[객체 소멸 관련 내용]

소멸자

C++의 Destructor는 ~Class() {...} 같이 클래스명 앞에 ~를 붙여 선언되며, 클래스 객체가 소멸될 때 리소스를 해제하기 위해 사용된다.

C#의 Destructor는 C++와 구분하기 위해 흔히 Finalizer로 불리우며, C++와 같이 ~Class() {....} 문장을 사용한다. 하지만, 근본적으로 다른 점은 이 Finalizer가 언제 실행될 지 개발자가 알 수 없고 결정할 수도 없다는 점이다. 이는 Managed Code의 경우 .NET의 Garbage Collection 메카니즘을 사용하기 때문인데, Finalizer와 관련된 Garbage Collector(GC) 동작을 요약하면 다음과 같다.

C# 클래스에 Finalizer(Destructor) 메서드가 있으면,

단계 1) 해당 객체는 생성자 호출시 Finalziation Queue라고 하는 GC안의 내부 큐에 해당 객체 레퍼런스가 추가된다.

단계 2) 이 후 Managed Heap이 차서 Garbage Collection이 필요하게 되면, Finalization Queue에 없는 객체들은 사용 중이 아니면 Garbage로 판단해서 삭제하게 되지만, Finalization Queue에 있는 객체는 이 객체 레퍼런스를 Freachable Queue라고 하는 다른 내부 큐로 이동시키게 된다.

단계 3) 이후, GC의 특별한 Finalization Thread가 이 Freachable Queue에서 객체를 하나씩 꺼내서 Destructor안의 코드를 (컴파일러는 이를 Finalize()메서드 호출로 변경한다) 실행한다.

단계 4) 이렇게 모든 Finalizer 코드들이 실행되고, 다음 Garbage Collection 시기가 오면 GC는 Finalizer가 있던 객체를 이제 Garbage로 인식해서 삭제하게 된다.

C# 클래스에 특정 코드 없이 Destructor를 추가하면 어떤 문제가 있는가?

C# 클래스에 특별한 이유없이 Destructor를 추가하면 해당 객체 혹은 이 객체가 포함하는 내부 객체들이 장기간 존속하게 되고, 객체수가 많은 경우 성능을 저하시키는 원인이 된다. 예를 들어, 배열 10000개를 만들었는데, 각 요소마다 Finalize 메서드를 호출해야 된다면 성능이 저하될 것이다.

GC 활동

1) 객체를 할당하여 할당하는 임계치가 넘어갈 때(각 세대별)
2) GC.Collection 메서드를 호출할 때
3) 시스템 메모리가 부족할 때

비관리 리소스 처리

클래스가 파일 핸들이나 네트워크 연결 같은 Unmanaged 리소스를 가지고 있는 경우, 이 리소스를 해제해 주기 위해 (1) Finalizer (Destructor), (2) Dispose 패턴을 사용할 수 있다.

(1) Finalizer는 Garbage Collector(GC)에 의해 객체 소멸 전에 호출되는 코드 블록으로 이 안에 파일 핸들을 닫거나 연결을 닫는 코드를 넣는다.

```
~MyClass() {
    if (file != null) {
        file.Close();
    }
}
```

이 방식의 문제점은 실행이 GC에 의해 자동으로 결정되기 때문에 언제 이 코드가 실행될 지 모른다는 점이다. 즉, 객체가 다 사용되었는데, 파일은 계속 열려 있는 상태가 되고, GC가 언제 이를 Close할지도 모른다는 것이다. 이러한 문제점을 해결하기 위해 Dispose 패턴을 사용한다.

(2) Unmanaged 리소스가 있는 클래스는 일반적으로 IDisposable이라는 인터페이스를 추가하고, IDisposable.Dispose() 메서드를 구현한다.

```
public void Dispose() {
    if (file != null) {
        file.Close();
    }
    //클래스에 Finalizer가 있을 경우
    GC.SuppressFinalize(this);
}
```

클래스 사용자는 이 Dispose()메서드를 실행하여 GC를 기다리지 않고 리소스를 즉시 해제한다. Dispose가 있더라도 클래스 사용자가 이를 호출하지 않았을 경우, 최후의 보루로 리소스를 해제하고 싶으면 개발자는 Finalizer를 추가할 수 있다. (물론 Finalizer를 추가하면 부작용이 생길 수 있다)

Dispose()메서드 안의 GC.SuppressFinalize를 왜 호출하는 이유

클래스가 Finalizer 메서드와 Dispose메서드를 가지고 있을 때, Dispose() 메서드 호출한 후에는 Finalizer코드를 다시 실행할 필요가 없다. 즉, 리소스를 두 번 해제할 필요가 없다. 이 경우 GC.SuppressFinalize를 호출하여 Finalization Queue안의 해당 객체에 플래그를 표시하여 Garbage Collection시 이 객체가 Freachable Queue로 이동하지 않게 한다. 따라서, Finalize()메서드가 호출되지 않게 하는 것이다.

