



05 - 08.
함수와 모듈화



목차

- 모듈화
- 함수
 - 문법
 - 함수 오버로딩
 - 인수 전달 방식
- 객체의 속성
 - 범위
 - 수명
- 함수의 동작 원리
 - 스택 프레임
 - 호출 규약
- 설계 가이드
 - 무엇을 함수로 만들 것인가?
 - 어떻게 만들 것인가?
 - 어떤 주석이 필요한가?
- 함수형 프로그래밍

모델화

모듈화

- 코드를 이해하기 위한 문맥의 양이 늘어날수록 프로그램의 복잡성이 증가한다.
- 따라서 복잡성을 낮추기 위해 프로그램 전체를 여러 부분으로 나뉘어야 한다. 이를 모듈화(Modularization)라 한다.
- 단, 잘못된 모듈화는 프로그램의 복잡성을 오히려 증가시킨다.[1] 모듈은 논리적 또는 기능적으로 분리되어 독립적인 일을 수행할 수 있어야 한다.
- 모듈화를 잘하려면 (1) 먼저 전체를 구현할 줄 알아야 하며, (2) 여러 경험을 통해 추상적 및 구조적 사고력을 길러야 한다.
- 프로그래밍 언어에서는 함수와 클래스를 통해 모듈화를 지원한다.

[1]: 즉, 왜 나눴는지 파악이 되지 않고, 나눔으로써 전체를 알기 어려워진 것이다.

함수

함수

- 일련의 과정에 이름을 붙인 것을 **함수**(Function)라 한다.
 - 다른 이름으로는 **메소드**(Method), **서브루틴**(Subroutine), **프로시저**(Procedure) 등이 있다.
- **지식 및 정보의 중복을 제거**하여 프로그램의 모듈성을 높인다.
 - 그래서 함수로 프로그램을 구성하던 방식을 **절차적 프로그래밍**(Procedural Programming)이라 한다.

함수

- 함수를 호출하면 함수 내 모든 코드가 실행된 후 호출 지점으로 돌아온다.

```
class Program
{
    static void Main()
    {
        Foo();

        // Foo()에 있는 모든 명령이 실행된 후 아래 구문이 실행된다.
        Console.WriteLine("Hello Function");
    }

    // 멤버는 인스턴스 멤버와 정적 멤버로 분류된다.
    // 정적 멤버를 의미하는 static은 추후에 배운다.
    static void Foo()
    {
        Console.WriteLine($"{nameof(Foo)}: 1");
        Console.WriteLine($"{nameof(Foo)}: 2");
        Console.WriteLine($"{nameof(Foo)}: 3");
    }
}
```

문법

- 인터페이스[1]를 의미하는 **헤더**(Header)와 함수의 동작을 정의한 **바디**(Body)로 구성된다.
 - 구문이 하나라면 식 본문 정의(Expression Body Definition)를 사용할 수 있다.
- 함수는 매개변수(Parameter)를 통해 데이터를 전달받을 수 있으며, 이때 전달되는 데이터를 인수(Argument)라 한다.
 - 전달받을 데이터가 없을 수 있다. 즉, 매개변수는 없을 수 있다.
 - params 한정자로 **가변 인수**(Variadic Argument)를 처리할 수 있다.
 - 인수를 전달할 때는 명명해서 전달할 수 있다.
- 호출자에게 결과값을 돌려줄 때는 return문을 사용하며, 없는 경우 void 타입을 사용한다.

[1]: 이 맥락에서 **인터페이스**(Interface)는 함수를 사용하기 위한 모든 정보를 말한다. 즉, "이 함수를 쓰려면 뭘 줘야 하고, 뭘 돌려받으며, 이 함수가 무슨 일을 하는가?"에 대한 답을 제공한다.

함수 오버로딩

- 하나의 프로그램 내에서 같은 이름을 가진 함수를 여러 개 정의할 수 있는 것을 함수 오버로딩(Function Overloading)이라 한다.
- 그래서 컴파일러가 함수를 구별하기 위해 함수 시그니처(Function Signature)를 사용한다.
 - 함수 시그니처에는 함수 이름과 매개변수의 타입 및 순서를 포함한다.(e.g. Add(int, int))
 - 달리 말하자면, 반환 타입만 달라서는 오버로딩이 불가능하다.
- 매개변수에 기본값을 지정해 오버로딩을 할 수도 있다.
 - 이때는 가장 오른쪽부터 기본값을 지정해야 한다.

