→ 전자책

C# 스타일 가이드로 깔끔하고 스케일링 가능한 게임 코드 작성하기

Unity®

Contents

들어가는 말

클린 코드에는 항상 작성자의 세심한 노력이 엿보입니다.

– 마이클 페더스, 레거시 코드 활용 전략 저자

대부분의 게임 개발자는 읽기 쉽고 효율적으로 관리 및 재사용할 수 있는 클린 코드가 필요하다는 데 의견을 같이할 것입니다. 로직을 구현했다면 리팩터링과 정리 프로세스가 시작됩니다.

이렇게 의견이 일치하는 데는 타당한 이유가 있습니다. 작성자 본인에게 명백해 보이는 내용이라도 다른 개발자에게는 불분명하게 보일 수 있기 때문입니다. 마찬가지로 지금 어떤 로직을 구현해도 세 달 후에는 이 코드 스니핏의 역할이 무엇인지 기억하지 못할 수 있습니다.

이미 각자 선호하는 코드 작성 방식이 있을 수 있지만, 미래의 팀원들은 다른 방식을 선호할 가능성이 큽니다. 코드를 작성하는 방식에는 정답이 없지만, 이 가이드에서는 스타일 가이드를 바탕으로 공통된 기준을 마련하는 데 도움이 되는 팁을 안내합니다.

이 가이드에는 기존의 업계 표준 코드 스타일 가이드를 사용하거나 수정하는 방법에 관한 업계 전문가의 조언이 담겨 있습니다. 모든 팀원이 따를 수 있는 가이드를 마련하면 코드베이스를 기반으로 프로젝트를 상업적 규모로 확장할 수 있습니다.

이러한 유용한 팁은 초기에 추가적인 노력이 필요하더라도 장기적으로는 개발 프로세스에 도움이 됩니다. 보다 깔끔하고 스케일링 가능한 코드베이스를 사용하면 팀을 확충하는 과정에서 신규 개발자를 효율적으로 온보딩할 수도 있습니다.

코드를 항상 깔끔하게 유지하여 본인을 비롯한 모든 프로젝트 참여 인원이 더 쉽게 작업하는 환경을 만드세요.

클린 코드의 목표는 개발할 때 스케일링 가능성을 높이고 다음과 같은 제작 기준을 준수하도록 하는 것입니다.

- 일관된 명명 규칙 준수
- 가독성을 위해 코드 포맷 지정
- 클래스와 메서드를 작은 규모로 읽기 쉽게 구성
- 이해하기 어려운 모든 코드에 주석 추가

모바일용 퍼즐 게임과 콘솔용 대규모 MMORPG 중 무엇을 제작하든 코드베이스를 깔끔하게 만들면 소프트웨어 유지 관리에 드는 총비용이 줄어듭니다. 이렇게 하면 새 기능 구현이나 기존 소프트웨어 패치가 더 쉬워집니다.

이 작업은 미래의 팀원과 자신에게 도움이 될 것입니다.

참고: 본 가이드는 이 전자책의 두 번째 에디션으로, Unity 6에 맞게 업데이트되었습니다.

도움을 주신 분들

본 가이드는 인디 게임 개발자이자 교육자인 윌머 린이 작성했습니다. 테크니컬 콘텐츠 마케팅 매니저인 토마스 크로그 야콥센과 시니어 Unity 엔지니어인 피터 안드레아센, 스콧 빌라스, 미하이 포페스쿠도 큰 도움을 주었습니다.

팀 차원의 개발

> 컴퓨터가 이해하는 코드는 누구라도 작성할 수 있습니다.
> 뛰어난 프로그래머는 사람이 이해하는 코드를 작성합니다.
>
> – 마틴 파울러, 리팩터링 저자

어떤 개발자도 혼자서 다 해낼 수는 없습니다. 게임 애플리케이션의 기술적 요구 사항이 늘어날수록 도움이 필요할 것이며 결국 다양한 기술을 가진 팀원을 충원할 수밖에 없습니다. 클린 코드는 계속 확장되는 팀에 코딩 표준을 도입하여 팀원 모두가 동일한 정보를 공유할 수 있게 합니다. 이제 모두가 보다 통일된 가이드라인에 따라 동일한 프로젝트에서 작업할 수 있습니다.

코드 스타일 가이드에 대해 자세히 살펴보기 전에, Unity 개발을 확장하는 데 도움이 되는 몇 가지 일반적인 규칙을 다시 한번 살펴보겠습니다.

KISS(Keep it simple, stupid)

엔지니어와 개발자가 일을 복잡하게 만들 수 있다는 점은 솔직히 인정해야 합니다. 컴퓨팅과 프로그래밍이 그만큼 어렵긴 해도 말이죠. KISS 원칙(Keep it simple, stupid)을 활용하여 당면한 문제를 해결하는 가장 간단한 방법을 찾으세요.

검증된 간단한 기법으로 문제가 해결된다면 시간을 낭비할 필요가 없습니다. Unity 스크립팅 API에는 이미 다양한 솔루션이 포함되어 있습니다. 예를 들어 기존 육각형 타일맵이 지금 개발 중인 전략 게임에 적합하거나 UnityEngine.Pool로 오브젝트 풀링 시스템에 필요한 사항을 충족할 수 있다면 코드를 직접 작성할 필요가 없습니다. 아무 코드도 작성하지 않는 것이 가장 좋은 코딩입니다.

KISS 원칙

디자인의 단순성을 강조하는 KISS 원칙은 다음과 같은 명언에서 알 수 있듯 여러 시대에서 널리 사용된 유명한 개념입니다.

"단순함이야말로 정교함의 궁극적인 형태이다."
– 레오나르도 다빈치

"단순한 일은 단순하게 처리하라!"
– 비야네 스트롭스트룹

"단순성은 신뢰성의 전제 조건이다."
– 에츠허르 W. 데이크스트라

"모든 것은 가능한 한 단순해야 한다. 더 이상 단순해질 수 없을 때까지."
– 알버트 아인슈타인

프로그래밍 측면에서는 코드를 최대한 단순하게 작성해야 한다는 의미로 받아들일 수 있습니다. 불필요하게 복잡도를 높이지 마세요.

YAGNI 원칙

관련된 YAGNI 원칙(You aren't gonna need it)에 따르면 현재 필요한 기능만 구현해야 합니다. 언젠가 필요할 수도 있는 기능에 대해 미리 걱정하지 마세요. 지금 필요한 가장 단순한 기능만 만들고, 문제없이 작동하도록 구현하세요.

문제를 덮는 코딩 피하기

소프트웨어 개발의 첫 단계는 해결하려는 문제가 무엇인지 파악하는 것입니다. 의외로 실제 문제를 파악하지 못하고 코드 구현의 수렁에 빠지거나 이유를 모두 알지 못한 채 코드가 작동할 때까지 수정을 거듭하는 개발자가 상당히 많습니다.

예를 들어 메서드 상단에 if-null 문을 사용하여 신속하게 null 레퍼런스 예외를 수정했다고 가정해 보겠습니다. 그것이 진정한 원인이었을까요? 아니면 더 깊숙한 곳에 있는 다른 메서드 호출이 문제였을까요?

문제를 수정하려면 코드를 추가할 것이 아니라 근본적인 원인을 조사해야 합니다. 임시방편을 적용하기보다 문제가 발생하는 이유를 자문해 보세요.

매일 점진적으로 개선

클린 코드를 작성하는 과정은 유동적이고 반복적인 프로세스입니다. 팀원 모두 이러한 사고방식을 가져야 하며, 개발자로서 코드 클린업을 일상적인 작업으로 받아들여야 합니다. 고장 난 코드를 작성하려는 사람은 없습니다. 코드는 시간에 따라 고장이 생기기 마련입니다. 코드베이스는 지속적인 유지 관리가 필요합니다. 확실한 유지 관리 일정을 세우고 실천으로 옮기세요.

더 낫게 하되 완벽을 추구하지는 말 것

그렇다고 완벽함을 추구하진 마시기 바랍니다. 코드가 제작 기준을 충족하면 커밋하고 다음 단계로 넘어가야 합니다.

근본적으로 코드는 어떤 역할을 수행해야 합니다. 새로운 기능 구현과 코드 클린업 사이에서 균형을 유지하세요. 리팩터링 자체를 위한 리팩터링이 아니라 본인이나 다른 누군가에게 이득이 된다고 생각될 때 리팩터링하세요.

계획과 조정

실용주의 프로그래머에서 앤디 헌트와 데이브 토마스는 “프로그래밍은 건설보다는 정원 가꾸기와 비슷하다”고 합니다. 소프트웨어 엔지니어링은 유기적인 프로세스입니다. 계획대로 되지 않는 상황에 대비하고 적응하세요.

도면을 아무리 정교하게 작성한다 해도 종이 위에 정원을 설계하는 것만으로는 결과를 보장할 수 없습니다. 예상과 다른 시점에 꽃이 필 수 있습니다. 정원을 제대로 가꾸려면 꽃과 나무처럼 코드를 가지 치고, 옮겨 심고, 교체해야 합니다.

소프트웨어 디자인은 건축가의 청사진과는 다른데, 그보다는 더 가변적이면서 덜 기계적이기 때문입니다. 코드베이스가 커질수록 그에 맞게 대응해야 합니다.

일관성 유지

문제를 어떻게 해결할지 결정했다면 유사한 부분에 동일한 방식으로 접근하세요. 어렵지는 않지만 지속적인 노력이 필요한 일입니다. 클래스, 메서드, 대소문자 등의 이름 지정부터 프로젝트 폴더 및 리소스 구성에 이르는 모든 것에 이 원칙을 적용하세요.

무엇보다도 팀이 스타일 가이드를 받아들이고 따르도록 해야 합니다.

협동의 필요성

‘엉클 밥’이라는 애칭으로 불리는 미국의 소프트웨어 엔지니어이자 작가인 로버트 C. 마틴의 말처럼, 누구나 코드를 깔끔하고 단순하게 정리하고 싶겠지만 ‘깔끔하고 단순하게’ 작성하는 것과 ‘쉽게’ 작성하는 것은 다릅니다. 깔끔하고 단순한 정리는 초보자와 숙련된 개발자 모두에게 어렵고 노력이 필요한 일입니다.

그냥 방치하면 프로젝트가 지저분해질 것입니다. 많은 인원이 프로젝트의 다양한 부분을 진행하면서 생기는 자연스러운 결과입니다. 코드가 지저분해지는 것을 방지하기 위해 모든 팀원이 노력하며 스타일 가이드를 읽고 따라야 합니다.

스타일 가이드 따르기

캐시 무효화와 이름 지정 작업만 제외하면
컴퓨터 공학은 매우 간단한 학문이다.

– 필 칼튼, 소프트웨어 엔지니어

팀과 함께 사용하는 스타일 가이드

이 가이드는 Unity에서 개발하며 접하게 되는 가장 일반적인 코딩 규칙을 중점적으로 다룹니다. 여기에서는 대부분 Microsoft 프레임워크 디자인 지침에 속한 규칙을 다루고 있습니다. 해당 지침에는 더 많은 규칙들이 포함되어 있습니다.

이 가이드라인은 권장 사항이지 반드시 따라야 하는 규칙이 아닙니다. 팀이 선호하는 내용에 따라 커스터마이즈하세요. 모든 사람에게 적합한 스타일을 선택하고 모든 사람이 이를 적용해야 합니다.

가장 중요한 요소는 일관성입니다. 이러한 권장 사항을 따르면 나중에 스타일 가이드를 수정해야 할 때도 몇 가지만 찾아서 변경하면 빠르게 코드베이스를 마이그레이션할 수 있습니다.

자체 스타일 가이드가 이 문서나 Microsoft 프레임워크 디자인 지침과 상충한다면, 팀이 프로젝트를 진행하며 일관된 스타일을 유지할 수 있도록 자체 스타일 가이드를 두 문서보다 우선적으로 적용해야 합니다.

애플리케이션은 서로 생각이 다른 사람들이 공동으로 만든 결과물입니다. 스타일 가이드는 이러한 차이를 제어하여 최종 결과물을 일관성 있게 만드는 데 도움이 됩니다. 아무리 많은 사람이 참여한 Unity 프로젝트라 할지라도 마치 한 명이 개발한 것처럼 보여야 합니다.

스타일 가이드를 사용하면 코딩 규칙과 포맷 지정에서 불확실한 요소가 제거됩니다. 따라서 지시에 따르기만 해도 일관된 스타일을 유지할 수 있습니다.

1인 개발자라면 처음엔 다소 구속처럼 느껴질 수 있겠지만 팀으로 일하려면 반드시 스타일 가이드를 따라야 합니다.

스타일 가이드를 나중에 배당금을 지급받는 초기 투자라고 생각하세요. 하나의 기준을 유지하면 작업자를 다른 프로젝트로 이전할 때 재학습에 소요되는 시간을 줄일 수 있습니다.

Microsoft와 Google에서 제공하는 다음과 같은 포괄적 예제 가이드는 모두 업계 표준으로 간주됩니다.

— Microsoft C# 코딩 규칙

— Google C# 스타일 가이드

각 가이드는 Unity 개발을 효과적으로 관리하는 데 도움이 되는 출발점으로, 이름 지정, 코드 포맷, 주석 처리에 대한 솔루션도 제시합니다.

유니티는 초보자 튜토리얼, 기술 자료, 샘플 프로젝트에 일관되게 동일한 코드 스타일을 적용하도록 강요하지 않습니다. 대규모 팀이나 전문적인 활용처의 경우에는 엄격한 가이드라인을 적용해야 할 필요가 있지만, 초보자용 콘텐츠에는 엄격한 스타일 가이드를 적용할 경우 진입 장벽이 높아질 수 있기 때문입니다. Unity와 첫 프로그래밍 언어를 가볍게 배우려는 사람과 2,000명 이상의 인원으로 구성된 엔지니어링 팀은 애초에 서로 다른 요구 사항을 가집니다.

따라서 이 가이드에서는 본 문서에서 다루지 않는 다양한 규칙이 정의된 Microsoft 프레임워크 디자인 지침의 연장선에 있는 코드 스타일을 따릅니다. 엄격함과 실용성 사이에서 균형을 잘 맞춘 스타일로, 유니티의 엔지니어링 및 기술 자료 팀이 사용하는 스타일과 유사하며, 대부분의 Unity 프로젝트에 적합하다고 간주되는 코드 스타일입니다.

핵심은 결국 자신과 팀에게 가장 적합한 스타일을 선택하고 프로젝트 전반에서 일관성을 유지하는 것입니다.

유니티에서 제작한 C# 스타일 시트 예시를 참고하여 가이드를 직접 만들어 보세요. 원하는 대로 복사하고 필요에 따라 조정해 사용할 수 있습니다.

그 방법을 자세히 살펴보겠습니다.

명명 규칙

무언가에 이름을 부여하는 행위에는 깊은 심리학적 요소가 연결되어 있습니다. 이름은 그 이름을 가진 존재가 세상과 어떻게 부합되는지 보여 줍니다. 그것이 무엇이고, 그 사람이 누구이며, 우리에게 어떤 도움을 주는지 말입니다.

변수와 클래스, 메서드의 이름은 단순한 꼬리표가 아니며 그 무게와 의미를 지닙니다. 바람직한 명명 스타일을 사용하면 프로그램을 읽는 사람이 작성자의 아이디어를 더 잘 이해하도록 만들 수 있습니다.

다음은 이름을 지정할 때 고려해야 할 몇 가지 가이드라인입니다.

식별자 이름

식별자는 형식(예: 클래스, 인터페이스, 구조체, 델리게이트, 열거형), 멤버, 변수, 네임스페이스에 할당하는 이름입니다.

백슬래시, 기호, 유니코드 문자 같은 특수 문자는 C#에서 허용되더라도 식별자에 사용하지 마세요. 특정 Unity 커맨드 라인 툴의 작동을 저해할 수 있습니다. 대부분의 플랫폼과 호환되도록 하려면 특수 문자를 가급적 사용하지 않아야 합니다.

대소문자 사용

C#에서는 공백 문자를 사용해 식별자를 구분하므로 이름에 공백을 사용해 변수를 정의할 수 없습니다. 소스 코드에서 복합적인 이름이나 구문을 사용하는 문제를 대소문자 체계로 완화할 수 있습니다. 널리 쓰이는 명명 규칙 및 대소문자 규칙을 몇 가지 소개해 드립니다.

