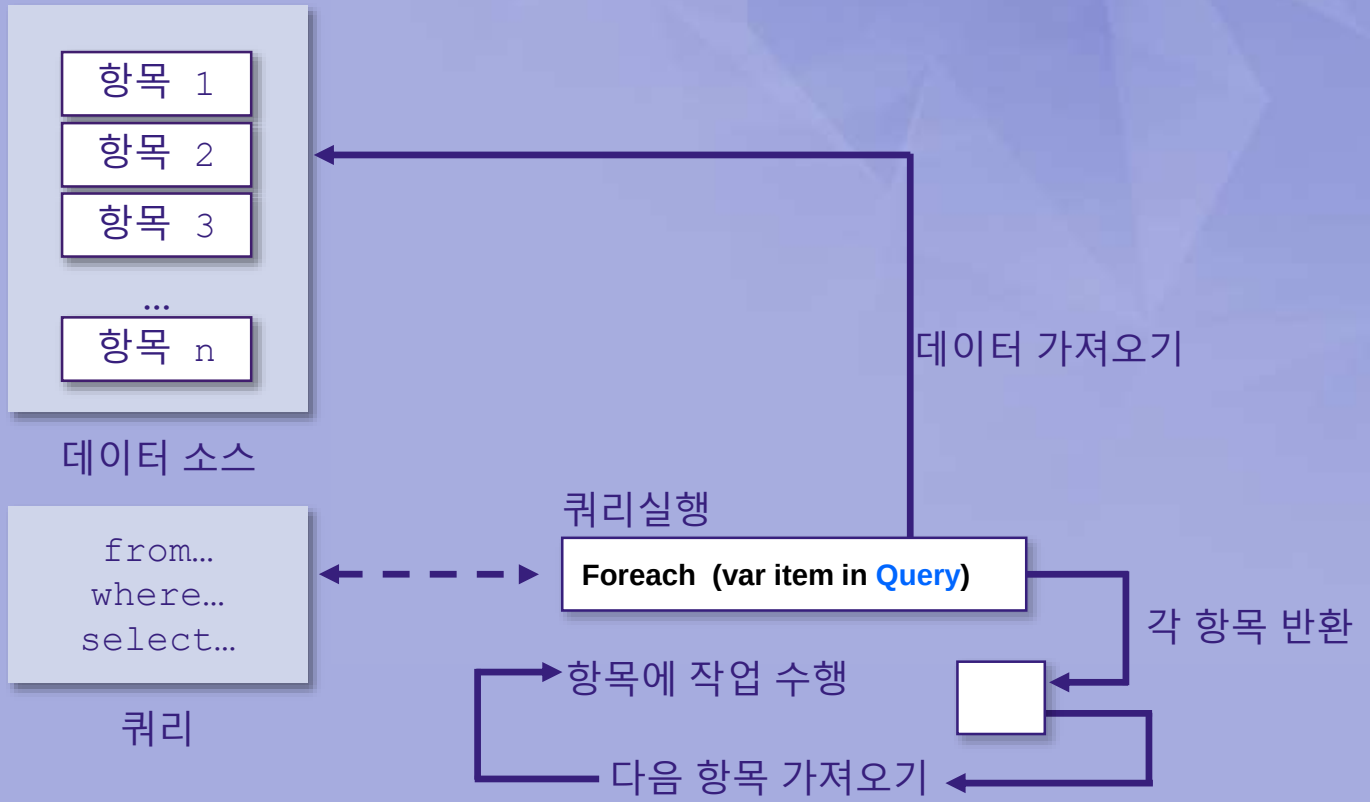
```
Person p1 = new Person(“Jason”, 10);
Person p2 = new Person(“Jason”, 10);
Person p3 = p1;

// ==
// IEquatable과 == & != 오버로딩후 p2 true
if (p1 == p2) Console.WriteLine(“p1 == p2”); // true
else Console.WriteLine(“p1 != p2”);
if (p1 == p3) Console.WriteLine(“p1 == p3”); // 동일한 ref므로 p1 == p3
else Console.WriteLine(“p1 != p3”);

// Equals
// Equals 오버로딩후 p1 Equals p2 true
if (p1.Equals(p2)) Console.WriteLine(“p1 equals p2”);
else Console.WriteLine(“p1 is not equal to p2”);
```

C# 3.5 LINQ(Language Integrated Query) 쿼리

- 쿼리는 데이터 소스에서 데이터를 검색하는 식
- 배열/데이터베이스에서 조건에 맞는 자료만 뽑는 기능을 제공
- **IEnumerable<T> (instance) = from (자동변수)**
  in (배열/DB)
  where (조건)
  select (리턴값)

```
// 1. Data source
   int[] numbers = new int[7] { 0, 1, 2, 3, 4, 5, 6 };
// 2. Query creation (짝수만 선택)
   var numQuery = from num in numbers
                  where (num % 2) == 0 select num;
// 3. Query execution
   foreach (int num in numQuery)
      Console.WriteLine("{0,1}", num);
```

LINQ


항목 1
항목 2
항목 3
...
항목 n
데이터 소스
from...
where...
select...
쿼리
쿼리실행
Foreach (var item in Query)
데이터 가져오기
각 항목 반환
항목에 작업 수행
다음 항목 가져오기

데이터 소스 (DataSource)