인수 전달 방식

- 인수 전달 방식에 따라 원본에 영향을 줄 수 있으므로 명확히 알고 있어야 한다.
- 값으로 전달(Pass By Value / Call By Value)
 - 객체에 저장된 값을 전달하는 것이다.
 - 원본에 영향을 주지 않는다.
 - 단, 참조 타입의 경우 얇은 복사에 주의한다.
 - 기본 전달 방식이다.
- 참조로 전달(Pass By Reference / Call By Reference)
 - 원본에 접근할 수 있는 참조를 전달한다.
 - 원본에 영향을 준다.
 - ref, in, out 한정자를 사용한다.

인수 전달 방식

```
class Program
{
    static void Foo(int x)
    {
        x = 20; // num에 영향을 주지 않는다.
    }

    static void Boo(int[] x)
    {
        x[0] = 10; // arr에 영향을 준다.
        x = new int[3]; // x가 가리키는 배열이 달라진다.
        x[0] = 7; // arr에 영향을 주지 않는다.
    }

    static void Main()
    {
        int num = 10;
        Foo(num);

        int[] arr = { 1, 2, 3 };
        // 얀 복사로 원본에 영향을 준다.
        Boo(arr);
    }
}
```

얀 복사에 주의한다.

인수 전달 방식

```
class Program
{
    static void Foo(ref int x)
    {
        x = 20; // num도 20이 된다.
    }

    static void Boo(ref int[] x)
    {
        x[0] = 10; // arr에 영향을 준다.
        x = new int[3]; // arr이 가리키고 있는 배열이 바뀐다.
        x[0] = 7;
    }

    static void Main()
    {
        int num = 10;
        Foo(num);

        int[] arr = { 1, 2, 3 };
        Boo(ref arr);
    }
}
```

참조를 전달하면 원본에 영향을 준다.

객체의 속성

객체의 속성

- 객체가 유효한 영역을 **범위**(Scope)라 한다.[1]
 - **지역 범위(Local Scope)**: 객체가 정의된 순간부터 블록({})이 끝날 때까지[2]
 - **인스턴스 범위 (Instance Scope)**: 클래스의 인스턴스를 통해서만 접근 가능
 - **클래스 정적 범위 (Class Static Scope)**: 클래스의 모든 인스턴스에서 접근 가능
- 객체가 유효한 시간을 **수명**(Lifetime)이라 한다.
 - **자동/지역 시간(Automatic/Local Time)**: 객체가 정의된 순간부터 지역 범위를 벗어날 때까지
 - **정적 시간(Static Time)**: 프로그램 실행 전부터 프로그램 끝날 때까지
 - **동적 시간(Dynamic Time)**: 객체가 정의된 순간부터 프로그래머가 명시적으로 해제할 때까지[3]

[1]: 언어마다 범위의 유형은 다 다르다. 파일 범위, 전역 범위, 이름공간 범위처럼 좀 더 세세한 경우도 있다.

[2]: 지역 범위를 가지는 변수를 **지역 변수**(Local Variable)라 부르며, 그 외에는 **전역 변수**(Global Variable)라고 부르는 경향이 있다.

[3]: C#에서는 물론 GC가 이를 수행한다. GC가 없는 언어에서 해제를 깜빡한 경우에는 프로그램의 종료와 함께 운영체제가 모두 해제한다.

함수의 동작 원리

스택 프레임

- 함수가 동작하는 데 필요한 모든 데이터를 스택 프레임(Stack Frame)이라 한다.
 - 지역 변수(Local Variable): 함수 내부에서 정의된 모든 변수
 - 매개변수(Parameter): 호출될 때, 전달받은 데이터
 - 복귀 주소(Return Address): 실행을 마친 후 이어서 실행할 코드의 메모리 주소
 - 저장된 레지스터 값(Saved Register Values): 함수가 실행되며 오염될 수 있는 CPU 레지스터의 데이터
 - 프레임 포인터 및 베이스 포인터(Frame Pointer / Base Pointer): 현재 스택 프레임의 시작 위치로 지역 변수 혹은 매개변수에 접근하는 기준점
- 함수가 호출될 때마다 스택에 새로운 스택 프레임이 추가(Push)되고, 함수 실행이 완료되면 해당 프레임이 다시 제거(Pop)된다.[1]

