C#
-CAHPTER13-

SOUL SEEK

목차

가지비 컬렉션

1. 가지비 컬렉션(GC)

가비지 컬렉터

- **CLR**이 제공하는 자동 메모리 관리**(Automatic Memory Management)** 기능
  ➔가비지 컬렉션**(Garbage Collection)**이 가장 핵심적인 기능이다**.**
- 사용하지 않는 객체인지 사용하는 객체인지 구분해서 사용하지 않는 것만 쓰레기로 간주해서 수거 한다**.**
- 소프트웨어이기 때문에 **CPU** 메모리 자원을 소모한다**.**

가비지 컬렉터의 동작

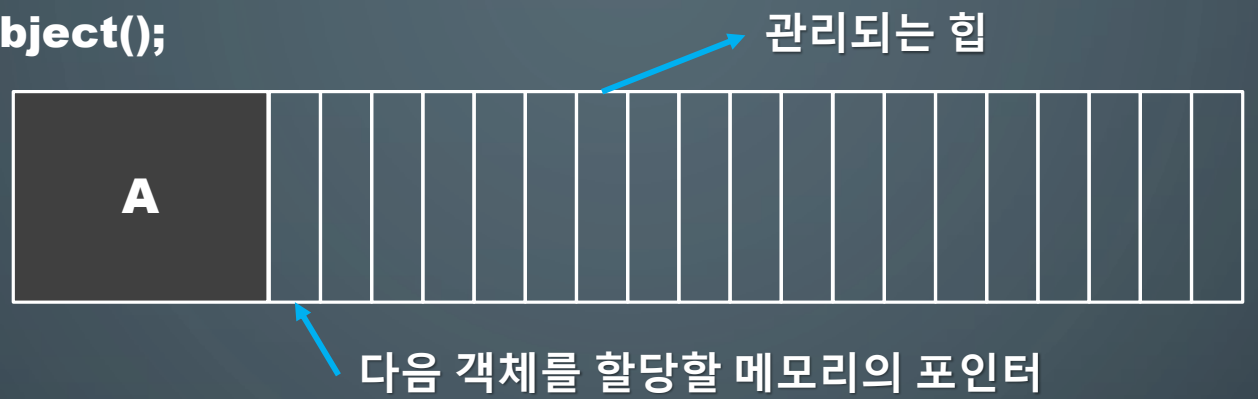
CLR이 객체의 메모리에 어떻게 할당하는 것인지 부터 알아야 한다**.**

- **C#**으로 작성한 소스 코드를 컴파일해서 실행 파일을 만들고 이 실행 파일을 실행하면**, CLR**은 이 프로그램을 위한 일정 크기의 메모리를 확보한다**.**
- ➔전체 메모리는 관리하는 힙**(Managed Heep)**을 마련한다**.**
- **CLR**은 관리되는 힙 메모리의 첫 번째 주소에 **“**다음 객체를 할당할 메모리의 포인터**”**를 위치한다**.**

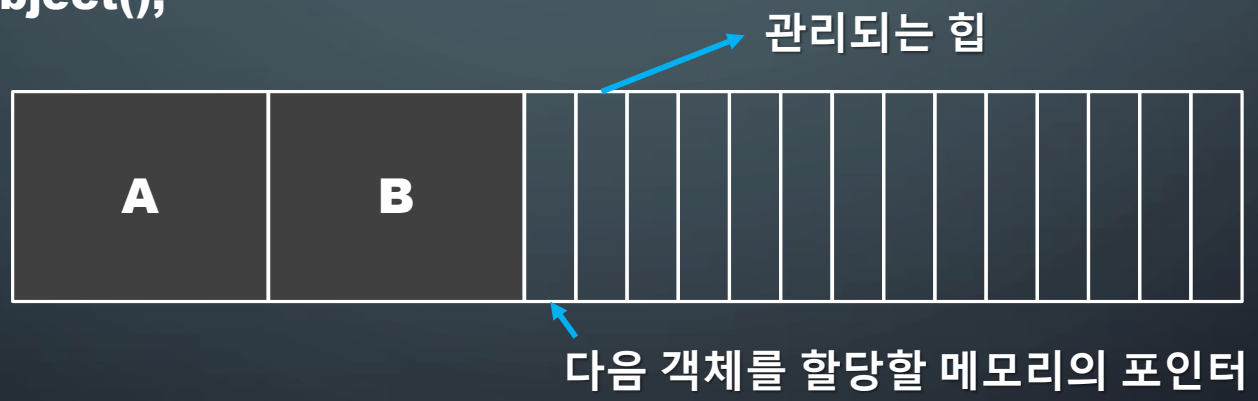
1. 가지비 컬렉션(GC)