- 데이터소스는 `IEnumerable<T>`나 `IQueryable<T>` 인터페이스에서 파생된 것이면 `LINQ`를 사용하여 쿼리 가능한 형식
- 테이터소스가 쿼리 가능한 형식으로 존재하지 않는 경우 `LINQ`

```
// Create a data source from an XML document using System.Xml.Linq
    XElement contacts = XElement.Load(@"C:\myContactList.xml");

 // LINQ to SQL
    Northwnd db = new Northwnd(@"C:\northwnd.mdf");
    IQueryable<Customer> custQuery =
                        from cust in db.Customers
                        where cust.City == "London" select cust;
```

필터링

- 가장 일반적인 쿼리 작업은 `boolean` 형식으로 필터를 적용하는 것
- 결과는 **`where`** 절을 사용하여 생성
- **`&&`** 및 **`||`** 연산자를 사용하여 `where` 절에 필요한 만큼의 필터 식을 적용가능

정렬

- **`orderby`**절은 반환된 시퀀스의 요소가 정렬하고 있는 형식의 기본 비교자에 따라 정렬됨

```
var queryLondonCustomers =
                from cust in db.Customers
                where cust.City == “London” && cust.Name == “Devon”
                orderby cust.Name ascending
                select cust;
```

그룹화

- **group**절을 사용하면 지정한 키에 따라 결과를 그룹화 가능
- 그룹화 결과를 참조해야 하는 경우 **into** 키워드 사용하여 계속 쿼리할 수 있는 식별자를 만들어야 함

조인

- 데이터 소스에서 명시적으로 모델링 않 된 시쿼스 간의 연결 생성
- **join**절은 DB 테이블에 직접 작업하는 대신 개체 컬랙션에 대해 작업

선택 (프로젝션)

- **select**절에서 쿼리 결과를 생성하고 각 반환된 요소의 형식

```
var queryCustomers =
        from cust in db.Customers
        group cust by cust.City into custGroup
        where custGroup.Count() > 2 // 3명이상의 고객을 포함하는 그룹만 반환
        orderby custGroup.Key
        select custGroup;
```

표준 쿼리 연산자 확장 메서드

- LINQ의 선언적 쿼리 구문은 컴파일 할 때 CLR에 대한 메서드 호출으로 변환
- 이러한 메서드 호출은 같은 이름을 사용하는 표준 쿼리 연산자 **Where, Select, GroupBy, Join, Max** 및 **Average** 메서드 구문을 직접 사용가능

```
int[] numbers = new int[7] { 0, 1, 2, 3, 4, 5, 6 };
    // Query Syntax
    var numQuery1 = from num in numbers
                    where (num % 2) == 0 orderby num select num;
    // Method Syntax
    var numQuery2 = numbers.Where(num => num%2 != 0).OrderBy(n=>n);
```

```
0
2
4
6
```

```
1
3
5
```

LINQ 예시

```
int[] numbersA = { 0, 2, 4, 5, 6, 8, 9 };
int[] numbersB = { 1, 3, 5, 7, 8 };
var pairs = from a in numbersA from b in numbersB
            where a < b select new { a, b };
Console.WriteLine("Pairs where a < b");
foreach (var pair in pairs) {
    Console.WriteLine("{0} is less then {1}", pair.a, pair.b);
```

```
Pairs where a
< b
0 is less than 1
0 is less than 3
0 is less than 5
0 is less than 7
0 is less than 8
2 is less than 3
2 is less than 5
2 is less than 7
2 is less than 8
4 is less than 5
4 is less than 7
4 is less than 8
5 is less than 7
5 is less than 8
6 is less than 7
6 is less than 8
```

LINQ 예시

```
var n1 = new int[] { 1, 2 };
 var n2 = new int[] { 3, 4 };
 var query1 = n1.Concat(n2);
 var query2 = n2.Concat(n1);
 var query3 = n1.Concat(n2).Concat(n1);
 foreach (var n in query1) {
    Console.WriteLine(n); // enumerates 1, 2, 3, 4
}
Console.WriteLine();
foreach (var n in query2) {
    Console.WriteLine(n); // enumerates 3, 4, 1, 2
}
Console.WriteLine();
n1[0] = 0;
foreach (var n in query1) {
    Console.WriteLine(n); // enumerates 0, 2, 3, 4
}
Console.WriteLine();
foreach (var n in query3) {
    Console.WriteLine(n); // enumerates 0, 2, 3, 4, 0, 2
}
Console.WriteLine();
```

```
1
2
3
4

3
4
1
2

0
2
3
4

0
2
3
4
0
2
```

LINQ 예시

```
Blue is owned by Dooly
Pink is owned by Heedong
Red is owned by Gildong
Green is owned by Tochi
Yellow is owned by Douner
```

```
Person dooly = new Person("Dooly", 1000);
Person heedong = new Person("Heedong", 3);
Person gildong = new Person("Gildong", 50);
Person tochi = new Person("Tochi", 100);
Person douner = new Person("Douner", 200);
List<Person> people = new List<Person>() { dooly, heedong, gildong, tochi, douner };

Pet blue = new Pet("Blue", dooly);
Pet pink = new Pet("Pink", heedong);
Pet red = new Pet("Red", gildong);
Pet green = new Pet("Green", tochi);
Pet yellow = new Pet("Yellow", douner);
List<Pet> pets = new List<Pet>() { yellow, green, blue, red, pink};
var query = from person in people
            join pet in pets on person equals pet.Owner
            select new { OwnerName = person.Name, PetName = pet.Name };
foreach (var petOwner in query) {
  Console.WriteLine($"{petOwner.PetName} is owned by {petOwner.OwnerName}");
}
```