호출 규약

- 함수의 호출 및 반환이 이뤄질 때 호출자(Caller)와 피호출자(Callee) 사이에 지켜야 할 규칙을 호출 규약(Calling Convention)이라 한다.
 - **인수 전달 방식 및 순서:** 인수를 전달할 때 스택 혹은 레지스터 중 무엇을 이용할 것인가? 어떤 순서로 스택에 넣을 것인가?
 - **반환 값 전달 방식:** 함수의 최종 결과를 호출자에게 어떻게 전달할 것인가?
 - **스택 정리 주체:** 호출자가 정리할 것인가, 피호출자가 정리할 것인가?
 - **레지스터 보존:** 호출된 함수가 자유롭게 사용할 수 있는 레지스터(Caller Saved)와 그렇지 못한 레지스터(Callee Saved)는 무엇인가?
- ARM은 여기, x86-64는 Microsoft 버전과 그 외의 OS에서 사용하는 System V AMD64가 있다.[1]
- 같은 코드라도 함수로 만들면 성능이 저하된다. 즉, 호출 규약으로 인한 오버헤드가 존재한다.[2]
 - 프로로그 파트와 에필로그 파트가 있기 때문이다.[3]

[1]: 원래 32비트 아키텍처 시절만 해도 호출 규약이 굉장히 다양했으나, 64비트 아키텍처가 개발되면서부터는 거의 하나로 합쳐졌다.

[2]: 그러나 컴파일러가 충분히 발달하여 프로파일링을 통해 병목 지점이 되는 것이 아닌 이상 걱정할 필요는 없다.

[3]: x64 아키텍처를 기준으로 후술한다.

호출 규약

- 빨간색 박스는 인수를 전달하는 것을 나타내고, 초록색 박스는 반환값이 전달되는 것을 나타낸다. 문서와 함께 비교해보자.

```
int c = Add(1, 2);
00007FF8A18D4961 mov     ecx,1
00007FF8A18D4966 mov     edx,2
00007FF8A18D496B call    CLRStub[MethodDescPrestub]@7ff8a172e3f8 (07FF8A172E3F8h)
00007FF8A18D4970 mov     dword ptr [rbp+2Ch],eax
00007FF8A18D4973 mov     eax,dword ptr [rbp+2Ch]
00007FF8A18D4976 mov     dword ptr [rbp+44h],eax
```

인수와 반환 값 전달을 위해 사용하는 레지스터는 호출자 보존 레지스터다.

호출 규약

- **프로로그(Prologue):** 함수의 코드를 실행 전 환경 설정

- **피호출자 보존 레지스터 저장:** 함수 종료 후 호출자의 문맥을 복원하기 위한 데이터(RBP, RSP 등)를 저장한다.
- **스택 프레임 생성:** 프레임 포인터를 설정하고, 지역 변수를 위한 공간을 할당한다.
- **반환 레지스터 초기화:** EAX/RAX를 0으로 초기화한다.

```
static int Add(int a, int b)
{
00007FF8A1C28130  push      rbp      경과 시간 1ms 이하
00007FF8A1C28131  push      rdi
00007FF8A1C28132  push      rsi
00007FF8A1C28133  sub       rsp,30h
00007FF8A1C28137  mov       rbp,rsp
00007FF8A1C2813A  xor       eax,eax
00007FF8A1C2813C  mov       dword ptr [rbp+2Ch],eax
00007FF8A1C2813F  mov       dword ptr [rbp+50h],ecx
00007FF8A1C28142  mov       dword ptr [rbp+58h],edx
}
```

호출 규약

- **에필로그(Epilogue):** 함수 반환 전 환경을 정리하고 복원

- **반환 값 설정:** 반환 값을 특정 레지스터(예: EAX/RAX)에 저장한다.
- **스택 프레임 정리:** 지역 변수 공간을 해제한다.
- **피호출자 보존 레지스터 복원:** 호출자의 문맥을 복원한다.
- **제어권 반환:** 복귀 주소를 꺼내 PC에 로드한다.

```
00007FF817581E6C  nop
    return a + b;
00007FF817581E64  mov     eax,dword ptr [rbp+50h]
00007FF817581E67  add     eax,dword ptr [rbp+58h]
00007FF817581E6A  mov     dword ptr [rbp+2Ch],eax
00007FF817581E6D  nop
    }
00007FF817581E6E  mov     eax,dword ptr [rbp+2Ch]
00007FF817581E71  lea     rsp,[rbp+30h]
00007FF817581E75  pop     rsi
00007FF817581E76  pop     rdi
00007FF817581E77  pop     rbp
00007FF817581E78  ret
```

설계 가이드

무엇을 함수로 만들 것인가?