예제 코드1

```
class Point
    {
        public Point()
        {
            Console.WriteLine("Point()");
        }

        ~Point()
        {
            Console.WriteLine("~Point()");
        }

    }
```

```
static void Main(string[] args)
        {
            Point pt = new Point();   // 객체 생성
            pt = null;                // 객체 null 할당

            Console.WriteLine("GC.Collect()");

            GC.Collect();   // 가비지 콜렉터 부른다.

            // 가비지 콜렉팅이 끝날 때까지 기다린다.
            GC.WaitForPendingFinalizers();

            Console.WriteLine("프로그램 종료");

        }
```

- 사용이 끝난 객체는 null 로 처리하여 관리힙에 저장된 객체를 가비지화 시킨다.

예제 코드2

```
class TestObj
    {
        public string name;  // 인스턴스 이름

        public TestObj( string name ) // 생성자
        {
            this.name = name;
            Console.WriteLine( name + " 생성 " );
        }

        ~TestObj() // 소멸자
        {
            Console.WriteLine( name + " 소멸" );
            Close(); // 리소스 정리
        }

        public void Close()
        {
            Console.WriteLine( name + " 리소스 풀기" );
        }

        public void Dispose()
        {
            Console.WriteLine(name + " Dispose()");
            Close(); // 리소스 풀어준다.

            GC.SuppressFinalize(this);  // 소멸자를 부르지 않습니다.
        }
    }
```

```
static void Main(string[] args)
        {
            TestObj obj = new TestObj( "[Obj]" ); // 객체 생성

            // --- obj 객체 사용

            obj.Dispose(); // 리소스 정리
        }
```

```
static void Main(string[] args)
        {
            using (TestObj obj = new TestObj("[Obj]")) // 객체 생성
            {

                // --- obj 객체 사용

            //    obj.Dispose(); // 리소스 정리
            }

        }
// IDisposable 인터페이스 상속 필요!
```

- Dispose() 함수 호출을 통해 리소스 종료 처리가 완료되면 반드시 GC.SuppressFinalize(this) 함수를 호출한다.

[세대별 가비지 컬렉션 알고리즘]

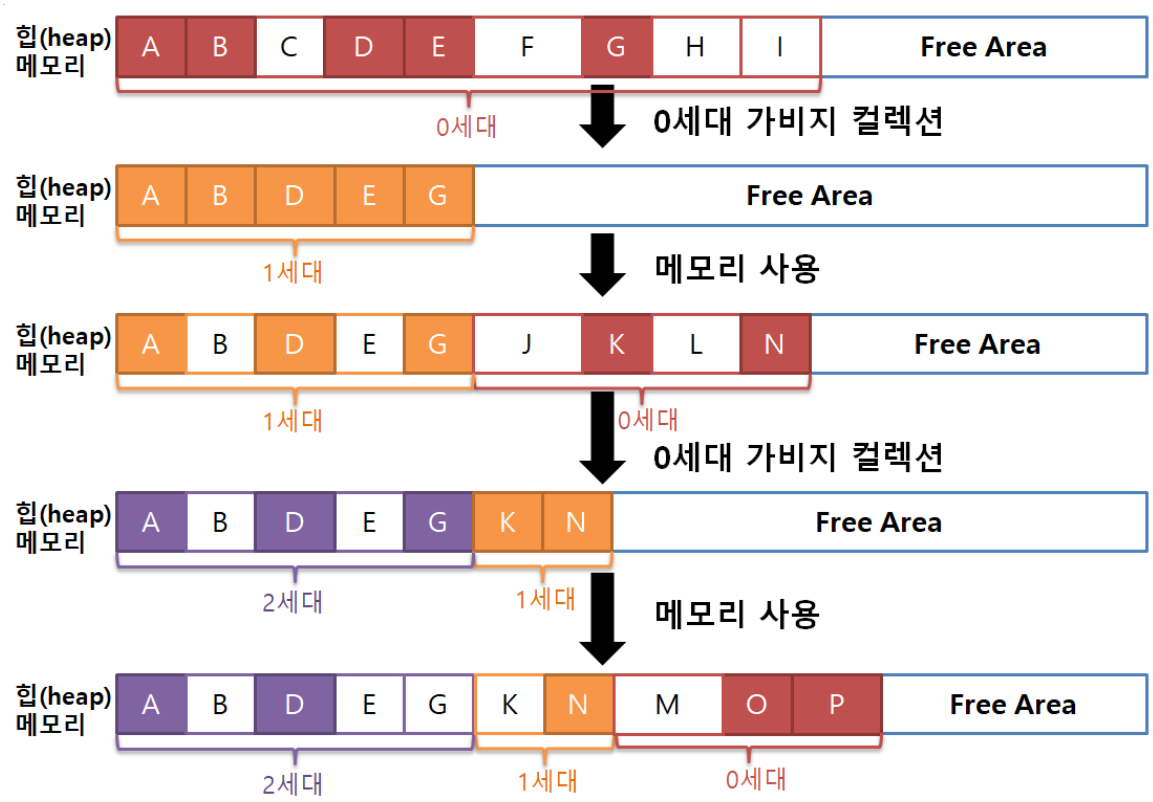
힙 영역에 할당된 메모리들은 0세대, 1세대, 2세대 세가지로 구분된다.