낙타 표기법(camelCase)

낙타 대문자로도 알려진 낙타 표기법은 공백이나 구두점 없이 구문을 작성하는 방식으로 단일 대문자로 단어를 구분합니다. 가장 첫 번째 문자는 소문자입니다. 로컬 변수와 메서드 파라미터는 낙타 표기법을 사용합니다. 예를 들면 다음과 같습니다.

examplePlayerController
maxHealthPoints
endOfFile

파스칼 표기법(PascalCase)

파스칼 표기법은 낙타 표기법의 배리에이션으로 맨 첫 번째 문자를 대문자로 표기합니다. Unity 개발에서는 클래스, public 필드와 메서드 이름에 이 표기법을 사용합니다. 예를 들면 다음과 같습니다.

ExamplePlayerController
MaxHealthPoints
EndOfFile

스네이크 표기법(snake_case)

이 표기법에서는 단어 사이의 공백이 밑줄 문자로 대체됩니다. 예를 들면 다음과 같습니다.

example_player_controller
max_health_points
end_of_file

케밥 표기법(kebab-case)

이 표기법에서는 단어 사이의 공백이 대시로 대체되며 단어가 대시 문자로 연결되어 표시됩니다. 예를 들면 다음과 같습니다.

example-player-controller
Max-health-points
end-of-file
naming-conventions-methodology

케밥 표기법은 웹 기술, 그중에서도 특히 CSS에서 널리 사용됩니다. 본 가이드의 뒷부분에서 다루는 UI 툴킷 USS에도 케밥 표기법 사용이 권장됩니다.

헝가리언 표기법

변수나 함수의 이름은 그 목적이나 형식을 나타내는 경우가 많습니다. 예를 들면 다음과 같습니다.

int iCounter
string strPlayerName

헝가리언 표기법은 오래된 규칙이며 Unity 개발에서는 잘 사용되지 않습니다.

필드 및 변수

변수와 필드에 다음 규칙을 고려해 보세요.

- **변수 이름에 명사를 사용합니다.** 변수 이름은 어떤 대상이나 상태를 나타내기 때문에 구체적이고 명확해야 하고 모호하면 안 됩니다. 따라서 변수가 부울 형식인 경우가 아니면 이름을 지정할 때 명사를 사용하세요(아래 참조).
- **부울의 접두사를 동사로 지정합니다.** 이러한 변수는 true 또는 false 값을 나타내며, 플레이어가 달리는 중인지 또는 게임이 종료되었는지 등의 질문에 답하는 경우가 많습니다. 동사를 접두사로 사용하면 그 의미가 더 분명해집니다. 주로 isDead, isWalking, hasDamageMultiplier 같은 설명이나 조건과 함께 사용되는 편입니다.
- **유의미한 이름을 사용합니다. 수학식이 아니면 약어를 사용하지 않습니다.** 변수의 이름은 변수의 의미를 전달합니다. 발음하고 검색하기 쉬운 이름을 선택하세요. 동료에게 도움이 될 뿐 아니라, AI 도구를 사용할 때 코드에 대한 맥락을 추가로 제공하여 코드 생성과 제안을 더 정확하게 이끌어낼 수 있습니다. 쉽게 읽을 수 있는 식별자 이름을 선택하세요. 예를 들어 HorizontalAlignment라는 프로퍼티 이름이 AlignmentHorizontal보다 가독성이 높습니다.

  한 글자 변수가 루프와 수식에는 괜찮을지 몰라도 그 외에는 약어를 사용하지 마세요. 몇 개의 모음을 생략해 시간을 절약하는 것보다 명확성이 더 중요합니다.

  프로토타이핑 단계에서는 짧고 의미 없는 이름을 쓰고 싶을 수 있지만, 그렇다고 해서 나중에 리팩터링 시간이 줄어들지는 않습니다. 처음부터 유의미한 이름을 선택하세요.

좋지 않은 예	권장 사용 예시	참고
`int d`	`int elapsedTimeInDays`	카운터나 표현식이 아니면 단일 문자 약어를 지양합니다. 측정 단위를 구체적으로 명시합니다.

`int hp,`
`string tName,`
`int mvmtSpeed`
	`int healthPoints,`
	`string teamName,`
	`int movementSpeed`
		변수 이름은 그 목적을 나타내므로 검색 가능하고 발음하기 쉬운 이름을 지정합니다.

`int getMovement-Speed`	`int movementSpeed`	명사를 사용하고, 부울이 아니라면 메서드에 사용할 수 있도록 동사를 아껴 둡니다(아래).
`bool dead`	`bool isDead,` `bool isPlayerDead`	부울은 true 또는 false로 답할 수 있는 질문을 합니다.

— **public 필드에는 파스칼 표기법(MyPropertyName)을 사용하고, private 변수에는 낙타 표기법(myPrivateVariable)을 사용합니다.** public 필드의 대안으로 프로퍼티를 public 게터와 함께 사용하세요(아래 '포맷 지정' 참고).

— **접두사나 특수 인코딩은 신중하게 사용합니다.** 어떤 가이드에서는 private 멤버 변수를 로컬 변수와 구분하기 위해 밑줄(_)을 접두사로 추가하라고 제안하기도 하지만, Unity 스타일 가이드에서는 이름만 보고 변수에 대한 더 많은 정보를 알 수 있도록 private 멤버 변수, 상수, 정적 변수에 각각 `m_`, `k_`, `s_` 라는 접두사를 사용합니다. 예를 들어 `movementSpeed` 대신 `m_movementSpeed`를 사용합니다.

파스칼 표기법에 접두사를 사용하는 것도 한 가지 옵션이기는 하지만(예: m_MovementSpeed), 최신 C#에서는 비교적 덜 사용되는 방식입니다.

아니면 this 키워드를 사용해 컨텍스트에서 멤버 변수와 로컬 변수를 구분하고 접두사를 생략할 수 있습니다. 보통 public 필드와 프로퍼티는 접두사가 없습니다. 로컬 변수와 파라미터는 접두사 없이 낙타 표기법을 사용합니다.

많은 개발자가 이러한 방식을 피하고 에디터에 의존합니다. 최신 IDE는 강조 표시, 색상 구분, 리치 컨텍스트를 지원합니다.

— **필드는 기본값으로 자동 초기화됩니다.** int와 같은 숫자형 형식의 기본값은 일반적으로 0이고, 참고 형식 필드(예: 오브젝트)는 기본적으로 null로 초기화되며, 부울 필드는 기본적으로 false로 초기화됩니다. 따라서 명시적으로 필드를 기본값으로 설정해야 하는 상황은 많지 않습니다.

— **상수 변수에 파스칼 표기법을 사용하고 접두사로 k_를 추가합니다.** 이렇게 하면 상수를 일반 변수나 프로퍼티와 구분하고 쉽게 읽고 관리할 수 있는 코드를 작성할 수 있습니다.

```
// 예시: 상수
public class MathConstants
{
    public const int k_MaxItems = 100;
}
```

— **액세스 수준 한정자를 일관되게 지정하거나 생략합니다.** 액세스 한정자를 제외하면 컴파일러가 액세스 수준을 `private`으로 추정합니다. 이렇게 해도 문제는 없으나 기본 액세스 한정자를 생략하는 방식에서 일관성을 유지해야 합니다.

MSFT 가이드라인에서는 액세스 수준을 명확하게 알리고 모호함을 없애기 위해 private을 명시적으로 지정하도록 권장합니다. 다른 가이드에서는 중복된 액세스 지정자를 없애고(즉, 'private'을 입력하지 않고) 중복된 이니셜라이저를 없애야 한다고(즉, int를 '= 0'으로 초기화하거나 참조 형식을 '= null'로 초기화하지 않아야 한다고) 주장합니다. 나중에 하위 클래스에서 사용하려 한다면 `protected`를 지정해야 합니다. 가독성을 해치지 않는다면 암묵적이고 중복되는 지정자(예: private)를 생략하여 단순하게 작성하는 것이 좋습니다.

— **가독성을 간결성보다 우선합니다.** MSFT 기술 자료의 예시에서 볼 수 있듯이, X축에 대한 모호한 참조인 ScrollableX보다 CanScrollHorizontally가 프로퍼티 이름으로 더 바람직합니다.

예시 코드 스니핏

이 가이드의 코드 스니핏은 작동하지 않고 축약되어 있으며, 스타일과 포맷 지정을 보여 주기 위해 수록되었습니다.

Microsoft의 프레임워크 디자인 지침의 수정된 버전인 Unity 개발자용 C# 스타일 시트 예시를 참조할 수도 있습니다. 이 시트는 팀의 스타일 가이드를 만드는 방법의 한 예시를 제시합니다.

스타일 가이드 예시의 각 규칙을 검토하고 팀이 선호하는 내용에 따라 커스터마이즈하세요. 개별 규칙의 세부 사항보다 중요한 점은 모든 팀원이 규칙을 일관되게 따른다는 동의를 얻는 것입니다. 확실하지 않은 경우에는 팀의 자체 가이드를 통해 스타일 불일치 요소를 정리하는 것이 좋습니다.

```
// 예시: public 변수와 private 변수가 함께 그룹화됩니다.
public float DamageMultiplier = 1.5f;
public float MaxHealth;
public bool IsInvincible;
private bool m_isDead;
```

```
private float m_currentHealth;

public void InflictDamage(float damage, bool isSpecialDamage)
{
    // 로컬 변수
    int totalDamage = damage;
    // 로컬 변수 대 public 멤버 변수
    if (isSpecialDamage)
    {
       totalDamage *= DamageMultiplier;
    }
    // 로컬 변수 대 private 멤버 변수
    if (totalDamage > _currentHealth)
    {
        /// ...
    }
}
```

- **한 줄에 하나의 변수만 선언합니다.** 간결함은 떨어질 수 있지만, 가독성은 향상됩니다.
- **중복되는 이름은 피합니다**. 클래스 이름이 `Player`라면 멤버 변수의 이름을 `PlayerScore` 또는 `PlayerTarget`으로 지정할 필요가 없습니다. `Score` 또는 `Target`으로 축약하세요.
- 중복된 이니셜라이저를 없앱니다(즉, int를 '= 0'으로 초기화하거나 참조 형식을 '= null'로 초기화하지 않습니다).
- **농담과 같은 표현이나 말장난은 지양합니다.** `infiniteMonkeys` 또는 `dudeWheresMyChar` 같은 변수가 지금은 웃음을 유발할지 몰라도, 여러 번 보고 나면 더 이상 재미있지 않을 것입니다. 게다가 이는 앞에서 설명한 맥락을 드러내는 이름을 지정한다는 원칙을 위반합니다.
- **모호성을 피하고 항상 가독성을 높일 방법을 모색한다면, 맥락을 보고 형식을 명확히 알 수 있는 경우 var을 사용해도 좋습니다.** 바람직한 명명 스타일을 사용한다면 변수 이름 자체가 이미 의도를 표현하므로 모호성이 큰 문제가 되지 않습니다. var를 사용하면 특정 형식을 추상화할 수 있어, 형식이 바뀌더라도 코드를 업데이트해야 하는 지점을 줄일 수 있으므로 리팩터링이 더 간단해집니다. foreach 루프에서 var는 반복에 사용되는 변수가 열거자에 의해 제공된 형식과 일치하도록 보장해 줍니다. 일치하지 않는 형식을 명시적으로 선언할 경우 이를 컴파일러가 허용하여 런타임 오류가 발생할 수 있습니다.

```
// 예시: var의 바람직한 사용
var powerUps = new List<PowerUps>();
var dictionary = new Dictionary<string, List<GameObject>>();

// 좋지 않은 예: 잠재적인 모호성
var powerUps = PowerUpManager.GetPowerUps();
```

열거형

열거형은 이름이 지정된 상수 세트로 정의된 특별한 값 형식입니다. 기본적으로 상수는 정수이며 0부터 셉니다.

열거형 이름과 값에는 파스칼 표기법을 사용합니다. 클래스 외부에 public 열거형을 배치하여 글로벌로 만들 수 있습니다. 열거형은 가능한 값 세트에 포함된 단일 값을 나타내므로 열거형 이름에는 단수 명사를 사용합니다. 접두사나 접미사는 사용하지 않아야 합니다.

> 참고: System.FlagsAttribute 속성으로 표시된 비트 열거형은 이 규칙의 예외에 해당되며, 둘 이상의 형식을 나타내므로 복수화하는 것이 일반적입니다.

```
// 예시: 단수형 명사를 사용하는 열거형
public enum WeaponType
{
    Knife,
    Gun,
    RocketLauncher,
    BFG
}

public enum FireMode
{
    None = 0,
    Single = 5,
    Burst = 7,
    Auto = 8,
}

// 예시: 하지만 비트 열거형이 복수형임(1 << 비트 값 스타일도 사용 가능)

[Flags]
public enum AttackModes
{
  // 십진수          // 이진수
   None = 0,                          // 000000
   Melee = 1,                         // 000001
   Ranged = 2,                        // 000010
   Special = 4,                       // 000100

   MeleeAndSpecial = Melee | Special  // 000101
}
```

클래스 및 인터페이스

클래스 및 인터페이스의 이름을 지정할 때는 다음 표준 규칙을 따릅니다.

- **클래스 이름에는 파스칼 표기법 명사 또는 명사구를 사용합니다.** 이렇게 하면 동사구로 지정되는 메서드 이름과 형식 이름이 구분됩니다.
- **파일에 MonoBehaviour가 있는 경우 소스 파일 이름이 일치해야 합니다.** 파일에 다른 내부 클래스를 둘 수도 있지만, MonoBehaviour는 각 파일에 하나만 있어야 합니다.
- **인터페이스 이름에 대문자 'I'를 접두사로 사용합니다.** 그 뒤에 기능을 설명하는 형용사를 붙입니다.

```
// 예시: 클래스 포맷 지정
public class ExampleClass : MonoBehaviour
{

    public int PublicField;
    public static int MyStaticField;
    private int m_packagePrivate;
    private int m_myPrivate;

    private static int m_myPrivate;

    protected int m_myProtected;

    public void DoSomething()
    {

    }
}

// 예시: 인터페이스
public interface IKillable
{
    void Kill();
}

public interface IDamageable<T>
{
    void Damage(T damageTaken);
}
```

메서드

C#에서 실행되는 모든 명령은 메서드의 컨텍스트에서 수행됩니다.

> **참고:** Unity 개발에서 '함수'와 '메서드'는 종종 구별 없이 같은 의미로 사용됩니다. 하지만 C#에서 클래스에 통합하지 않으면 함수를 작성할 수 없으므로 허용되는 용어는 '메서드'입니다.

메서드는 동작을 수행하므로 상황에 따라 다음 규칙을 적용해 이름을 지정합니다.

- **이름을 동사 또는 동사구로 시작합니다.** 필요한 경우 컨텍스트를 추가합니다(예: `GetDirection`, `FindTarget` 등).
- **파라미터에 낙타 표기법을 사용합니다.** 로컬 변수처럼 메서드에 전달되는 파라미터의 포맷을 지정합니다.
- **부울을 반환하는 메서드는 질문 형식으로 지정합니다.** 부울 변수와 마찬가지로 true-false 조건을 반환하는 메서드는 동사를 접두사로 사용합니다. 그 결과 `IsGameOver`, `HasStartedTurn` 같은 질문 형식의 이름이 지정됩니다.

```
// 예시: 동사로 시작하는 메서드
public void SetInitialPosition(float x, float y, float z)
{
    transform.position = new Vector3(x, y, z);
}

// 예시: 메서드가 부울을 반환하는 경우 질문 형식으로 지정
public bool IsNewPosition(Vector3 currentPosition)
{
    return (transform.position == newPosition);
}
```

이벤트 및 이벤트 핸들러

C#에서 이벤트는 관찰자(Observer) 패턴을 구현합니다. 이 소프트웨어 설계 패턴은 하나의 오브젝트인 주체(또는 퍼블리셔)가 관찰자(또는 구독자)라는 종속 오브젝트 목록에 통지할 수 있는 관계를 정의합니다. 따라서 주체는 관련된 오브젝트를 긴밀히 결합하지 않고도 상태 변경 사항을 관찰자에게 브로드캐스트할 수 있습니다. 디자인 패턴 및 SOLID 원칙으로 코딩 스킬 업그레이드 전자책에서 Unity 프로젝트에서 관찰자 패턴을 비롯한 다양한 패턴과 디자인 원칙을 사용하는 방법을 자세히 알아보세요.

주체 및 관찰자에는 이벤트 및 관련 메서드에 대한 여러 명명 체계가 존재합니다. 다음과 같은 방식을 활용해 보세요.