A라는 첫 번째 객체를 할당하면 **CLR**은 코드를 실행하면서 **“**다음 객체를 할당할 메모리의 포인터**”**가 가리키고 있는 주소에 **A** 객체를 할당하고 포인터를 **A**객체가 차지하고 있는 공간 바로 뒤로 이동시킨다**.**

object A = new object();



object B = new object();



CLR은 **C-**런타임처럼 메모리는 **LinkedList**로 연결하는 형태의 덩어리로 나누어 놓지 않는다**.** 단지 메모리에 저장될 지점을 가르쳐 주는 역할만 하게 되고 메모리를 뒤지기위해서 **LinkedList**를 탐색하는 시간도 절약이 되며**,** 훨씬 더 호율적이다**.**

1. 가지비 컬렉션(GC)

해제를 해야 하는 객체와 해제 하지 않는 객체를 구분하는 방법.

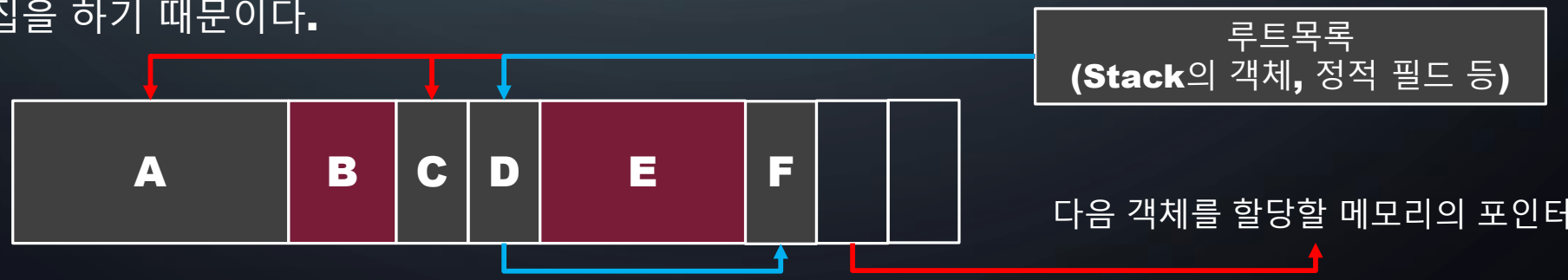
Stack 형식의 객체들은 코드블록이 끝나면 생명을 다하고 사라진다. 하지만 참조**(Heap)**형식의 객체들은 **Heap**에 할당되어 코드 블록과 관계없이 계속 살아남아 있다.

```
if(true)
{
        object a = new object();
}
```

위의 코드를 가지고 각각 어떤 영역에서 어떻게 존재하는지 알아보자.

a는 단지 **new object()**를 하면서 발생하는 객체 **A**의 실제 메모리에 할당된 **Heap** 메모리의 주소를 참조 하고 있을 뿐이다. 그리고 **a**가 코드 블록에 의해 사라지면 더 이상 **A**와의 연결고리는 끊어지게 되고 사용하고 싶어도 사용할 수가 없기 때문에 가리지 컬렉터가 가져가게 된다.
위 예제 코드에서 **a**처럼 할당된 메모리의 위치를 참조하는 객체를 일컬어 루트**(Root)**라고 부른다.
루트는 스택에서 생성될 수도 있고 정적 필드처럼 **Heap**에 생성될 수도 있다.