가장 처음 할당되는 메모리들은 0세대이며 가비지 컬렉션이 한번이 이루어 질때마다 한 세대씩 증가한다.

가비지 컬렉션은 0세대 가비지 컬렉션 부터 발생하며 0세대 가비지 컬렉션은 0세대 메모리들만을 메모리 해제 한다. 남은 메모리들은 1세대씩 증가한다.

0세대 가비지 컬렉션이 이루어 졌는데도 메모리가 부족하다 판단되면 1세대 가비지 컬렉션이 발생한다.

그러면 0세대 메모리와 1세대 메모리 모두 사용하지 않는 메모리는 해제된다. 그리고 한 세대씩 증가하게 된다.

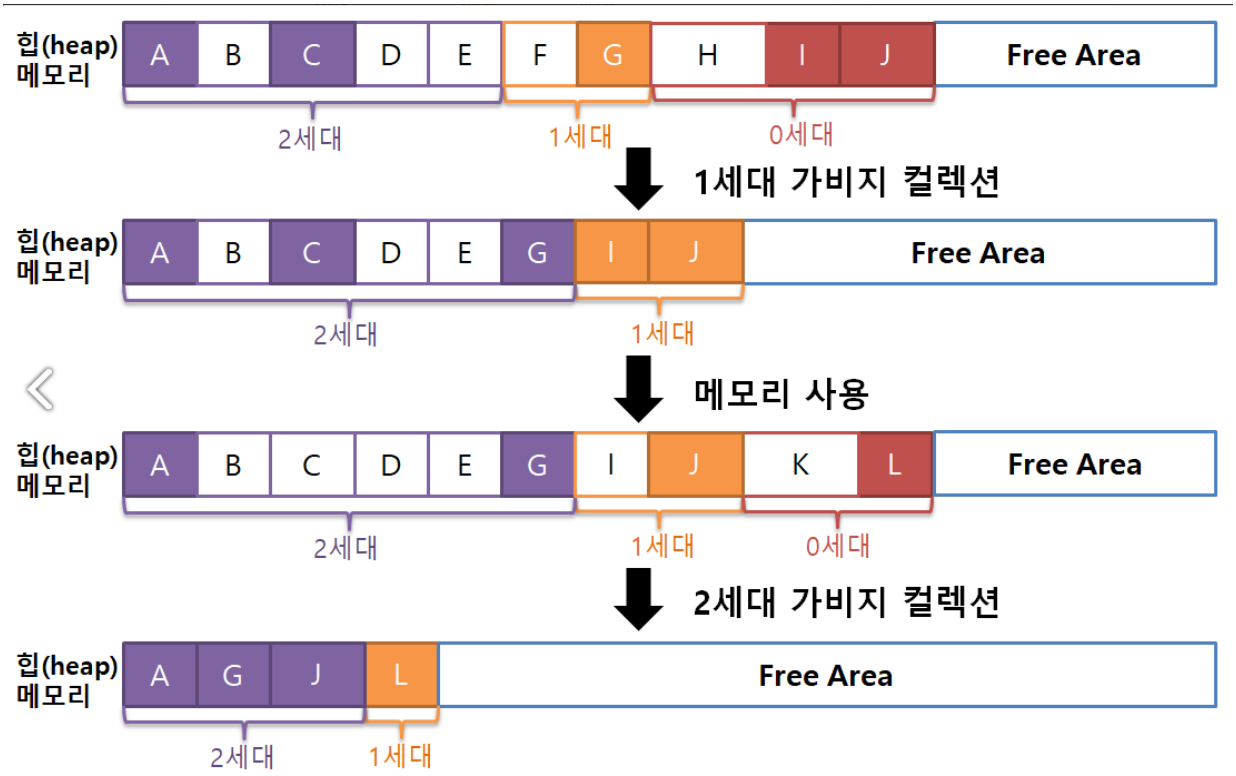


힙 영역에 할당된 메모리들은 0세대, 1세대, 2세대 세가지로 구분된다.

가장 처음 할당되는 메모리들은 0세대이며 가비지 컬렉션이 한번이 이루어 질때마다 한 세대씩 증가한다.

가비지 컬렉션은 0세대 가비지 컬렉션 부터 발생하며 0세대 가비지 컬렉션은 0세대 메모리들만을

아래는 1세대와 2세대 가비지 컬렉션 동작 원리이다.



위의 세대별 가비지 컬렉션 알고리즘으로 인해 생성과 삭제가 빈번한 객체에 대해서만

가비지 컬렉터가 신경을 쓸 수 있게 하는 메커니즘이 완성되었다.

하지만 2세대가 포화되게 된다면 전체 가비지 컬렉션이 수행된다는 항목에 주의를 기울여야 한다.

전체 가비지 컬렉션이 수행되면 CLR은 앱의 실행을 잠시 멈추고 전체 가비지 컬렉션을 수행함으로써

여유 메모리를 확보하려 한다. 앱이 차지하고 있던 메모리가 크면 클수록 전체 가비지 컬렉션의 활동이 방대해 진다.

```
class AnyClass : IDisposable
    {
        string name;

        public AnyClass() { }
        public AnyClass(string name)
        {
            this.name = name;
        }

        ~AnyClass()
        {
            Console.WriteLine(name + " 의 소멸자 호출");

            // 코드내에서 Dispose()를 호출하지 않았을 경우 GC에 의해서 Dispose()를 호출하도록 지정
            Dispose();
        }

        public void Dispose()
        {
            // 비관리 리소스에대한 메모리 해제를 여기서 진행
            Console.WriteLine(name + " 의 Dispose() 호출");

            GC.SuppressFinalize(this);
        }

        public override string ToString()
        {
            return name;
        }
                                                    }
```

```
static void Info(AnyClass[] objects)
        {