- **이벤트를 동사구로 명명합니다.** 상태 변경 사항을 정확하게 전달하는 이름을 사용하는 것이 좋습니다. 현재분사나 과거분사를 사용하여 '이전' 이벤트인지 '이후' 이벤트인지 명시하세요. 예를 들어 문을 열기 전의 이벤트를 'OpeningDoor'라고 하거나 문을 연 이후의 이벤트에 'DoorOpened' 등의 이름을 사용할 수 있습니다.

— **이벤트에 System.Action 델리게이트를 사용합니다.** 대부분의 경우 Action<T> 델리게이트는 게임플레이에 필요한 이벤트를 처리할 수 있습니다. 반환 형식이 void인 다양한 형식의 입력 파라미터를 0~16 중에서 전달할 수 있습니다. 사전 정의된 델리게이트를 사용하면 코드가 줄어듭니다.

참고: EventHandler 또는 EventHandler<TEventArgs> 델리게이트를 사용할 수도 있습니다. 모든 사람이 이벤트를 어떻게 구현할 것인지 팀 차원에서 합의합니다.

```
// 예시: 이벤트

// System.Action 델리게이트 사용

public event Action OpeningDoor;    // 이벤트 이전
public event Action DoorOpened;     // 이벤트 이후

public event Action<int> PointsScored;
public event Action<CustomEventArgs> ThingHappened;
```

— **주체의 이벤트 발생 메서드에 'On' 접두사를 사용합니다.** 이벤트를 발생시키는 주체는 보통 'On' 접두사가 있는 메서드(예: 'OnOpeningDoor' 또는 'OnDoorOpened')에서 이벤트를 호출합니다.

```
// 구독자가 있는 경우 이벤트 발생
public void OnDoorOpened()
{
    DoorOpened?.Invoke();
}

public void OnPointsScored(int points)
{
    PointsScored?.Invoke(points);
}
```

— **관찰자의 이벤트 처리 메서드에 주체의 이름과 밑줄(_)을 접두사로 사용합니다.** 주체의 이름이 'GameEvents'라면 관찰자는 'GameEvents_OpeningDoor' 또는 'GameEvents_DoorOpened'라는 메서드를 가질 수 있습니다.

여기에서 말하는 것은 '이벤트 처리 메서드'이며 EventHandler 델리게이트와 혼동해서는 안 됩니다.

— **필요한 경우에만 커스텀 EventArgs를 생성합니다.** 커스텀 데이터를 이벤트에 전달해야 하는 경우 System.EventArgs **또는** 커스텀 구조체**에서 상속된 새로운 형식의 EventArgs를 생성합니다.**

```
// 필요한 경우 EventArgs 정의

// 예시: ID 및 Color 전달에 사용하는 읽기 전용 커스텀 구조체
public struct CustomEventArgs
{
    public int ObjectID { get; }
    public Color Color { get; }

      public CustomEventArgs(int objectId, Color color)
      {
          this.ObjectID = objectId;
          this.Color = color;
      }
}
```

네임스페이스

네임스페이스를 사용하면 클래스, 인터페이스, 열거형 등이 다른 네임스페이스 또는 글로벌 네임스페이스의 기존 항목과 충돌하지 않게 할 수 있습니다. 네임스페이스는 에셋 스토어에서 가져온 타사 에셋과의 충돌도 방지합니다.

네임스페이스는 다음과 같이 적용합니다.

— 특수 기호나 밑줄 없이 파스칼 표기법을 사용합니다.

— 파일 상단에 using 지시문을 추가해 네임스페이스 접두사의 반복 입력을 방지합니다.

— 서브 네임스페이스도 생성합니다. dot(.) 연산자를 사용하여 이름 수준을 구분하면 계층적 카테고리로 스크립트를 구성할 수 있습니다. 예를 들어 MyApplication.GameFlow, MyApplication.AI, MyApplication.UI 등을 생성해 게임의 다양한 로직 컴포넌트를 보유할 수 있습니다.

— 어떤 개발자는 관련 클래스와 컴포넌트가 논리적으로 그룹화된 폴더 구조를 반영하는 네임스페이스를 선호하며, 이러한 경우 코드베이스의 구조를 찾고 파악하기도 쉽습니다.

```
namespace Enemy
{
    public class Controller1 : MonoBehaviour
    {
        ...
    }

    public class Controller2 : MonoBehaviour
    {
        ...
    }
}
```

코드에서 이러한 클래스는 각각 `Enemy.Controller1` 및 `Enemy.Controller2`로 불립니다. 다음과 같이 `using` 문을 추가하면 접두사를 입력하지 않아도 됩니다.

```
using Enemy;
```

컴파일러가 `Controller1` 및 `Controller2`라는 클래스명을 발견하면 `Enemy.Controller1` 및 `Enemy.Controller2`임을 인식합니다.

스크립트가 다른 네임스페이스의 동일한 이름을 가진 클래스를 참조해야 하는 경우 접두사를 사용해 구분합니다. 예를 들어 `Player` 네임스페이스에 `Controller1` 및 `Controller2` 클래스가 있는 경우 `Player.Controller1` 및 `Player.Controller2` 라고 상세히 명시하여 충돌을 피할 수 있습니다. 이렇게 하지 않으면 컴파일러가 오류를 보고합니다.

포맷 지정

코드를 쉽게 쓰고 싶다면 읽기 쉽게 만들어야 한다.

– 로버트 C. 마틴, 클린 코드 및 클린 소프트웨어 저자

포맷 지정에 신경을 덜 쓰면 다른 작업에 더 집중할 수 있습니다.

명명과 마찬가지로 포맷 지정은 추측을 줄이고 코드 명확성을 개선하는 데 도움이 됩니다. 표준화된 스타일 가이드를 따르면 코드의 형태보다 코드가 수행하는 역할에 더 집중해서 코드를 검토할 수 있습니다.

팀의 요구 사항에 맞게 이 예시 규칙을 생략하고 확장하고 수정하세요.

모든 경우에 팀이 각 포맷 지정 규칙을 어떻게 구현할지 고려한 다음 모든 사람이 일관되게 적용하도록 합니다. 불일치는 팀의 스타일을 참조하여 해결합니다.

Unity 개발 스타일 가이드를 만들 때는 다음과 같은 각각의 코드 포맷 지정 권장 사항을 고려하세요.

프로퍼티

프로퍼티는 클래스 값을 읽고, 쓰고, 계산하는 유연한 메커니즘을 제공합니다. 프로퍼티는 public 멤버 변수처럼 동작하나 실제로는 접근자(accessor)라는 특수 메서드입니다. 각 프로퍼티에는 private 필드에 액세스하기 위한 get 및 set 메서드가 있으며 이를 지원 필드(backing field)라고 합니다.

프로퍼티는 이런 식으로 데이터를 캡슐화하여 사용자나 외부 오브젝트로 인해 의도하지 않은 변경 사항이 적용되지 않도록 숨깁니다. getter 및 setter에는 각각 자체 액세스 한정자가 있어 프로퍼티를 읽기-쓰기, 읽기 전용, 쓰기 전용으로 설정할 수 있습니다.

접근자를 사용하여 데이터를 검증하거나 변환할 수도 있습니다(예: 데이터가 선호 포맷에 맞는지 검증하거나 값을 특정 단위로 변경).

프로퍼티의 구문은 달라질 수 있으므로 그 포맷을 지정하는 방법을 스타일 가이드에 정의해야 합니다. 다음 팁을 사용하여 코드에서 프로퍼티의 일관성을 유지하세요.

— **한 줄로 작성된 읽기 전용 프로퍼티**에는 (=>) 표현식 본문을 사용하는 프로퍼티를 사용합니다. 이는 private 지원 필드를 반환합니다.

```
// 예시: 표현식이 적용된 프로퍼티
public class PlayerHealth
{
    // private 지원 필드
    private int m_maxHealth;

    // 읽기 전용, 지원 필드 반환
    public int MaxHealth => m_maxHealth;

    // 다음과 동등함:
    // public int MaxHealth { get; private set; }
}
```

— **그 외 모든 항목은 이전의** `{ get; set; }` **구문**을 사용합니다. 지원 필드를 지정하지 않고 public 프로퍼티만 노출하려는 경우 자동으로 구현된 프로퍼티를 사용합니다.

`set` 및 `get` 접근자에 표현식이 적용된 구문을 적용합니다.

쓰기 액세스 권한을 주지 않으려면 `setter`를 `private`으로 만들어야 합니다. 다중 라인 코드 블록에서 닫는 중괄호를 여는 중괄호에 맞춥니다.

```
// 예시: 표현식이 적용된 프로퍼티
public class PlayerHealth
{
    // 지원 필드
    private int m_maxHealth;

    // 명시적으로 getter 및 setter 구현
    public int MaxHealth
    {
        get => m_maxHealth;
        set => m_maxHealth = value;
    }

    // 쓰기 전용(지원 필드 사용하지 않음)
    public int Health { private get; set; }

    // 쓰기 전용, 명시적인 setter 없음
    public SetMaxHealth(int newMaxValue) => _maxHealth = newMaxValue;
}
```

— 아래 예시와 같이 함수를 사용하여 private 데이터를 노출할 수도 있지만, 간단한 get/set 연산에는 프로퍼티를 사용하는 것이 권장됩니다. 복잡한 로직이나 계산이 필요한 연산에서는 메서드를 사용하는 것이 권장됩니다.

```
// 예시: 표현식이 적용된 프로퍼티
public class PlayerHealth
{
    // 지원 필드
    private int m_maxHealth;

    public int GetMaxHealth
    {
        return m_maxHealth;
    }
}
```

직렬화

스크립트 직렬화는 데이터 구조체나 오브젝트 상태를 Unity에서 저장하고 나중에 재구성할 수 있는 포맷으로 변환하는 자동 프로세스입니다. 성능상의 이유로 Unity는 다른 프로그래밍 환경과 다른 방식으로 직렬화를 처리합니다.

직렬화된 필드는 인스펙터(Inspector)에 표시되나 정적, 상수 또는 읽기 전용 필드는 직렬화할 수 없습니다. 반드시 `public`이거나 `[SerializeField]` 속성으로 태그가 지정되어야 합니다. Unity는 특정 필드 형식만 직렬화하므로 모든 직렬화 규칙에 대한 내용은 기술 자료 페이지를 참조하시기 바랍니다.

직렬화된 필드를 사용하는 경우 다음 몇 가지 기본 가이드라인을 따르세요.

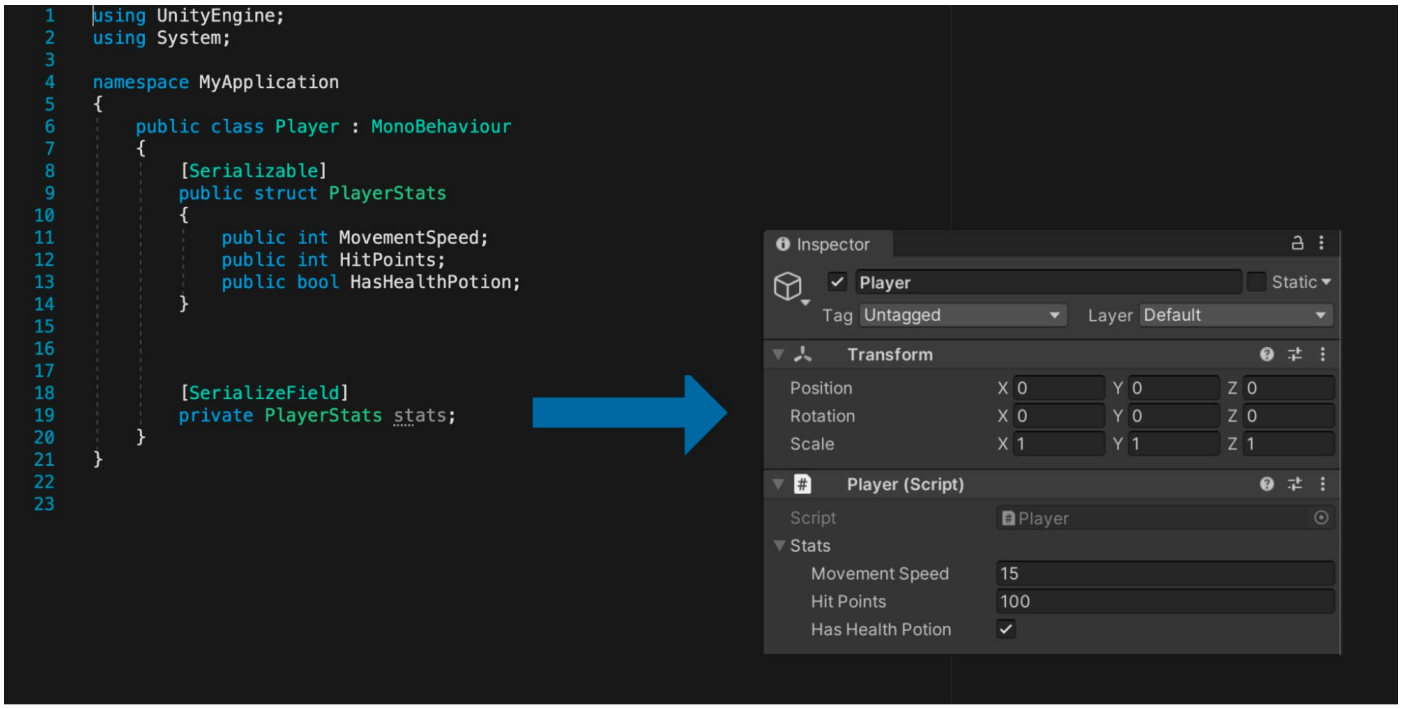
— `[SerializeField]` **속성을 사용합니다**. `SerializeField` 속성을 `private` 또는 `protected` 변수와 함께 사용하면 해당 변수를 인스펙터에 나타나게 할 수 있습니다. 이렇게 하면 변수를 `public`으로 만드는 것보다 데이터를 더 효과적으로 캡슐화할 수 있으며 외부 오브젝트가 변수 값을 덮어쓰는 것을 방지할 수 있습니다.

— **Range 속성을 사용하여 최솟값 및 최댓값을 설정합니다.** `[Range(min, max)]` 속성은 사용자가 숫자 필드에 할당할 수 있는 항목을 제한하려는 경우 유용하며, 인스펙터의 슬라이더로 필드를 편리하게 나타낼 수도 있습니다.

— **직렬화 가능한 클래스 또는 구조체에서 데이터를 그룹화하여 인스펙터를 정리합니다.** `public` 클래스 또는 구조체를 정의하고 `[Serializable]` 속성으로 표시합니다. 인스펙터에서 노출하려는 각 형식의 `public` 변수를 정의합니다.

```
// 예시: PlayerStats용으로 직렬화 가능한 클래스

using System;
using UnityEngine;

public class Player : MonoBehaviour
{
    [Serializable]
    public struct PlayerStats
    {
        public int MovementSpeed;
        public int HitPoints;
        public bool HasHealthPotion;
    }

// 예시: 인스펙터에서 private 필드 표시

    [SerializeField]
    private PlayerStats m_stats;
}
```

직렬화 가능한 이 클래스를 다른 클래스에서 참조합니다. 결과 변수가 인스펙터의 축소 가능한 단위 내에 나타납니다.



직렬화 가능한 클래스 또는 구조체를 사용하여 인스펙터를 구성할 수 있습니다.

중괄호 및 들여쓰기 스타일

C#에는 다음과 같은 두 가지 일반적인 들여쓰기 스타일이 있습니다.

— Allman 스타일은 여는 중괄호를 새 줄에 배치하는 방식으로, BSD Unix의 BSD 스타일이라고도 합니다.