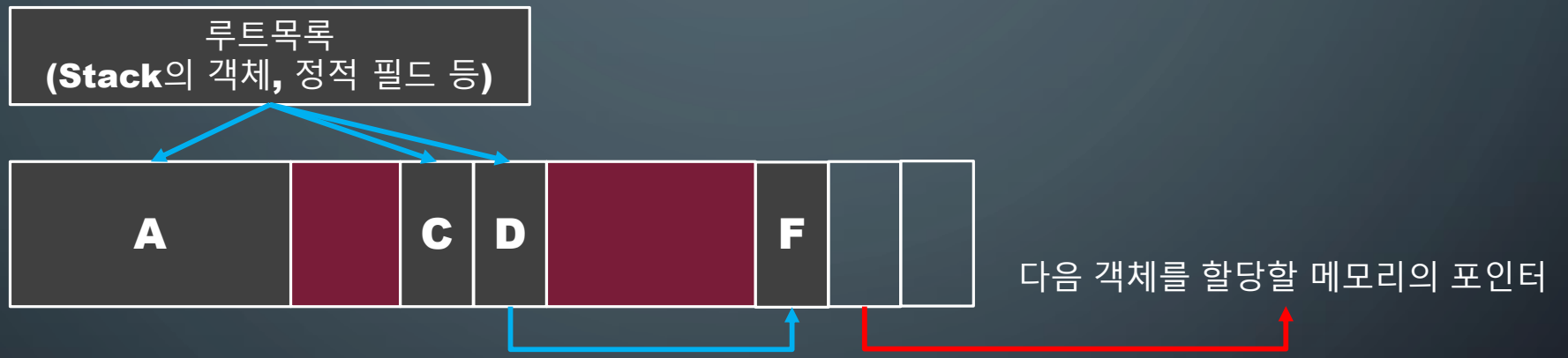
.NET App가 실행되면 **JIT** 컴파일러가 이 루트들을 목록으로 만들고, **CLR**은 이 루트 목록을 관리하며 상태를 갱신한다. 이 루트가 중요한 이유는 가비지 컬렉터가 **CLR**이 관리하고 있던 루트목록을 참조해서 쓰레기 수집을 하기 때문이다.



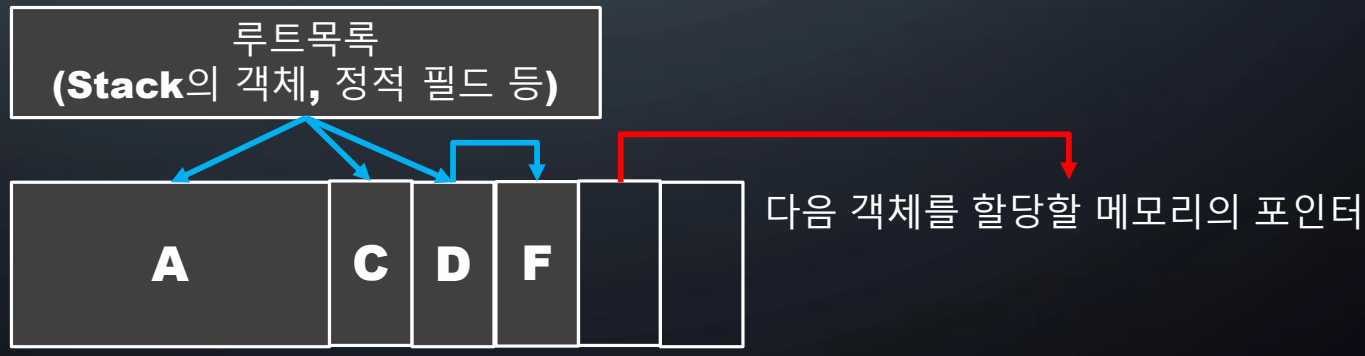
1. 가지비 컬렉션(GC)

가비지 컬렉터가 사용하지 않는(쓰레기)객체를 정리하는 과정 – 기본적인 원리

1. 작업을 하기전에, 가비지 컬렉터는 모든 객체가 쓰레기라고 가정한다. 즉, 루트 목록 내의 어떤 루트도 메모리를 가리키지 않는다고 가정한다.
2. 루트 목록을 순회하면서 각 루트가 참조하고 있는 힙 객체와의 관계 여부를 조사한다, 루트가 참조하고 있는 힙 객체가 또 다른 힙 객체를 참조하고 있다면 이 역시도 해당 루트와 관계가 있는 것으로 판단한다, 이 때 어떤 루트와도 관계가 없는 힙의 객체들은 쓰레기로 확정된다.
3. 쓰레기 객체가 차지하고 있던 메모리는 이제 '비어 있는 공간'이 된다.