- **DRY(Don't Repeat Yourself) 원칙**

- "반복하지 말라."
- 코드의 물리적 중복이 아니라 정보나 지식의 중복을 방지한다.
- 공통 패턴이 명확하게 나타나기 전까지는 작은 규모의 중복을 허용해도 좋다.[1]

- **SRP(Single Responsibility Principle)**

- "모듈은 변경되는 이유가 하나여야 한다."
- 함수는 한 가지 책임만 져야 한다.
- 처음부터 완벽한 설계를 하기보다는 점진적으로 바뀌나갈 수 있는 구조를 지향하라.

어떻게 만들 것인가?

- 함수는 한 화면에 들어올 수 있도록 **간결**하게 유지한다.(평균 20~30줄, 최대 50줄)
- 매개변수는 가능한 **최소화**하라.(3개 이하 권장)
 - 넘겨야 할 데이터가 많다면 DTO(Data Transfer Object) 혹은 기능 분리를 고려할 수 있다.[1]
- 사전 조건과 사후 조건[2]을 검사하기 위해 Assert[3]를 사용한다.
 - 단, 예상된 실패와는 구분해야 한다.
- 함수의 이름은 동사 + 명사 형태로 짓는다.
 - 반환 값이 있는 경우 그 반환 값을 묘사한다.
 - 단, 반환 타입이 bool인 경우 질문 형태로 적는다.

[1]: 혹은 클래스로 만들어야 하는 것은 아닌지도 고려해볼 수 있다.

[2]: **사전 조건**(Precondition)은 함수 실행 전 반드시 참이어야 하는 조건을 의미하며, **사후 조건**(Postcondition)은 함수 실행 후 반드시 참이어야 하는 조건을 의미한다.

[3]: Assert는 디버그 모드에서만 동작하는 코드로, 개발 단계에서 버그를 잡아내기 위해 사용한다. Assert를 사용할 때, 함수 호출이나 로직 코드를 넣는 것을 주의해야 한다.

어떻게 만들 것인가?

- 항상 **입력 데이터를 검증**한 후 본래 기능을 위한 코드를 작성한다.
 - 입력 데이터가 잘못되었다면 즉시 함수를 끝낸다.[1]
 - 널 안전성(Null Safety), 데이터 타입이 표현할 수 있는 모든 값, 범위의 시작과 끝, 인덱스 오류, 무한 루프 등등을 고려하라.
- 함수가 실패했을 경우 **호출자에게 이를 알려줘야 한다**.
 - 오류 코드를 열거형으로 정의하거나 **결과 패턴**(Result Pattern)을 사용한다.[2]
 - 단, 애플리케이션 로직인 경우 오류가 발생한 부분에서 로그를 남기거나, 프로그램을 복구하거나, 종료해야 한다.
- **함수형 프로그래밍 패러다임**을 적용한다.

[1]: Guard Clause, Early Return, Fail Fast를 참고하라.

[2]: 현업에서는 아직 열거형을 반환하는 경우가 많으나, 최근 프로그래밍에서는 결과 패턴을 이용하는 것이 권장된다.

어떤 주석이 필요한가?

- 사용자의 관점에서 작성하며, 주석만 보고도 함수 사용에 필요한 모든 정보를 얻을 수 있도록 한다.
- C# XML 주석 형식을 따르며, 일관된 문체를 사용하는 것이 중요하다.[1]

XML 태그	지침
<summary>	함수의 목적 및 기능을 간결하게 설명
<param>	- 매개변수의 역할, 허용 범위, 단위 명시 - `null` 허용 여부, 유효한 값의 범위, 기본값 등을 빼먹지 않도록 주의
<returns>	- 반환 값의 의미를 상세하게 명시 - 특정 값이 반환되는 조건을 빼먹지 않도록 주의
<exception>	어떤 조건에서 예외가 발생하는지 명시
<remarks>	함수의 부수 효과와 복잡한 사전 조건 및 사후 조건 명시
<example>	함수의 사용을 코드 스니펫으로 제공

함수형 프로그래밍

함수형 프로그래밍

- 함수의 사용과 조합으로 선언적(Declarative)으로 코드를 작성하는 프로그래밍 패러다임을 함수형 프로그래밍(Functional Programming)이라 한다.
- 4가지 특징이 있다.
 - 순수 함수[1]를 지향하여 예측 가능하고 테스트하기 쉬운 코드를 작성한다.
 - 데이터는 불변성을 가지며, 데이터의 수정이 필요할 때마다 새로운 데이터를 생성한다.
 - 함수를 다른 함수의 인수나 반환 값으로 전달할 수 있다.[2]
 - 고차 함수로 제어 구조를 추상화한다.
- C#에서는 람다 식, 대리자, LINQ, 불변 컬렉션, 확장 메소드, 튜플, 패턴 매칭 등으로 이를 지원하고 있다.