            Console.WriteLine("-------------------------------------------------------");
            Console.WriteLine("managed heap memory size : "
                + GC.GetTotalMemory(false).ToString() + "byte");
            Console.WriteLine();

            foreach (AnyClass obj in objects)
            {
                if(obj != null)
                    Console.Write("{0} ", obj.ToString());
            }
            Console.WriteLine();
            foreach(AnyClass obj in objects)
            {
                if(obj != null)
                    Console.Write("{0} ", GC.GetGeneration(obj));
            }
            Console.WriteLine();
            Console.WriteLine();
        }


        static void Main(string[] args)
        {
```

```
            Console.WriteLine("managed heap memory size : "
                + GC.GetTotalMemory(false).ToString() + "byte");
            Console.WriteLine();

            // managed heap에 객체 생성
            AnyClass[] objects = new AnyClass[9];
            for(int i=0; i<objects.Length; i++)
            {
                char value = (char)(i+'A');
                objects[i] = new AnyClass(value.ToString());
            }

            Info(objects);
            Console.ReadKey();


            //C, F, H, I 가비지 처리
            objects[2] = objects[5] = objects[7] = objects[8] = null;

            GC.Collect();     //0세대 가비지 컬렉션 ----------------------------------------------

            Info(objects);
            Console.ReadKey();

        }
```