4. 루트 목록에 대한 조사가 끝나면, 가비지 컬렉터는 이제 힙을 순회하면서 쓰레기가 차지하고 있던 '비어 있는 공간'에 쓰레기의 인접 객테들을 이동시켜 차곡차곡 채워 넣는다. 모든 객체의 이동이 끝나면 다음과 같이 깨끗한 상태의 메모리는 얻게 된다.



1. 가지비 컬렉션(GC)

세대별 가비지 컬렉션

- 가비지 컬렉션이 성능을 최대한 끌어올리기 위해 동작하는 방법
- **CLR**의 메모리도 구역을 나누어 메모리에서 빨리 해제될 객체와 오래도록 살아남을 것 같은 객체들을 따로 담아 관리한다**.** 구체적으로 이야기하면**, CLR**은 메모리를 **0, 1, 2**의 **3**개 세대로 나누고 **0**세대에는 빨리 사라질 것으로 예상되는 객체들을**, 2**세대에는 오랫동안 살아남을 것으로 예상되는 객체들을 위치시킨다**.**
- **CLR**은 객체의 나이가 어릴수록 메모리에서 빨리 사라지고 나이가 많을 수록 메모리에 오랫동안 살아 남는다고 간주한다**.**
  ➔가비지 컬렉션을 겪은 횟수를 기준으로 나이를 측정한다**. 0~2**세대 까지 나누어진다**.**

.NET Framework는 **App**가 일을 시작함에 따라 힙을 생성하고 메모리를 할당한다**.**

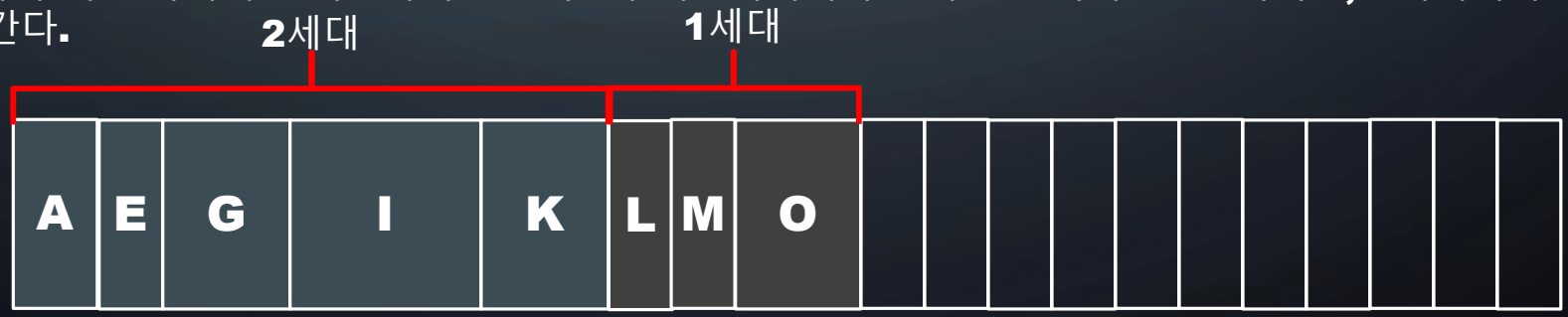
할당된 객체들의 총 크기가 **0**세대 가비지 컬렉션 임계치에 도달하면 가비지 컬렉터는 **0**세대에 대해 가비지 컬렉션을 수행하고**,** 여기에서 살아 남은 객체들을 **1**세대로 옮긴다**.**

1. 가지비 컬렉션(GC)

새로운 객체들이 생성되면 0세대로 할당이 된다. 1세대 이전 가비지 컬렉션에서 살아남은 객체들, 새로운 객체들은 0세대로 생성된다.