[1]: 순수 함수(Pure Function)는 동일한 입력에 대해 동일한 출력을 보장하고, 부수 효과가 없는 함수를 말한다.

[2]: 이를 **1급 함수**(First-Class Functions)라 한다.

[3]: 고차 함수(Higher-Order Functions)는 함수를 인자로 받거나 함수를 반환 값으로 넘겨주는 함수다.

예시

- 함수형 패러다임을 적용하면 코드가 명령형(Imperative)이 아닌 선언형이 된다.

```
public int[] GetSquaredOddNumberArray(int[] numbers)
{
    int[] result = new int[numbers.Length];

    int resultIdx = 0;
    for (int i = 0; i < numbers.Length; ++i)
    {
        if (numbers[i] % 2 != 0)
        {
            result[resultIdx++] = numbers[i] * numbers[i];
        }
    }

    return result;
}
```

코드는 어떤 절차를 표현하는 명령형의 형태다.

예시

- 함수형 패러다임을 적용하면 코드가 명령형(Imperative)이 아닌 선언형이 된다.

```
using System.Linq;

public int[] GetSquaredOddNumberArray(int[] numbers)
{
    return numbers
        .Where(number => number % 2 != 0) // 홀수만 필터링
        .Select(number => number * number) // 각 요소를 제곱
        .ToArray();                       // 최종 결과인 배열(int[])로 변환
}
```

코드가 무엇을 하는지 나타내는 선언형의 형태다.

부록

더 나아가기

- 모듈화에 대해서 설명해 보세요.
- params로 가변 인수를 지정하는 것과 함수를 오버로딩하는 것은 무엇이 다를까요?
- Pass(Call) By Value와 Pass(Call) By Reference의 차이점은 무엇인가요?
- 정적 변수, 자동 변수, 동적 변수의 메모리 관리 방식에는 어떤 차이점이 있나요?
- 스택 프레임이란 무엇인가요?
- 함수는 호출 규약에 따른 오버헤드가 존재합니다. 이에 대해 서술해 보세요.
- 호출 규약에 의한 오버헤드를 제거하기 위하여 인라인(Inline) 기법을 사용하기도 합니다. 인라인에 대해서 정리해 보세요.
- 위와 같은 오버헤드 외에도 명령어 파이프라인 지연에 따른 오버헤드도 존재합니다. 명령어 파이프라이닝(Instruction Pipelining)에 대해서 조사해 보고, 그 오버헤드에 대해 설명해 보세요.

더 나아가기

- DRY 원칙, SRP, YAGNI 원칙에 대해 설명해 보세요.
- 결과 패턴에 대해 정리해 보세요.
- 함수형 프로그래밍이란 무엇인지 정의하고, 어떤 장점이 있는지 서술해 보세요.
- 오류 처리를 함수형으로 구현한 철도 지향 프로그래밍(ROP; Railway-Oriented Programming)에 대해서 정리해 보세요.

참고자료

- [소프트웨어 설계를 위한 추상적, 구조적 사고 | 인프콘2023](#)
- [코드를 성급하게 DRY하지 마세요 | GeekNews](#)
- [2. 변수명과 함수명 짓기 - 어? 쓰흠... 하아....](#)
- [코딩 책 한 권만 읽으면 일어나는 일](#)
- [1. 좋은 함수 만들기 - 부작용과 거리두기](#)
- [2. 좋은 함수 만들기 - 암묵적 입력/출력](#)
- [3. 좋은 함수 만들기 - Null 을 다루는 방법](#)
- [버그가 생기는 이유와 기초 예방법](#)
- [GitHub - karanraj-tech/result-pattern: How to implement the Result Pattern in .NET and C#.](#)

참고자료

- <https://www.inflearn.com/courses/lecture?courseId=334645>
- [Assert 어디에 넣을지 모르면 넌 주니어](#)
- [아름다운 코드에 대하여 | kciter.so](#)
- [어설트\(Assert\) 사용 시 저지르는 치명적인 실수! 누구나 한 번은 한다는데...](#)