— K&R 스타일 또는 'One True Brace 스타일'은 여는 중괄호를 이전 헤더와 같은 줄에 배치합니다.

```
// 예시: Allman 또는 BSD 스타일에서는 여는 중괄호가 새로운 줄에 놓임

void DisplayMouseCursor(bool showMouse)
{
     if (!showMouse)
     {
          Cursor.lockState = CursorLockMode.Locked;
          Cursor.visible = false;
     }
     else
     {
```

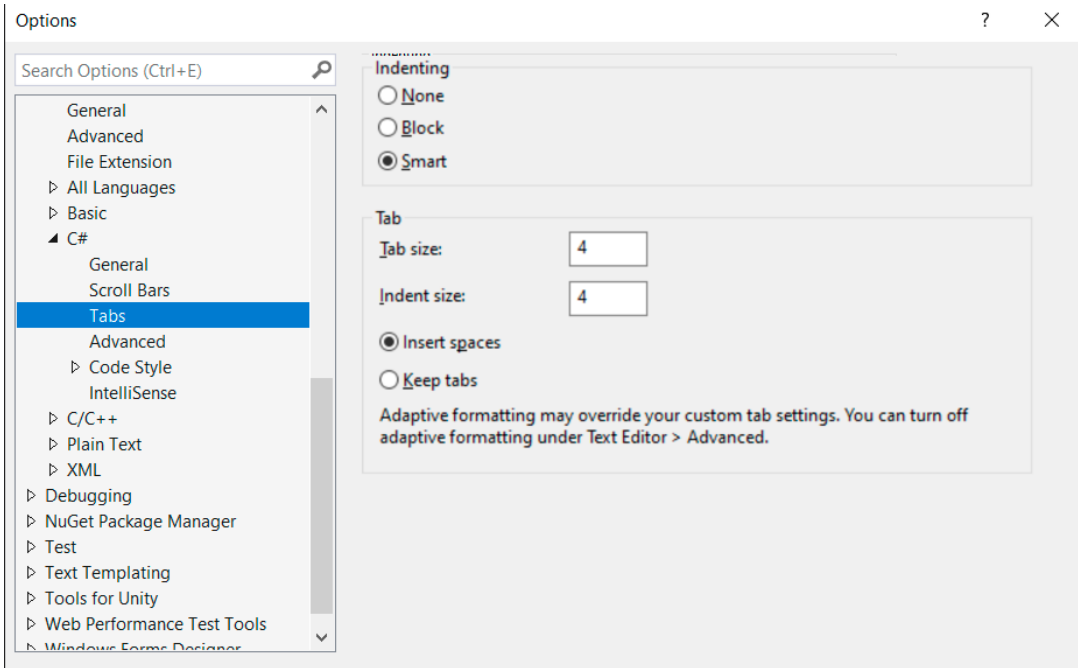
```
          Cursor.lockState = CursorLockMode.None;
          Cursor.visible = true;
     }
}

// 예시: K&R 스타일에서는 여는 중괄호가 이전 줄에 놓임

void DisplayMouseCursor(bool showMouse){
     if (!showMouse) {
          Cursor.lockState = CursorLockMode.Locked;
          Cursor.visible = false;
     }
     else {
          Cursor.lockState = CursorLockMode.None;
          Cursor.visible = true;
     }
}
```

이 들여쓰기 스타일에는 다양한 배리에이션이 있습니다. 이 가이드의 예시에서는 Microsoft 프레임워크 디자인 지침의 Allman 스타일을 사용합니다. 팀이 어떤 스타일을 선택하든 모든 구성원이 동일한 들여쓰기 및 중괄호 스타일을 따라야 합니다.

다음 팁을 사용해 보세요.

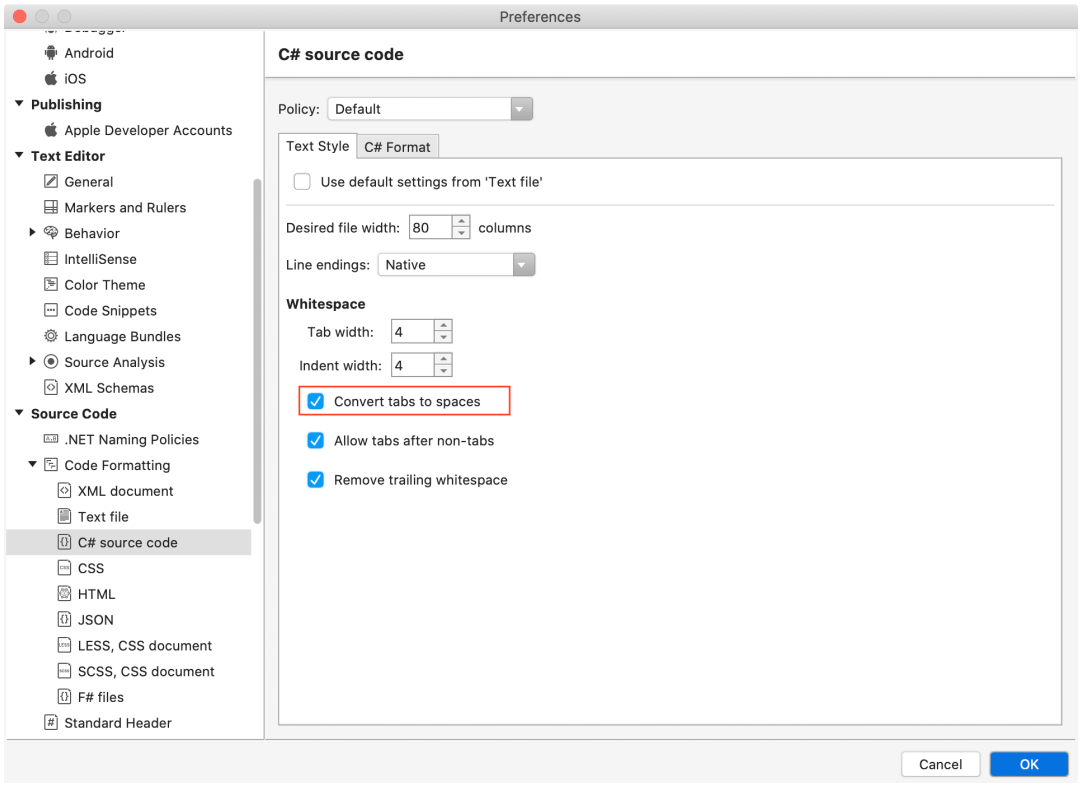
— **일관된 들여쓰기 방식을 정합니다.** 일반적으로 4개 또는 2개의 공백입니다. 팀원들과 에디터 설정을 통일하여 '탭 vs. 스페이스'를 둘러싼 과열된 논쟁을 피하세요. Visual Studio에서는 탭을 공백으로 변환하는 옵션을 제공합니다.

Visual Studio(Windows)의 경우 **Tools > Options > Text Editor > C# > Tabs**로 이동합니다.



Visual Studio의 Tabs 설정

Mac용 Visual Studio의 경우 **Preferences > Source Code > C# Source Code**로 이동합니다. Text Style을 선택하여 설정을 조정합니다.



일관된 들여쓰기를 위해 탭을 공백으로 전환

— **가능하다면 단일 줄 구문에서도 중괄호를 생략하지 않습니다.** 이렇게 하면 일관성이 향상되어 코드를 더 쉽게 읽고 관리할 수 있습니다. 다음 예시에서 중괄호는 DoSomething 동작을 루프로부터 명확하게 분리합니다.

나중에 디버그 줄을 추가하거나 DoSomethingElse를 실행해야 하는 경우, 중괄호는 이미 제자리에 있을 것입니다. 절을 항상 별도의 줄에 배치하면 중단점을 쉽게 추가할 수 있다고 말하는 프로그래머도 있습니다.

```
// 예시: 명확성을 고려해 중괄호를 유지하세요...

for (int i = 0; i < 100; i++) { DoSomething(i); }

// … 또는 절을 별도의 줄에 두세요.
for (int i = 0; i < 100; i++)
{
    DoSomething(i);
}

// 좋지 않은 예: 중괄호 생략
for (int i = 0; i < 100; i++) DoSomething(i);
```

— **중첩된 다중 줄 구문에서는 중괄호를 제거하지 않습니다.** 이 경우에는 중괄호를 없애도 오류는 발생하지 않지만, 혼란을 줄 수는 있습니다. 필수는 아니더라도, 가독성 향상을 위해 중괄호를 적용하세요. 중괄호를 사용하면 감싸는 구조체를 리팩터링하지 않고도 새 로직 추가와 같은 수정 사항을 안전하게 적용할 수 있다는 이점도 있습니다.

```
// 예시: 명확성을 고려해 중괄호 유지
for (int i = 0; i < 10; i++)
{
       for (int j = 0; j < 10; j++)
     {
             ExampleAction();
     }
}
// 좋지 않은 예: 중첩된 다중 줄 구문에서 중괄호 제거
for (int i = 0; i < 10; i++)
       for (int j = 0; j < 10; j++)
             ExampleAction();
```

— `switch` **문을 표준화합니다.** 길게 이어지는 if-else 문은 switch 문으로 대체하면 보통 가독성이 높아집니다. 다음은 case 문에 들여쓰기를 적용한 예시입니다. 기본 case를 포함하는 것이 권장됩니다. 기본 case가 필요하지 않더라도(예: 가능한 모든 case를 포함한 경우) 기본 case를 포함하면 코드가 예기치 않은 값을 처리할 수 있습니다.

```
// 예시: switch 문에서 case 들여쓰기
switch (someExpression)
{
   case 0:
      DoSomething();
      break;
   case 1:
      DoSomethingElse();
      break;
   case 2:
      int n = 1;
      DoAnotherThing(n);
      break;
    default:
      // 예기치 않은 case 또는 기본 case 처리
      break;
}
```

EditorConfig란?

여러 개발자가 서로 다른 에디터와 IDE를 사용해 동일한 프로젝트를 진행한다면 **EditorConfig** 파일을 사용해 보세요.

EditorConfig 파일을 사용하면 팀 전체에 적용되는 코딩 스타일을 정의할 수 있습니다. Visual Studio와 Rider 같은 대다수 IDE는 기본 지원이 함께 제공되며 별도의 플러그인이 필요하지 않습니다.

EditorConfig 파일은 가독성이 우수하며 버전 관리 시스템과 연동됩니다. 예시 파일을 확인해 보세요. EditorConfig의 코딩 스타일은 코드와 함께하며 Visual Studio 외부에서도 코딩 스타일을 적용할 수 있습니다.

EditorConfig 설정은 글로벌 Visual Studio 텍스트 에디터 설정보다 우선합니다. 개별 에디터 환경 설정은 **.editorconfig** 파일 없이 코드베이스로 작업할 때마다 또는 **.editorconfig** 파일이 특정 설정을 오버라이드하지 않을 때 계속 적용됩니다.

실제 샘플은 GitHub 저장소를 확인하세요.

가로 공백

공백 등의 간단한 요소로도 화면에서 코드의 가독성을 높일 수 있습니다. 포맷 지정 환경 설정은 개인별로 다를 수 있지만 다음 제안 사항에 따라 가독성을 개선해 보세요.

— **공백을 추가해 코드 밀도를 낮춥니다.** 공백을 추가하면 줄의 각 부분을 시각적으로 분리하는 느낌을 주어 가독성을 높일 수 있습니다.

```
// 예시: 공백을 추가해 줄 가독성 높이기
for (int i = 0; i < 100; i++) { DoSomething(i); }

// 좋지 않은 예: 공백 없이 표시
for(inti=0;i<100;i++){DoSomething(i);}
```

— **함수 인수 사이의 쉼표 다음에 하나의 공백을 사용합니다.**

```
// 예시: 인수 사이의 쉼표 뒤에 공백 하나 추가
CollectItem(myObject, 0, 1);

// 좋지 않은 예: 공백 없이 표시
CollectItem(myObject,0,1);
```

— **괄호와 함수 인수 뒤에 공백을 추가하지 않습니다.**

```
// 예시: 괄호와 함수 인수 뒤 공백 제거
DropPowerUp(myPrefab, 0, 1);

// 좋지 않은 예:
DropPowerUp( myPrefab, 0, 1 );
```

— **함수 이름과 괄호 사이에 공백을 사용하지 않습니다.**

```
// 예시: 함수 이름과 괄호 사이의 공백 생략
DoSomething()

// 좋지 않은 예
DoSomething ()
```

— **대괄호 안에 공백을 사용하지 않습니다.**

```
// 예시: 대괄호 내부의 공백 생략
x = dataArray[index];

// 좋지 않은 예
x = dataArray[ index ];
```

— **흐름 제어 조건 앞에 하나의 공백을 사용합니다.** 흐름 비교 연산자와 괄호 사이에 공백을 추가해 보세요.

```
// 예시: 조건 앞에 공백 추가, 공백으로 괄호 분리
while (x == y)

// 좋지 않은 예
while(x==y)
```

— **비교 연산자 앞뒤로 하나의 공백을 사용합니다.**

```
// 예시: 조건 앞에 공백 추가, 공백으로 괄호 분리
if (x == y)

// 좋지 않은 예
if (x==y)
```

— **라인을 항상 짧게 작성하고 가로 공백을 고려합니다.** 표준 줄 너비를 80~120자 정도로 지정합니다. 줄이 긴 경우 그대로 두지 말고 여러 개의 작은 구문으로 나눕니다.

— **들여쓰기/계층 구조를 유지합니다.** 코드에 들여쓰기를 적용하면 가독성이 향상됩니다.

— **가독성을 높이는 효과가 없다면 열 정렬을 사용하지 않습니다.** 이런 방식의 띄어쓰기를 사용하면 변수를 정렬하는 효과는 있지만 형식과 이름을 연결하기가 어려워집니다.

단, 데이터가 많은 비트 식 또는 구조체에서는 열 정렬이 유용할 수 있습니다. 항목을 추가할수록 열 정렬 관리에 더 많은 작업이 필요할 수 있다는 점을 유념하세요. 열에서 정렬되는 부분을 변경하는 자동 포맷터도 있습니다.

```
// 예시: 형식과 이름 사이에 공백 하나 추가

    public float Speed = 12f;
    public float Gravity = -10f;
    public float JumpHeight = 2f;

    public Transform GroundCheck;
    public float GroundDistance = 0.4f;
    public LayerMask GroundMask;

// 좋지 않은 예: 열 정렬

    public float                Speed = 12f;
    public float                Gravity = -10f;
    public float                JumpHeight = 2f;
    public Transform            GroundCheck;
    public float                GroundDistance = 0.4f;
    public LayerMask            GroundMask;
```

세로 공백

세로 공백 역시 유용하게 사용할 수 있습니다. 스크립트에서 관련된 부분을 한데 모으고 필요에 따라 공백 라인을 사용하세요. 다음 제안 사항에 따라 코드를 하향식으로 정리해 보세요.

- **종속성이 있거나 유사한 메서드를 그룹화합니다.** 코드는 논리적이고 일관성이 있어야 합니다. 동일한 동작을 수행하는 메서드를 나란히 두어 다른 사람이 로직을 읽을 때 파일을 여기저기 찾아보지 않아도 되도록 합니다.
- **필요에 따라 세로 공백을 사용하여 클래스에서 서로 다른 부분을 분리합니다.** 예를 들어, 다음 사이에 빈 줄 두 개를 추가할 수 있습니다.
    - 변수 선언 및 메서드
    - 클래스 및 인터페이스
    - `if-then-else` 블록(가독성에 도움이 되는 경우)

    이 작업을 최소한으로 적용하고 해당되는 경우 스타일 가이드에 기록합니다.

Region

#region 지시문을 사용하면 C# 파일에서 코드 섹션을 축소하고 숨길 수 있어 대용량 파일을 더 쉽게 관리하고 읽을 수 있습니다.

하지만 이 가이드의 클래스 관련 일반 조언을 따른다면 클래스 크기가 관리할 수 있는 수준일 것이며 #region 지시문은 필요하지 않을 것입니다. region 뒤에 코드 블록을 숨기지 말고 코드를 작은 클래스로 나누세요. 소스 파일이 짧으면 region의 필요성이 줄어들 것입니다.

참고: 많은 개발자가 region을 코드 스멜 또는 안티패턴으로 간주합니다. 여러분의 팀은 어떤 편에 속하는지 결정하세요.