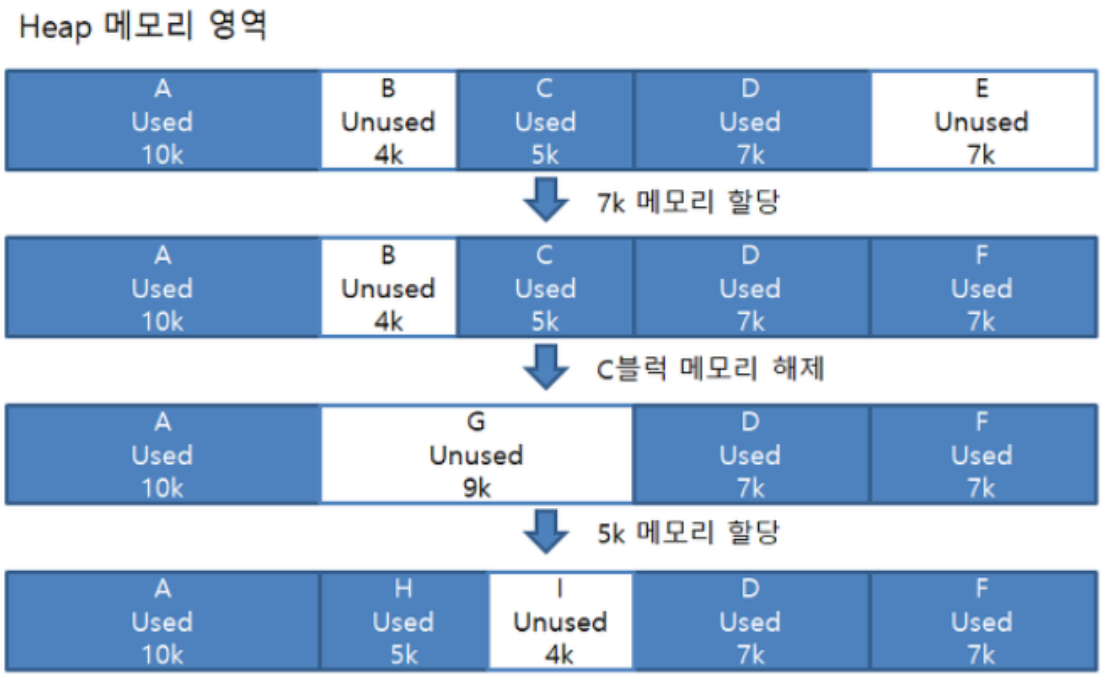
LOH(Large Object Heap)의 가비지 컬렉션 방식

C#에 CLR(Common Language Runtime) 에서는 객체의 크기에 따라 메모리를 크게 두가지로 나눠서 관리 한다.

- 용량이 작은 객체는 (85KB 미만) SOH(Small Object Heap)
- 용량이 큰 객체는 (85KB 이상) LOH(Large Object Heap)

가비지 컬랙터가 두 힙(Heap)에 대해서 서로 다른 방식으로 관리 한다.

- SOH의 경우는 3가지 세대로 나눠서 가비지 컬랙션을 진행한다
- LOH의 경우는 C와 C++의 메모리 관리 방식과 비슷하게 관리한다.



다른 것은 가장 작은 객체의 사이즈가 85KB 이상이란 것이고 메모리 해제를 가비지 컬렉터가 알아서 한다는 것이다.

SOH와 LOH 가장 큰 차이점은 다음과 같다.

1. SOH는 가비지 컬렉션이 발생하면 해제된 메모리 공간들에 사용중인 메모리들로 재배치하지만 LOH는 메모리해제를 해도 해제된 공간은 그대로 둔다.

이는 메모리 사이즈가 큰 LOH들의 경우 메모리의 위치를 재배치하는 것은 오버해드가 크기 때문이다. 그때문에 메모리 단편화가 일어날 수 있다.

2. SOH는 0세대, 1세대, 2세대로 3가지 세대로 나눠서 메모리 관리를 하지만 LOH는 2세대밖에 존재하지 않는다.

그래서 2세대 가비지 컬렉션이 일어날 때에만 사용하지 않는 메모리를 해제 시킨다. 하지만 이때 많은 오버헤드가 발생하게 된다.

LOH때문이라도 2세대 가비지 컬렉션이 되도록 일어나지 않도록 주의해서 객체를 생성해야 한다.
그리고 메모리 단편화 때문에 용량이 큰 객체는 너무 자주 생성하고 해제를 반복하면 안된다.
최악의 경우 메모리 할당에 실패하는 경우가 발생한다.
그래서 큰 메모리를 자주 할당하는 것은 주의가 꼭 필요하다.