0세대 객체의 용량이 0세대 가비지 컬렉션 임계치를 넘어섰고, 가비지 컬렉터가 다시 움직여야 할 때가 왔다. 가비지 컬렉션을 수행해서 정리를 하기 시작한다.

0세대는 깨끗하게 비워졌지만 또다시 애플리케이션에 의해 새로운 객체들이 할당된다. 이번에는 1세대도 초과했기 때문에 1세대에 대해 가비지 컬렉션을 수행한다. 이 때 가비지 컬렉터는 하위 세대에 대해서도 가비지 컬렉션을 수행하기 때문에 0세대와 1세대에 대한 가비지 컬렉션이 수행된다. 이 때 0세대에서 살아남은 객체들은 1세대로, 1세대에서 살아남은 2세대로 옮겨간다.



1. 가지비 컬렉션(GC)

또 한 차례의 가비지 컬렉션이 끝났지만, 애플리케이션은 묵묵히 자기의 일을 한다. 그리고 **0**세대가 객체들로 차오르기 시작한다.

각 세대의 메모리 임계치에 따라 가비지 컬렉션이 수행되고, 가비지 컬렉션이 반복됨에 따라 **0** 세대의 객체들은 **1**세대로, **1**세대는 **2**세대로 계속 이동한다. **2**세대는 더 이상 이동할 곳이 없으며 **2**세대도 포화되면 **2**세대에 대한 가비지 컬렉션이 일어나는데 이렇게 되면 **1, 0** 세대도 전부 진행하게 되므로 전체 가비지 컬렉션**(FULL GC)**이 일어난다.

결론,

- 가비지 컬렉션의 빈도는 **2**세대 **< 1**세대 **< 0**세대 순으로 가비지 컬렉션의 빈도가 높다.
- **2**세대의 객체들이 가장 오랫동안 살아남을 확률이 높고, 따라서 가비지 컬렉터도 상대적으로 관심을 덜 주는 편이 된다.
- 그렇게 계속 **2**세대에 계속 쌓여가다가 포화가 되면 **FULL GC**를 발생시키고 차지하는 메모리가 크면 클수록 진행하는데 시간이 오래 걸리므로 애플리케이션은 모든 자원을 활용해 이를 해결하려고 하기 때문에 원래의 목적과 맞지않게 애플리케이션을 방해하게 된다.

1. 가지비 컬렉션(GC)

가비지 컬렉션을 최소화 하자!

1. 객체를 너무 많이 할당하지 말자.
   - 서로가 서로를 참조하는 형식으로 참조하면 연결구간이 남아 있기 때문에 더 이상 사용하지 않고 있어도 **GC**에서 살아남는 상황을 만든다.
2. 너무 큰 객체 할당을 피하자
   - **CLR**은 보통 크기의 객체를 할당하는 힙과는 별도로 **85KB** 이상의 대형 객체를 할당하기 위한 **“대형 객체 힙(LOH : Large Object Heap)”**을 따로 유지한다. **CLR**이 항상 하던 메모리 할당방식과 다르게 **C-**런타임의 할당방식처럼 변하게 된다. 그렇기 때문에 성능저하의 원인이 되기도 하고 **LOH**는 **2**세대 힙으로 간주하기 때문에 **LOH**에 **GC**에 발생하면 **2**세대에 대한 **GC**가 일어나기 때문에 **GC**가 일어난다.
3. 너무 복잡한 참조 관계는 만들지 말자.
   - 힙의 메모리의 포화상태를 너무 과하게 만들어 놓으면 **GC**에 더 많은 자원과 시간을 투자한다.
4. 루트를 너무 많이 만들지 말자.
   - 루트 목록이 작으면 작을 수록 가비지 컬렉터가 검사를 수행하는 횟수가 줄어드므로 더 빨리 가비지 컬렉션을 끝낼 수 있다.