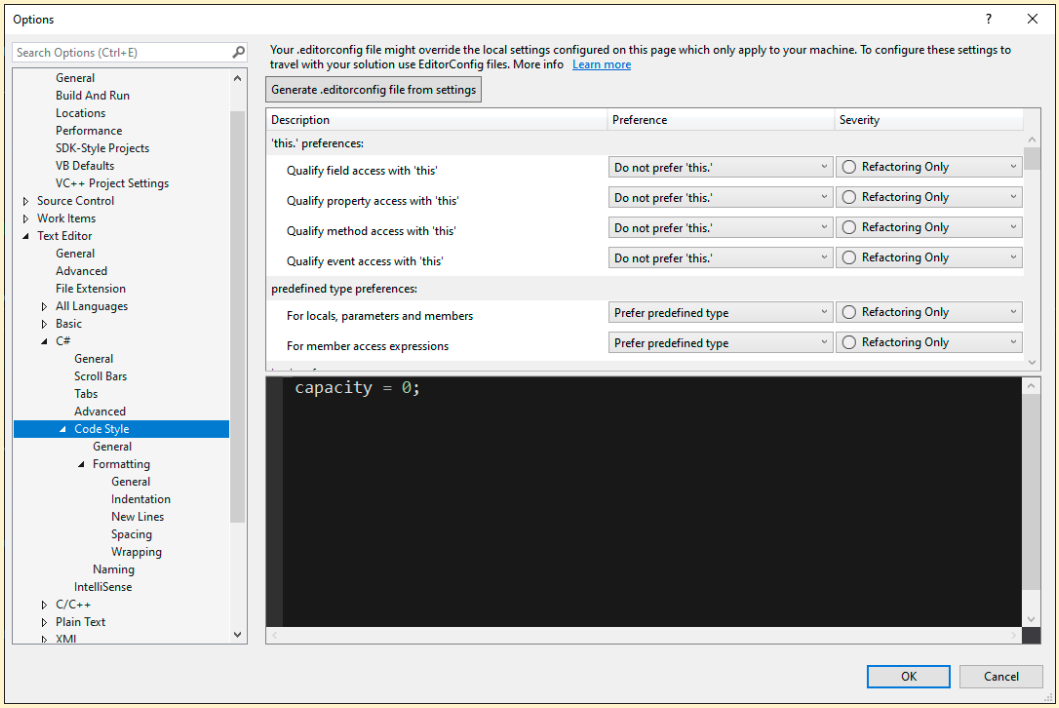
Visual Studio에서 코드 포맷 지정

포맷 지정 규칙이 매우 복잡해 보이더라도 좌절하지 마세요. 최신 IDE를 사용하면 효율적으로 설정하고 적용할 수 있습니다. 포맷 지정 규칙 템플릿을 생성한 다음 프로젝트 파일을 한 번에 전환하면 됩니다.

스크립트 에디터의 포맷 지정 규칙을 설정하려면 다음과 같은 과정을 따릅니다.

- Visual Studio(Windows)의 경우 **Tools > Options**로 이동하여 **Text Editor > C# > Code Style Formatting**을 찾습니다.

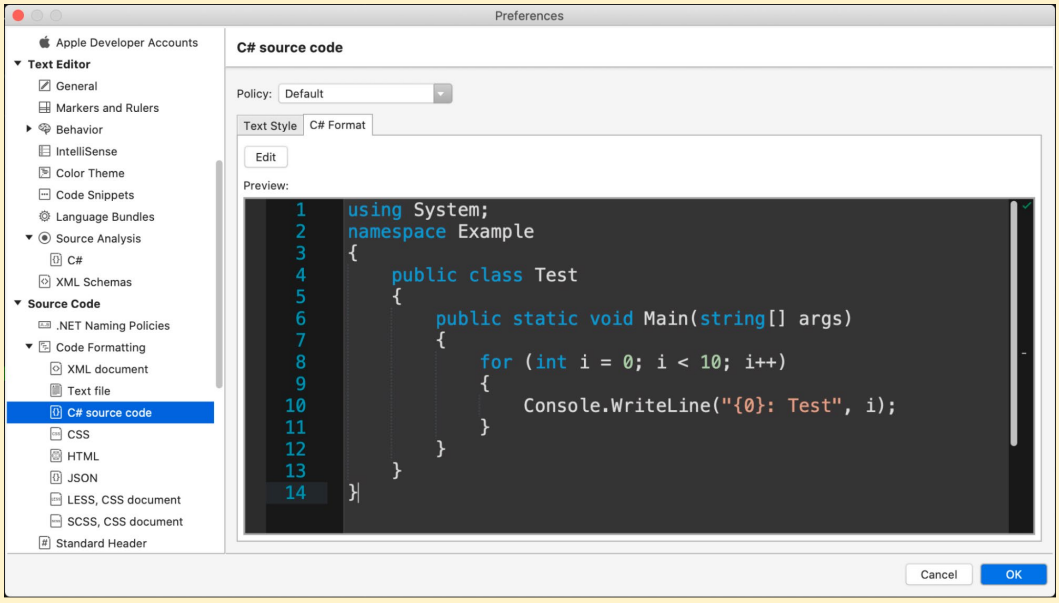
    설정을 사용하여 General, Indentation, New Lines, Spacing, Wrapping 옵션을 수정합니다.



코드 스타일 포맷 지정 옵션

— Mac용 Visual Studio의 경우 **Visual Studio > Preferences**를 선택한 다음 **Source Code > Code Formatting > C# Source Code**로 이동합니다.

상단의 Policy를 선택합니다. 그런 다음 Text Style 탭에서 띄어쓰기 및 들여쓰기를 설정합니다. C# Format 탭에서 Indentation, New Lines, Spacing, Wrapping 설정을 조정합니다.



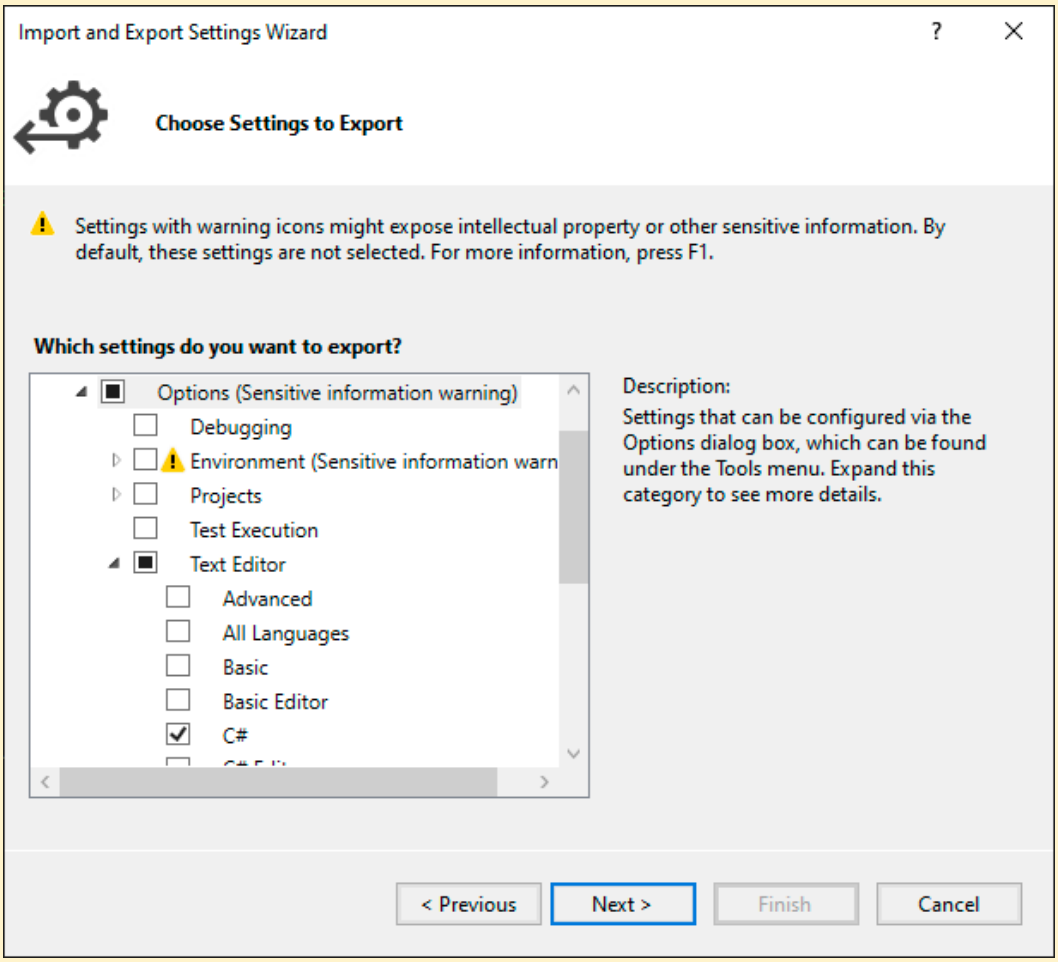
스타일 가이드 선택 사항을 미리 보여 주는 Preview 창

스크립트 파일이 스타일 가이드를 준수하도록 하려면 다음과 같은 과정을 따릅니다.

— Visual Studio(Windows)의 경우 **Edit > Advanced > Format Document**(**Ctrl+K, Ctrl+D** 단축키 조합)로 이동합니다. 공백 및 탭 정렬의 포맷만 지정하려는 경우 에디터 하단의 Run Code Cleanup(**Ctrl+K , Ctrl+E**)을 사용할 수도 있습니다.

— Mac용 Visual Studio의 경우 **Edit > Format Document**(**Ctrl+I** 단축키)로 이동합니다.

Windows에서는 **Tools > Import and Export Settings**에서 에디터 설정을 공유할 수도 있습니다. 스타일 가이드의 C# 코드 포맷 지정이 포함된 파일을 익스포트한 다음 모든 팀원이 이 파일을 임포트하도록 합니다.



공유할 C# 코드 포맷 지정 익스포트.

Visual Studio를 사용하면 간편하게 스타일 가이드를 준수할 수 있으며 포맷 지정이 마치 단축키를 사용하듯 간편해집니다.

참고: Visual Studio 설정을 임포트 및 익스포트하는 대신 EditorConfig 파일(위 참조)을 설정할 수 있습니다. 이렇게 하면 여러 IDE 간에 포맷 지정을 더 쉽게 공유할 수 있으며 버전 관리를 사용할 수 있다는 이점도 누릴 수 있습니다. 자세한 내용은 .NET 코드 스타일 규칙 옵션을 참고하세요.

클린 코드에만 해당하는 사항은 아니지만 생산성을 높여 주는 Visual Studio 팁과 노하우(Visual Studio tips & tricks to boost your productivity) GDC 세션을 확인해 보세요. 이러한 생산성 팁을 적용하면 클린 코드를 훨씬 더 쉽게 포맷하고 리팩터링할 수 있습니다.

Visual Studio Code에서 .editorconfig 파일을 설정하는 방법은 다음과 같습니다.

1. 프로젝트의 루트 디렉토리에서 새 파일을 생성하고 이름을 .editorconfig로 지정합니다.
2. .editorconfig 파일을 열고 원하는 구성 설정을 추가합니다. 다음은 C#용 구성 예시입니다.

```
# top-most EditorConfig file
root = true

# Unix-style newlines with a newline ending every file
[*]
end_of_line = lf
insert_final_newline = true

# 4 space indentation
[*.cs]
indent_style = space
indent_size = 4
charset = utf-8
trim_trailing_whitespace = true

# Tab indentation for Makefiles
[Makefile]
indent_style = tab

# Specific settings for JSON files
[*.json]
indent_style = space
indent_size = 2
```

클래스

짧은 컴퓨팅 역사에서 누구도 완벽한 소프트웨어를 작성한 적은 없습니다. 아마 앞으로도 그럴 사람은 없을 것입니다.

- 앤디 헌트, 실용주의 프로그래머 저자

로버트 C. 마틴의 클린 코드에 따르면 클래스의 첫 번째 규칙은 크기가 작아야 한다는 것입니다. 두 번째 규칙은 그보다 더 작아야 한다는 것입니다.

클래스마다 크기를 제한하면 집중도와 응집도가 높아집니다. 기존 클래스 위에 계속 추가하면 기능이 지나치게 확장되기 쉽습니다. 클래스를 짧게 유지하도록 의식적으로 노력하세요. 클래스가 무겁고 부피가 크면 내용을 읽고 문제를 해결하기 어렵습니다.

신문 비유

클래스의 소스 코드를 뉴스 기사라고 생각해 보세요. 기사의 표제와 필자란이 시선을 끄는 상단부터 읽기 시작할 것입니다. 도입부 문단에서 요약된 내용을 파악할 수 있고 아래쪽으로 읽어 내려가며 자세한 내용을 알게 됩니다.

기자들은 이러한 구조를 역피라미드라고 부릅니다. 첫 부분에서는 가장 기삿거리가 되는 내용의 개략적인 설명을 볼 수 있고 기사의 세부적인 내용은 끝까지 읽어야 알 수 있습니다.

클래스도 이 기본 패턴을 따라야 합니다. 하향식으로 구성하고 함수로 계층 구조를 형성한다고 생각하세요. 어떤 메서드는 더 높은 수준의 역할을 수행하며 전체 그림의 기반을 만듭니다. 이 메서드는 앞에 두고 낮은 수준의 함수는 구현 세부 사항과 함께 나중에 배치합니다.

예를 들어 다른 메서드 `SetInitialVelocity` 및 `CalculateTrajectory`를 참조하는 `ThrowBall`이라는 메서드를 만들 수 있습니다. 주요 동작을 설명하는 `ThrowBall`을 앞에 두고 보조 메서드를 그 아래에 추가합니다.

각각의 뉴스 기사는 짧지만 신문이나 뉴스 웹사이트에는 이러한 기사가 많이 수집되어 있습니다. 이렇게 모인 여러 기사는 통일된 기능적 완전체를 구성합니다. Unity 프로젝트도 그런 관점에서 생각하세요. Unity 프로젝트에는 더 크고 일관된 애플리케이션을 구성하기 위해 함께 두어야 하는 수많은 클래스가 있습니다.

ⓘ 클래스 구성

각 클래스에는 어느 정도 표준화가 필요합니다. 다음과 같이 클래스 멤버를 섹션으로 그룹화하여 구성하세요.

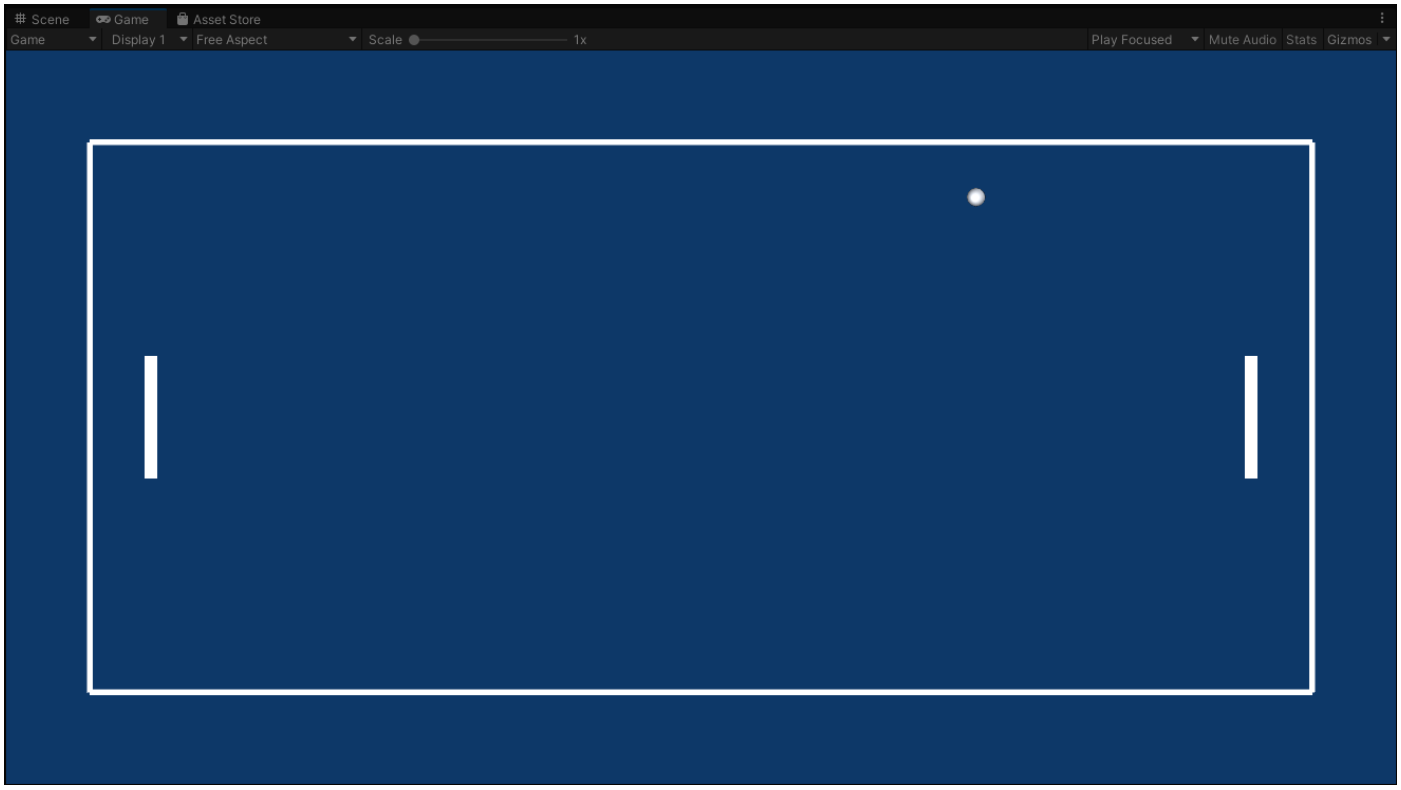
- 필드
- 프로퍼티
- 이벤트/델리게이트
- Monobehaviour 메서드(Awake, Start, OnEnable, OnDisable, OnDestroy 등)
- public 메서드
- private 메서드

Unity의 권장 클래스 명명 규칙에 따르면 소스 파일 이름은 파일의 MonoBehaviour 이름과 일치해야 합니다. 파일에 다른 내부 클래스가 있을 수도 있지만, 네임스페이스로 구분하지 않은 이상 MonoBehaviour는 파일당 하나만 있어야 합니다.

단일 책임 원칙

각 클래스를 짧게 만드는 것이 목표라는 점을 기억하세요. 소프트웨어 설계에서는 단일 책임 원칙에 따라 단순성을 지향합니다.

각 모듈, 클래스 또는 함수가 하나의 역할을 책임진다는 아이디어를 기반으로 하는 원칙입니다. 탁구 게임을 제작한다고 가정해 보세요. 탁구채, 탁구공, 벽 클래스부터 프로젝트를 시작할 수 있습니다.

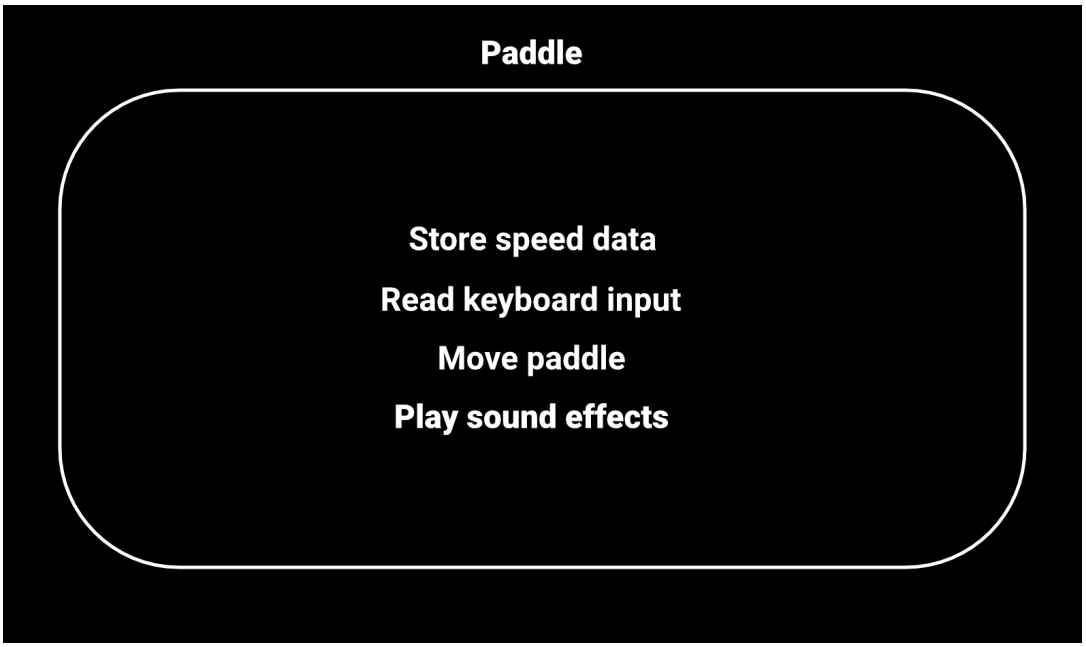


탁구 한 게임 하시겠어요?

이를테면 탁구채(Paddle) 클래스에는 다음과 같은 기능이 필요합니다.

- 탁구채 속도에 관한 기본 데이터 저장
- 키보드 입력 확인
- 반응을 통해 탁구채 이동
- 탁구공 충돌 시 음향 재생

게임 설계가 단순하므로 이 모든 사항을 기본 Paddle 클래스에 통합할 수 있습니다. 필요한 모든 작업을 수행하는 하나의 Monobehaviour를 생성하는 것도 가능합니다.

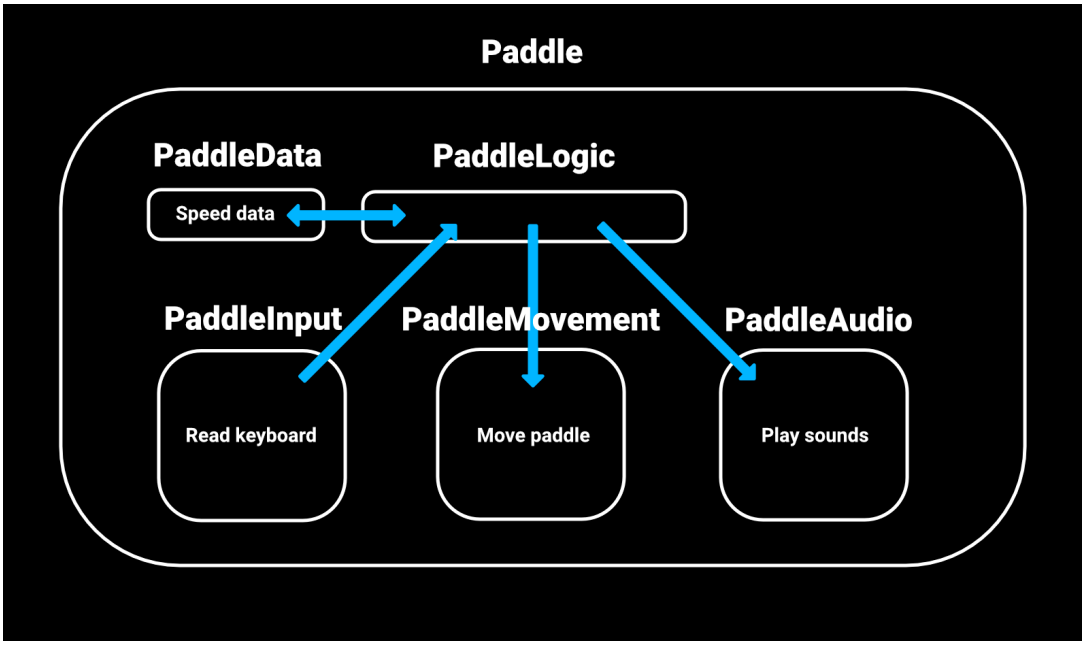


모든 작업을 수행하는 단일 Monobehaviour

하지만 모든 요소를 한 클래스의 일부로 구성하면 규모가 작더라도 책임 관계가 뒤섞여 설계가 복잡해집니다. 데이터는 입력과 긴밀히 연관되어 있는데 클래스는 로직을 양측에 적용해야 합니다. KISS 원칙과 달리 간단한 몇 가지 요소를 서로 얽히게 만들어 놓았습니다.

그보다는 Paddle 클래스를 작은 클래스로 나누고 클래스마다 하나의 책임을 부여하세요. 데이터를 자체 PaddleData 클래스로 분리하거나 ScriptableObject를 사용합니다. 그런 다음 그 외 모든 항목을 PaddleInput 클래스, PaddleMovement 클래스, PaddleAudio 클래스로 리팩터링합니다.

PaddleLogic 클래스는 PaddleInput의 입력을 처리할 수 있습니다. PaddleData의 속도 정보를 적용하면 PaddleMovement를 사용해 탁구채를 움직일 수 있습니다. 마지막으로 PaddleLogic은 탁구공이 탁구채와 충돌하면 음향을 재생하라고 PaddleAudio에 알릴 수 있습니다.



Paddle 클래스를 단일 책임으로 리팩터링

이렇게 재설계하면 각 클래스는 하나의 역할을 수행하며 쉽게 파악할 수 있는 작은 요소로 나뉩니다. 코드를 살펴보느라 여러 화면을 스크롤할 필요가 없습니다.

Paddle 스크립트는 여전히 필요하지만 다른 클래스를 하나로 묶는 역할만 수행합니다. 대부분의 기능은 다른 여러 클래스로 분할됩니다.

클린 코드가 항상 가장 컴팩트한 코드는 아닙니다. 짧은 클래스를 사용하는 경우에도 리팩터링 중에 총 라인 수가 늘어날 수 있습니다. 하지만 개별 클래스의 가독성은 더 높아집니다. 디버깅을 하거나 새로운 기능을 추가할 때도 단순화된 구조 덕분에 모든 요소를 제자리에 유지할 수 있습니다.

ⓘ 리팩터링 예시

단순한 프로젝트의 리팩터링을 자세히 살펴보려면 프로젝트가 확장될 때 코드 설계 방법을 참고하세요. 이 문서는 단일 책임 원칙을 사용하여 대형 Monobehaviour를 작은 부분으로 나누는 방법을 설명합니다.

미카엘 칼름스가 유나이트 베를린에서 발표한 독창적인 프레젠테이션, '간단한 탁구 게임을 15인 프로젝트로 확장하기(From Pong to 15-person project)'도 시청하실 수 있습니다.

메서드

각 루틴이 예상했던 결과와 거의 비슷하다면
여러분은 클린 코드를 작성 중인 것입니다.

- 워드 커닝햄, Wiki 창안자 겸 eXtreme Programming 공동 창립자

메서드는 클래스처럼 크기가 작고 하나의 책임만 가져야 합니다. 각 메서드는 하나의 동작을 설명하거나 하나의 질문에 답해야 하며 두 가지 모두를 수행할 수는 없습니다.

메서드 이름은 그 역할을 반영해 지정하는 편이 좋습니다. 예를 들어 `GetDistanceToTarget`이라는 이름은 메서드가 의도하는 목적을 명확히 나타냅니다. 프로젝트에 여러 개의 측정 단위가 사용되는 경우라면 어떤 개발자는 거리가 미터 단위로 반환된다는 사실을 명확히 전달하기 위해 `GetDistanceToTargetInMeters`라고 이름을 지정하는 편을 선호할 것입니다.

커스텀 클래스의 메서드를 생성할 때 다음과 같은 제안 사항을 고려하세요.

— **인수를 적게 사용합니다.** 인수는 메서드의 복잡도를 높일 수 있습니다. 인수의 개수를 줄여 더 쉽게 읽고 테스트할 수 있는 메서드를 만드세요.

— **지나친 오버로드를 피합니다.** 메서드 오버로드는 무한대 순열로 생성될 수 있습니다. 메서드 호출 방식을 반영하는 소수의 메서드만 골라 선택적으로 구현하세요. 메서드를 오버로드하면 각 메서드 서명에

고유한 수의 인수가 있는지 확인해 혼동을 방지해야 합니다.

— **부작용을 방지합니다.** 메서드는 이름이 나타내는 역할만 수행하면 됩니다. 역할 범위를 벗어나는 사항은 수정하지 않아야 합니다. 가능하다면 레퍼런스 대신 값으로 인수를 전달하는 것이 좋습니다. out 또는 ref 키워드를 통해 결과를 다시 전송하는 경우, 그 메서드가 정말 그 역할만 하는지 확인하세요.

부작용은 특정 작업에서 유용한 경우도 있지만 의도하지 않은 결과를 초래할 수 있습니다. 예기치 않은 동작을 줄이려면 부작용이 없는 메서드를 작성하는 게 좋습니다.

— **플래그를 전달하는 대신 다른 메서드를 만듭니다.** 메서드를 플래그에 기반해 두 가지 모드에서 작동하도록 설정하지 않습니다. 두 메서드를 서로 다른 이름으로 만들어야 합니다. 예를 들어 플래그 설정에 기반하여 각도 또는 라디안을 반환하는 `GetAngle` 메서드를 만드는 것이 아니라, `GetAngleInDegrees` 및 `GetAngleInRadians` 메서드를 만들어야 합니다.

부울 플래그를 인수로 사용하면 아무 문제도 없어 보이나 구현이 뒤얽히거나 단일 책임 관계가 손상될 수 있습니다.

메서드 대 함수

Unity와 C#에서는 '함수'라는 용어 대신 '메서드'를 사용합니다. 메서드는 클래스 또는 오브젝트의 컨텍스트 내에서 정의되는 함수이기 때문입니다. C#은 OOP(객체 지향 프로그래밍) 언어이므로 모든 요소가 클래스와 객체를 중심으로 운영되며, 그러한 객체가 수행할 수 있는 동작이 메서드입니다. 반면에 함수는 특정 작업을 수행하는 코드 블록을 가리키는 보다 일반적인 용어입니다. 절차적 프로그래밍 언어에서는 함수가 독립적으로 존재할 수 있지만 C#과 같은 객체 지향 언어에서는 함수가 클래스 내부에 캡슐화되므로 메서드라고 불립니다.

확장 메서드

확장 메서드는 봉인될 수도 있는 기능을 클래스에 추가하는 방법을 제공하며 UnityEngine API를 확장하는 방법으로 활용할 수 있습니다.

확장 메서드를 생성하려면 정적 메서드를 만들고 첫 번째 인수 앞에 `this` 키워드를 사용합니다. 이 인수가 바로 확장되는 형식입니다.

예를 들어 게임 오브젝트에서 스케일링, 회전 또는 변환을 제거하기 위해 ResetTransformation이라는 메서드를 만든다고 가정해 보세요.

첫 번째 인수로 Transform을 전달하는 정적 메서드를 생성하고 `this` 키워드를 사용하면 됩니다.

```
// 예시: 확장 메서드 정의
public static class TransformExtensions
{
    public static void ResetTransformation(this Transform transform)
    {
        transform.position = Vector3.zero;
        transform.localRotation = Quaternion.identity;
        transform.localScale = Vector3.one;
    }
}
```

그런 다음 해당 메서드를 사용하고 싶을 때 `ResetTransformation` 메서드를 호출합니다. `ResetOnStart` 클래스는 `Start` 도중 현재 트랜스폼에서 메서드를 호출합니다.

```
// 예시: 확장 메서드 호출
public class ResetOnStart : MonoBehaviour
{
    void Start()
    {
        transform.ResetTransformation();
    }
}
```

정리를 위한 목적으로 정적 클래스에서 확장 메서드를 정의합니다. 예를 들어 트랜스폼 확장 메서드용 TransformExtensions, Vector3s 확장용 Vector3Extensions 같은 클래스를 생성할 수 있습니다.

확장 메서드는 Monobehaviour를 더 생성하지 않고도 다수의 유용한 유틸리티를 빌드할 수 있습니다. 확장 메서드를 게임 개발 과정에 추가하려면 Unity Learn: Extension Methods를 참고하세요.

DRY(Don’t repeat yourself) 원칙

실용주의 프로그래머에서 앤디 헌트와 데이브 토마스는 DRY(Don’t repeat yourself) 원칙을 고안했습니다. 소프트웨어 엔지니어링에서 회자되는 이 명언은 프로그래머에게 중복 또는 반복 로직을 피하라고 조언합니다.

그렇게 하면 버그 수정 작업과 유지 관리 비용을 줄일 수 있습니다. 단일 책임 원칙을 따르면 클래스나 메서드를 수정할 때마다 관련 없는 코드 부분을 변경하지 않아도 됩니다. DRY 프로그램에서는 논리적 버그를 제거하면 해당 버그가 모든 곳에서 중단됩니다.

DRY의 반대 원칙은 WET('We enjoy typing' 또는 'Write everything twice')입니다. 코드에 불필요한 반복이 있는 프로그래밍은 WET에 해당합니다.

ParticleSystem(explosionA, explosionB) 및 AudioClip(soundA, soundB)이 두 개씩 있다고 생각해 보세요. ParticleSystem마다 자체 음향을 재생해야 하는데 이는 다음과 같은 간단한 방법으로 구현할 수 있습니다.

```
// 예시: WET(WRITE EVERYTHING TWICE)

    private void PlayExplosionA(Vector3 hitPosition)
    {
        explosionA.transform.position = hitPosition;
        explosionA.Stop();
        explosionA.Play();

        AudioSource.PlayClipAtPoint(soundA, hitPosition);
    }

    private void PlayExplosionB(Vector3 hitPosition)
    {
        explosionB.transform.position = hitPosition;
        explosionB.Stop();
        explosionB.Play();

        AudioSource.PlayClipAtPoint(soundB, hitPosition);
    }
```

여기서 각 메서드는 Vector3 위치를 사용하여 ParticleSystem을 재생 위치로 이동합니다. (이미 재생되고 있는 경우) 파티클을 중단하고 시뮬레이션을 재생합니다. 그러면 AudioSource의 정적 PlayClipAtPoint 메서드가 동일한 위치에서 음향 효과를 생성합니다.

한 메서드는 다른 메서드를 잘라서 붙여 넣고 약간의 텍스트를 교체한 버전입니다. 이렇게 하면 폭발을 생성하려 할 때마다 중복 로직이 포함된 새로운 메서드를 만들어야 합니다.

그 대신 다음과 같이 하나의 PlayFXWithSound 메서드로 리팩터링합니다.

```
// 예시: 리팩터링된 DRY 버전

    private void PlayFXWithSound(ParticleSystem particle, AudioClip clip,
Vector3 hitPosition)
    {
        particle.transform.position = hitPosition;
        particle.Stop();
        particle.Play();
        AudioSource.PlayClipAtPoint(clip, hitPosition);
    }
```

이렇게 하면 ParticleSystem과 AudioClip이 추가된 경우 동일한 메서드를 사용하여 매끄럽게 재생할 수 있습니다.

DRY 원칙을 위반하지 않고도 코드를 복제할 수 있습니다. 더 중요한 점은 로직을 복제하지 않는 것입니다.

여기서는 핵심 기능을 PlayFXWithSound 메서드로 추출했습니다. 로직을 조정해야 하는 경우 `PlayExplosionA` 및 `PlayExplosionB` 양측에서 변경하는 것이 아니라 하나의 메서드에서 변경하면 됩니다.

주석

코드는 농담과 비슷합니다. 설명이 필요하다면 좋은 농담이 아니죠.

- 코리 하우스, 소프트웨어 아키텍트 겸 저자

잘 배치된 주석은 코드의 가독성을 높여 줍니다. 과도하거나 장난스러운 주석은 역효과를 낼 수 있습니다. 다른 요소와 마찬가지로 주석을 사용할 때도 균형을 유지하세요.

KISS 원칙을 따르고 코드를 이해하기 쉬운 논리적 부분으로 나눈다면 대부분의 코드에서는 주석이 필요하지 않습니다. 변수와 함수에 이름을 적절하게 지정하면 이름만 봐도 역할을 파악할 수 있습니다.

유용한 주석은 단순히 무언가를 대답하기보다는 부족한 부분을 메우며 '이유'를 설명합니다. 곧바로 이해되지 않는 결정을 내렸나요? 설명이 필요한 까다로운 로직이 있나요? 유용한 주석은 코드 자체로는 파악할 수 없는 정보를 전달합니다.

주석과 관련하여 따라야 할 몇 가지 규칙은 다음과 같습니다.

— **잘못된 코드를 대체할 목적으로 주석을 추가하지 않습니다.** 복잡하게 얽힌 로직을 설명하기 위해 주석을 추가해야 한다면 코드를 더 명백하게 재구성하세요. 그러면 주석이 필요하지 않게 될 것입니다.

— **적합한 이름이 부여된 클래스, 변수 또는 메서드는 주석을 대신합니다.** 코드를 쉽게 이해할 수 있는 이름이 부여되었나요? 불필요한 내용을 줄이고 주석을 생략하세요.

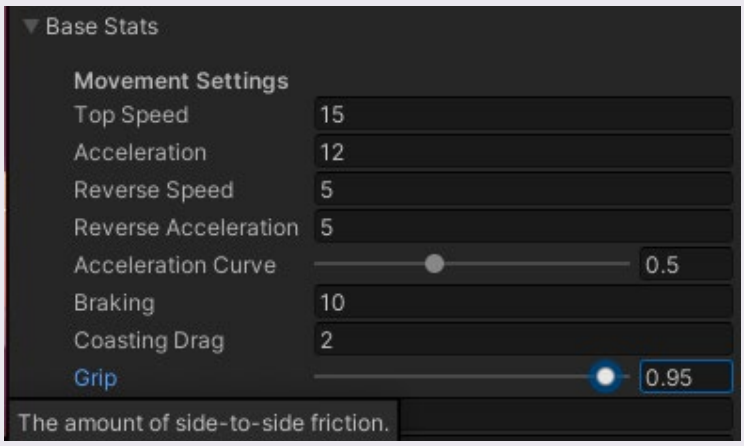
```
// 좋지 않은 예: 불필요하고 장황한 주석

// 표적이 될 대상
Transform targetToShoot;
```

— **가급적 주석을 코드 줄의 끝이 아니라 별도의 줄에 배치합니다.** 대부분의 경우, 명확성을 위해 각 주석을 별도의 줄에 작성하세요.

— **대부분의 상황에서 이중 슬래시(//) 주석 태그를 사용합니다.** 시작 부분에 장문의 주석을 작성하기보다는, 설명이 필요한 코드 바로 옆에 주석을 배치하세요. 코드와 가까운 위치에 주석을 두면 설명하는 내용과 로직을 더 쉽게 연결할 수 있습니다.

— **직렬화된 필드에는 주석 대신 툴팁을 사용합니다.** 인스펙터 필드에 설명이 필요한 경우 툴팁 속성을 추가하고 별도의 주석은 생략합니다. 그러면 툴팁이 두 가지 역할을 수행합니다.

```
// 예시: 툴팁으로 주석을 대체

[Tooltip("The amount of side-to-side friction.")]
public float Grip;
```



인스펙터의 툴팁

— **public 메서드 또는 함수 앞에** 요약 **XML 태그도 사용할 수 있습니다.** Visual Studio에서는 여러 가지 일반 XML 스타일 주석에 대해 IntelliSense를 제공합니다.

```
// 예시:
// 다음은 흔하게 사용되는 주석입니다.
// 의도와 로직 흐름, 접근 방식을 설명할 때 사용하세요.

// summary XML 태그를 사용해도 됩니다.
//
/// <summary>
/// 무기 시스템을 제어합니다(발사, 재장전, 대미지 주기 등).
/// </summary>
public void Fire()
{
    ...
}
```

— **주석 구분 기호(//)와 주석 텍스트 사이에 1개의 공백을 삽입합니다.**

— **법적 고지 사항을 추가합니다.** 주석은 라이선스 또는 저작권 정보 전달에 적합합니다. 하지만 법적 개요 전체를 코드에 삽입하기보다는 전체 법적 정보를 볼 수 있는 외부 페이지를 링크하세요.

— **주석에 스타일을 지정합니다.** 주석을 일관된 모양으로 유지하세요(예: 각 주석을 대문자로 시작하여 마침표로 끝내기). 팀에서 결정한 모든 내용을 스타일 가이드에 추가하고 준수해야 합니다.

— **주석 주위에 별표나 특수 문자로 포맷된 블록을 생성하지 않습니다.** 가독성이 떨어지며 코드가 복잡해지는 주된 원인입니다.

— **주석으로 처리한 코드를 삭제합니다.** 테스트 및 개발 도중에 구문을 주석으로 처리하여 비활성화할 수 있지만 주석으로 처리된 코드를 그대로 두면 안 됩니다. 이전 버전의 코드를 확인하려면 소스 관리를 사용하세요. 해당하는 두 코드 라인을 과감히 삭제하세요.

— **TODO 주석을 최신 상태로 유지합니다.** 작업을 완료하면 리마인더 목적으로 남긴 TODO 주석을 정리해야 합니다. 오래된 주석이 있으면 코드가 산만해집니다.

  TODO에 이름과 날짜를 추가하면 책임 소재를 보다 분명히 하고 컨텍스트 정보를 남길 수 있습니다.

  그리고 현실적으로 생각하세요. 코드에 남긴 TODO의 생성일이 5년 전이라면 절대 사용할 일이 없을 겁니다. YAGNI를 기억하세요. 구현해야 할 시점이 되기 전까지는 TODO 주석을 삭제해 두세요.

— **일지처럼 작성하지 않습니다.** 주석은 개발 일지를 위한 것이 아닙니다. 새로운 클래스를 시작할 때 모든 내용을 주석에 기록할 필요는 없습니다. 소스 관리를 적절히 사용하면 이러한 작업이 필요하지 않습니다.

— **기여자를 표시하지 않습니다.** `// added by devA` (또는 devB)와 같이 작성자 이름을 추가할 필요가 없습니다. 그런 작업은 소스 관리 시스템이 처리합니다.

UI 툴킷 명명 규칙

디버깅이 버그를 잡아내는 과정이라면,
프로그래밍은 버그를 심는 과정이라고 할 수 있습니다.

- 에츠허르 W. 데이크스트라, 컴퓨터 공학의 선구자

지금까지 C# 코드 스타일을 중점적으로 살펴보았습니다. 이제 UI 툴킷을 사용하고 UXML 및 CSS로 작업하는 상황에서의 명명 규칙도 살펴보겠습니다. UI 툴킷을 사용할 때는 문자열 식별자를 사용하여 시각적 요소와 USS(Unity 스타일 시트)를 쿼리해야 하기 때문에 표준 방식을 정해두면 오류를 줄일 수 있고 훨씬 가독성 높은 코드를 작성할 수 있습니다.

UXML의 시각적 요소와 스타일 시트 클래스에는 **BEM(Block Element Modifier)** 명명 규칙을 사용할 것을 권장합니다. BEM은 UI 툴킷 탄생의 배경이 된 CSS 및 최신 웹 개발 맥락에서 널리 사용됩니다.

요소의 BEM식 이름을 보면 해당 요소의 기능, 사용되는 곳, 주변 다른 요소와의 관계를 한눈에 파악할 수 있습니다. BEM은 다음과 같은 규칙을 기반으로 세 가지 주요 컴포넌트를 사용합니다.

```
block-name__element-name--modifier-name
```

예시를 들어 보겠습니다.

```
navbar-menu__shop-button--small
```

각 이름 부분은 라틴 문자, 숫자, 대시 기호로 구성될 수 있습니다. 각 이름 부분은 이중 밑줄__ 또는 이중 대시--로 연결됩니다.

블록 이름(block-name)은 레이아웃에서 뚜렷하게 구분되는 중요한 UI 컴포넌트인 탐색 메뉴나 캐릭터 스탯과 같은 고수준 컴포넌트를 나타냅니다. 특정 블록 전용으로 사용되지 않는 일반적인 버튼은 명시하지 않아도 됩니다(예: `button--small`) .

`element-name` 요소는 블록의 하위 요소이거나 일부이므로, 의미상 해당 블록과 연관되어 있습니다. 다시 말하면, 요소는 블록 없이 존재할 수 없고 블록에 따라 맥락이 정해집니다. 예시에서 `shop-button` 의 이름 스타일을 보면 `navbar-menu` 블록에 속하는 다른 버튼과 다른 것을 알 수 있습니다(예: `navbar-menu__shop-button`의 `shop-button`) .

새 요소가 생성자에서 하위 요소를 인스턴스화한다면 하위 요소에 관련 클래스를 할당합니다. 예를 들면 `my-block__first-child`, `my-block__other-child`와 같이 지정합니다.

마지막으로, 한정자는 블록 또는 요소의 배리에이션 또는 상태를 나타냅니다. 버튼이 눌린 경우, 텍스트 상자 항목이 선택되어 강조 표시된 경우, 또는 앞의 예시처럼 shop 버튼의 크기가 작은(`small`) 배리언트일 수 있습니다. 이 방식을 사용하면 코드를 복제하지 않고도 다양한 시나리오에 쉽게 대응할 수 있습니다.

아래에서 몇 가지 **BEM 명명 예시**를 살펴보세요.

- `menu__home-button`
- `menu__shop-button`
- `navbar-menu__shop-button--small`

BEM 클래스 이름은 이름 자체에 설명을 포함하기 때문에 개발자들이 컴포넌트의 구조와 목적을 쉽게 파악할 수 있고 프로젝트의 규모가 커져도 명확한 계층 구조를 바탕으로 스타일을 관리하고 업데이트하는 데 도움이 됩니다.

이 예시에서는 CSS 명명에 흔히 사용되는 케밥 표기법(Kebab case)이라고도 하는 하이픈 구분 방식을 사용합니다. 유니티의 다른 일반 가이드라인과 마찬가지로, 팀은 가장 적합한 명명 체계를 사용하되, 프로젝트 초기에 선택한 체계를 이후에도 일관되게 유지한다는 목표를 가져야 합니다.

이 문서와 UI 툴킷 기술 자료에서 CSS 명명 규칙에 관해 자세히 알아보세요.

UI 툴킷의 명명 규칙 팁

다음은 효과적인 명명 규칙을 위한 몇 가지 가이드라인입니다.

- 모호하지 않고 짧으면서 명확한 이름을 지으세요. 해당 요소가 UI에서 갖는 목적과 역할을 충분히 전달하는 간결하면서도 구체적인 이름을 사용해야 합니다.
- BEM 선택자에 형식 이름(Button, Label) 또는 요소 이름(#my-button)을 사용하지 마세요. 그렇게 하면 중복과 혼란을 방지할 수 있습니다. BEM 이름은 형식이 아닌 역할과 상태를 나타내야 합니다.
- 바뀔 수 있는 이름/수식어는 피하세요. 예를 들어 컬러 체계가 아직 확정되지 않은 상태라면 ‘button–red’보다는 ‘button–quit’이 바람직합니다. 표현 중심의 이름 대신 의미 중심의 이름을 사용하면, 스타일 세부 사항이 변경되더라도 이름은 계속 의미에 맞게 유지됩니다.
- 명명 규칙을 UI 툴킷 인터페이스와 관련 스프라이트 및 텍스처 등 아트 에셋으로 확장하세요. 코드와 에셋의 이름을 일관되게 지정하면 명확한 관계를 나타낼 수 있고 프로젝트를 진행하는 동안 더 나은 체계를 유지할 수 있습니다.
- 요소를 다른 프로젝트에서 사용한다면 기존 사용자 클래스 이름과 충돌하지 않도록 클래스에 접두사를 추가하세요. 네임스페이스나 접두사를 추가하면 다른 프로젝트 또는 라이브러리와 통합할 때 충돌을 피할 수 있습니다.
- 요소 인스턴스에 관련 USS 클래스를 추가하려면 생성자에서 AddToClassList()를 사용하세요. 이 메서드를 사용하면 요소가 인스턴스화되는 시점에 필요한 클래스가 추가되어 적절한 스타일이 적용되고 UI 코드에서 일관성과 명확성이 유지됩니다.

흔히 발생하는 문제

클린 코드는 우연의 산물이 아닙니다. 팀 차원에서 고민하고 코딩하는 개인의 의도적인 노력이 만든 결과물입니다.

물론 모든 일이 계획대로 되지는 않습니다. 아무리 열심히 노력한다 해도 언클린 코드는 발생하기 마련이니 그런 코드를 샅샅이 찾아내야 합니다.

코드 스멜은 프로젝트에 문제가 되는 코드가 있을 수 있다고 알려 주는 신호입니다. 다음과 같은 증상이 나타났다고 반드시 문제가 숨겨져 있는 것은 아니지만 일단 조사해 보는 편이 좋습니다.

— **의미를 알 수 없는 이름:** 누구나 멋진 수수께끼를 좋아하지만, 코딩 표준에서는 예외입니다. 클래스와 메서드, 변수에는 직관적이고 간단명료한 이름이 필요합니다. 자세한 내용은 명명 섹션을 참고하세요.

— **지나친 복잡도:** 클래스에 필요한 모든 요소를 예측하려 들면 지나친 엔지니어링이 발생합니다. 너무 많은 역할을 수행하는 긴 메서드나 커다란 클래스를 가진 절대자 오브젝트(God object)가 그 예입니다. 하나의 커다란 클래스를 전용 부분으로 작게 분할하여 각 부분에 저마다의 책임을 부여하세요.

— **경직성:** 작은 변경 하나 때문에 여러 곳을 수정해야 하는 일이 없어야 합니다. 이런 경우 단일 책임 원칙을 위반하고 있지는 않은지 다시 확인하세요.

  한 요소에 둘 이상의 책임을 부여하면 모든 상황을 예상하기 어려워지므로 더 쉽게 문제가 발생할 수 있습니다. 하나의 역할을 수행하는 메서드를 업데이트했고 업데이트된 로직이 계속 작동한다면 그 후 나머지 코드가 계속 작동할 것이라고 기대할 수 있습니다.

- **취약성**: 사소한 변경 사항을 적용했으나 모든 요소가 작동을 멈춘다면 이는 문제가 있는 경우가 많습니다.
- **부동성:** 다른 컨텍스트에서 재사용할 수 있는 코드를 작성하는 경우가 많을 것입니다. 다른 곳에 배포하는 데 많은 종속성이 필요하다면 로직의 작동 방식을 분리합니다.
- **중복 코드:** 코드를 잘라 붙여 넣은 것이 눈에 띈다면 리팩터링해야 합니다. 핵심 로직을 자체 함수로 추출하여 다른 함수에서 호출하세요. 복사하여 붙여 넣은 코드는 변경 사항이 생길 때마다 여러 위치에서 로직을 업데이트해야 하기 때문에 관리하기가 어렵습니다. 자세한 내용은 DRY 원칙 섹션을 참고하세요.
- **과도한 주석:** 직관적이지 않은 코드를 주석으로 설명할 수 있지만, 이를 과도하게 사용하는 경우가 있을 수 있습니다. 모든 변수나 구문에 주석을 계속 추가하는 것은 불필요한 일입니다. 이름만으로 의도를 파악할 수 있고 주석이 필요하지 않은, 적절한 이름이 지정된 메서드나 클래스를 사용하는편이 바람직합니다. 로직을 작은 부분으로 나누면 코드 스니핏이 짧아져 설명이 덜 필요하게 됩니다.

맺는말

프로그래밍은 제로섬 게임이 아닙니다. 동료 프로그래머에게 알려 준다 해서 손해 보는 것은 없습니다.

- 존 카맥, id Software 공동 창립자

클린 코딩 및 코드 스타일 가이드의 원칙에 대한 소개가 유익하셨기를 바랍니다.

여기에 제시된 기법은 구체적인 규칙보다는 일련의 습관과 같으며, 다른 습관과 마찬가지로 일상적인 작업을 통해 스스로 발견해야 합니다.

앞서 가이드에서 언급한 것처럼 Unity 개발자를 위한 C# 스타일 시트를 자유롭게 복사해 가이드를 작성하는 출발점으로 활용하세요. 선호하는 방식에 따라 필요한 대로 조정하여 적용하세요. 가장 중요한 것은 팀 전체가 하나의 표준에 합의하고 다 같이 준수하는 것입니다.

일관된 스타일 규칙을 적용하는 것만으로는 충분하지 않습니다. 작은 모듈 단위로 나누어 스케일링이 가능하도록 코드를 작성하세요. 장시간에 걸쳐 이루어지는 개발 여정에서는 코드를 반복적으로 재작성하게 됩니다. 요구 사항이 변경되며 제작 프로세스가 매우 힘들어질 수 있습니다.

체계적이고 가독성 높은 클린 코드베이스를 작성하는 방법에 관한 더 많은 리소스가 필요하다면 Unity 에셋 스토어에서 디자인 패턴 및 SOLID 원칙으로 코딩 스킬 업그레이드 전자책과 관련 샘플 프로젝트를 확인해 보세요.

참고 자료

이 가이드에는 컴퓨팅에서 사용되는 몇 가지 베스트 프랙티스가 수록되어 있습니다. 자세한 내용은 이 문서의 원본 스타일 가이드인 Microsoft 프레임워크 디자인 지침을 참고하세요.

클린 코드에 대해 이미 포괄적으로 다루고 있는 여러 서적을 통해서도 많은 내용을 학습할 수 있습니다. 클린 코드를 더욱 폭넓게 이해하고 싶다면 다음 서적을 참고해 보시기 바랍니다.

클린 코드: 애자일 소프트웨어 장인 정신. 로버트 C. 마틴, 2008. Prentice Hall. ISBN 978-0132350884.

실용주의 프로그래머, 20주년 기념판. 데이비드 토마스 및 앤드류 헌트, 2019, Addison Wesley, ISBN 978-0135957059.

리팩터링: 코드 구조를 체계적으로 개선하여 효율적인 리팩터링 구현하기, 제2판. 켄트 벡 및 마틴 파울러, 2018. Addison-Wesley. ISBN 978-0-134-75759-9.

테스트 주도 개발, 제1판. 켄트 벡, 2002. Addison-Wesley. ISBN 978-0321146533.

부록:
스크립트 템플릿

말은 얼마든지 할 수 있죠. 코드를 보여 주세요.

- 리누스 토발즈, Linux 및 Git 창시자

스크립트 템플릿은 MonoBehaviour와 같은 새 C# 파일을 생성하거나 Unity에서 스크립터블 오브젝트를 생성할 때(예: **Assets > Create > C# Script**) 사용되는 미리 정의된 코드 스니핏입니다.

스타일 가이드의 포맷 지정 규칙을 확립했다면 코드베이스에서 가이드라인을 일관적으로 적용하고 반복 작업을 줄이기 위해 스크립트 템플릿을 설정할 수 있습니다.

해당 파일은 아래 경로에서 확인할 수 있습니다.

Windows:
C:\Program Files\Unity\Hub\Editor\[UnityVersion]\Editor\Data\Resources\ScriptTemplates

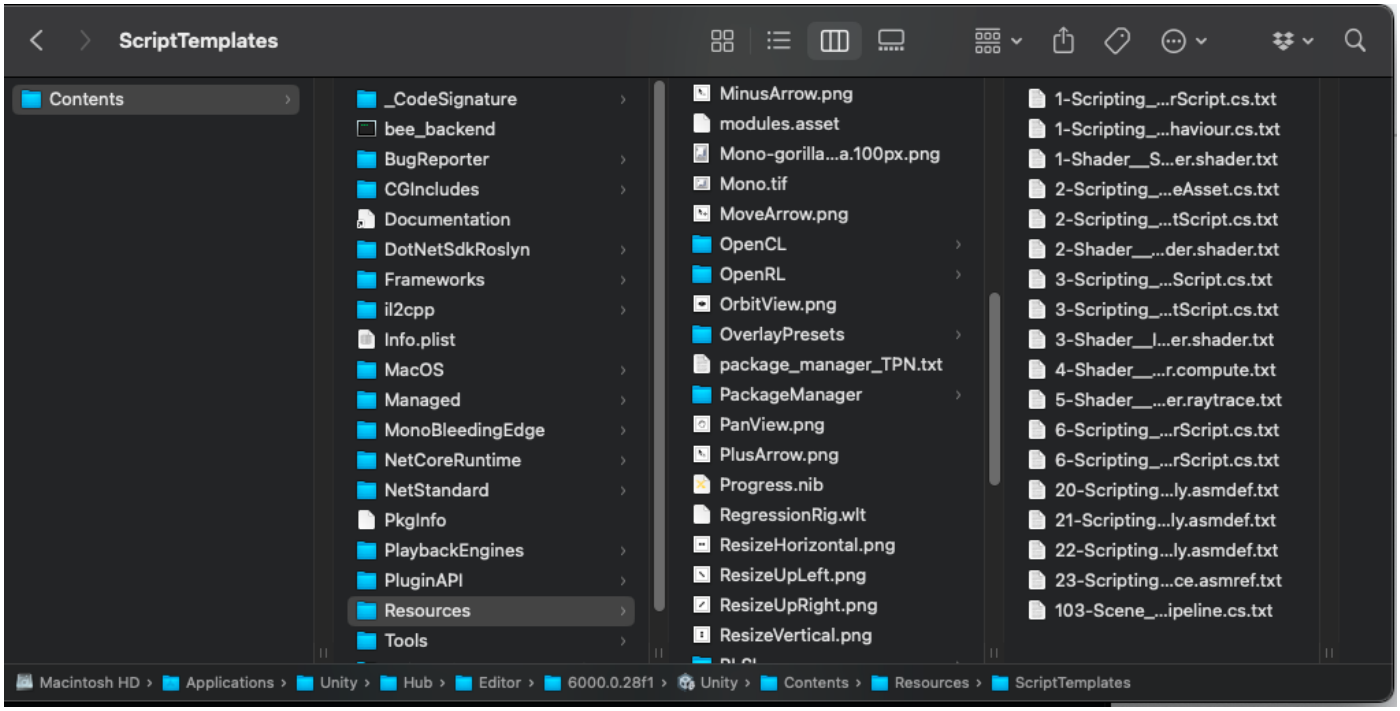
Mac:
/Applications/Unity/Hub/Editor/[UnityVersion]/Unity.app/Contents/Resources/ScriptTemplates

macOS의 경우 Unity.app 패키지의 콘텐츠를 열어 Resources 하위 디렉토리를 표시합니다.

이 경로 안에 다음과 같은 기본 템플릿이 있습니다.

`1-Scripting__MonoBehaviour Script-NewMonoBehaviourScript.cs.txt`

`2-Scripting__ScriptableObject Script-NewScriptableObjectScript.cs.txt`



Create 메뉴의 프로젝트(Project) 창에서 스크립팅된 에셋을 새로 만들 때마다 이 템플릿 중 하나가 사용됩니다.

텍스트 에디터를 사용하여 1-Scripting__MonoBehaviour Script-NewMonoBehaviourScript.cs.txt 파일을 열면 다음과 같은 내용이 표시됩니다.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class #SCRIPTNAME# : MonoBehaviour
{
    // 첫 번째 프레임 업데이트 전에 Start가 호출됨
    void Start()
    {
        #NOTRIM#
    }
    // 프레임마다 Update가 한 번 호출됨
    void Update()
    {
        #NOTRIM#
    }
}
```

다음 키워드를 참고하세요.

— **#SCRIPTNAME#**: 스크립트에 지정한 이름입니다. 이름을 커스터마이즈하지 않으면 NewBehaviourScript 같은 기본 이름이 사용됩니다.

— **#NOTRIM#:** 중괄호 사이에 공백 한 줄이 포함되도록 합니다.

스크립트 템플릿은 커스터마이즈할 수 있습니다. 예를 들어 네임스페이스를 추가하거나 기본 **Update** 메서드를 제거할 수 있습니다. 템플릿을 수정하면 스크립팅된 에셋 중 하나를 생성할 때마다 코드를 추가 또는 제거하지 않아도 됩니다.

스크립트 템플릿의 파일 이름은 다음 패턴을 따릅니다.

PriorityNumber–MenuPath–DefaultName.FileExtension.txt

대시(-) 문자가 다음과 같이 이름의 서로 다른 부분을 구분합니다.

— **PriorityNumber**는 Create 메뉴에서 스크립트가 표시되는 순서이며 숫자가 작을수록 우선순위가 더 높습니다.

— **MenuPath**에서는 Create 메뉴에 파일이 표시되는 방식을 커스터마이즈할 수 있습니다. 이중 밑줄(__)로 카테고리를 생성할 수 있습니다.

  예를 들어 ‘CustomScript__Misc__ScriptableObject’를 통해 **Create > CustomScript > Misc** 메뉴 아래에 메뉴 항목 ScriptableObject가 생성됩니다.

— **DefaultName**은 이름을 지정하지 않은 경우 에셋에 부여되는 기본 이름입니다.

— **FileExtension**은 에셋 이름에 추가되는 파일 확장자입니다.

또한 각 스크립트 템플릿에는 FileExtension에 추가된 .txt가 있습니다.

스크립트 템플릿을 특정 Unity 프로젝트에 적용하려는 경우 프로젝트의 Assets 바로 아래에 전체 ScriptTemplates 폴더를 복사하여 붙여 넣습니다(/Assets/ScriptTemplates). 원하는 경우 편집 중인 섹션만 복사해도 됩니다.

그런 다음 새 스크립트 템플릿을 생성하거나 원본 스크립트 템플릿을 원하는 대로 수정합니다. 변경하지 않으려는 경우 프로젝트에서 스크립트 템플릿을 삭제합니다. 사용하려는 파일만 복사해도 됩니다.

애플리케이션 리소스에서 원본 스크립트 템플릿을 변경해도 되지만 주의를 기울여야 합니다. 해당 버전의 Unity를 사용하는 모든 프로젝트에 영향을 주기 때문입니다.

스크립트 템플릿 커스터마이징에 대한 자세한 내용은 지원 문서를 참고하세요. 첨부된 프로젝트에서 스크립트 템플릿의 몇 가지 추가 예시도 확인하세요.

부록: 테스트 및 디버깅

디버깅은 자신이 범인이자 탐정인 범죄 영화 속 주인공이 되는 것과 다름없습니다.

- 필리페 포르테스

자동화된 테스트는 코드 품질을 개선하고 버그 수정에 드는 시간을 줄이는 데 효과적인 툴입니다. TDD(테스트 주도 개발)는 소프트웨어를 개발하면서 단위 테스트를 생성하는 개발 방법론입니다. 사실 특정 기능 함수를 만들기 전에 테스트 케이스를 작성하는 것이 일반적입니다.

소프트웨어 개발은 자동화된 모든 테스트 프로세스를 반복해서 실행하며 이뤄집니다. 소프트웨어를 먼저 작성하고 테스트 케이스를 나중에 제작하는 것과는 완전히 다릅니다. TDD에서는 코딩과 테스트, 리팩터링이 서로 얽혀 있습니다.

다음은 켄트 벡의 테스트 주도 개발에 제시된 기본 팁입니다.

1. **단일 단위 테스트 추가:** 애플리케이션에 추가하려는 새로운 기능 하나를 설명하며, 팀이나 사용자층에서 수행해야 하는 작업을 명시합니다.
2. **테스트 실행**: 새로운 기능을 프로그램에 구현하지 않았으므로 테스트는 실패합니다. 이렇게 하면 테스트 자체의 유효성도 검증할 수 있습니다. 항상 통과할 수 없는 것이 기본입니다.
3. **새로운 테스트를 통과하는 가장 단순한 코드 작성:** 새로운 단위 테스트를 통과할 정도로만 로직을 작성합니다. 이 시점에는 반드시 클린 코드일 필요가 없습니다. 단위 테스트를 통과할 수만 있다면 명쾌하지 않은 구조나 하드 코딩된 매직 넘버 등을 사용해도 됩니다.
4. **모든 테스트 통과 확인**: 완전 자동화된 테스트 모음을 실행합니다. 이전의 단위 테스트는 모두 통과할 것입니다. 새로운 코드는 새로운 테스트 요구 사항과 이전 요구 사항을 모두 충족합니다.

   그렇지 않은 경우 모든 테스트가 통과할 때까지 새 코드만 수정합니다.

5. **리팩터링**: 다시 돌아가 새로운 코드를 정리합니다. 모든 항목이 스타일 가이드를 준수하는지 확인합니다.

   논리적으로 구성되도록 코드를 이동합니다. 유사한 클래스와 메서드를 한데 모읍니다. 중복 코드를 삭제하고 식별자의 이름을 변경하여 주석의 필요성을 최소화합니다. 지나치게 긴 메서드나 클래스가 있으면 분할합니다.

   리팩터링이 끝날 때마다 자동화된 테스트 모음을 실행합니다.

6. **반복**: 새로운 기능을 추가할 때마다 이 프로세스를 진행합니다. 각 단계에서 점진적인 소규모 변경이 이루어집니다. 소스 관리에서 커밋을 자주 수행합니다. 디버깅 시 단위 테스트마다 소량의 새 코드를 검사하기만 하면 됩니다. 이렇게 하면 작업 범위가 단순해집니다. 다른 모든 방법이 실패하면 이전 커밋으로 롤백하여 다시 시작합니다.

지금까지 핵심 요점을 설명했습니다. 이 방법론을 사용하여 소프트웨어를 개발하면 필연적으로 KISS 원칙을 따르게 됩니다. 한 번에 하나의 기능을 추가하고 작업을 진행하면서 테스트하세요. 테스트마다 꾸준히 리팩터링하여 코드 정리를 정기적으로 진행하세요.

대부분의 클린 코드 방침과 마찬가지로 TDD는 단기적인 추가 작업이 필요하지만 장기적으로는 유지 관리와 가독성이 개선되는 경우가 많습니다.

Unity Test Framework

이전에 Unity 테스트 러너라고 불렸던 **UTF(Unity Test Framework)**는 Unity 개발자를 위한 표준 테스트 프레임워크를 제공합니다. UTF는 .NET 언어를 위한 오픈 소스 테스트 라이브러리인 NUnit을 사용합니다.

Unity Test Framework는 에디터(**편집 모드** 또는 **플레이 모드** 사용)와 타겟 플랫폼(예: 스탠드얼론, Android, iOS)에서 단위 테스트를 수행할 수 있습니다. 패키지 관리자를 통해 UTF를 설치합니다. 이때 기술 자료가 도움이 될 것입니다.

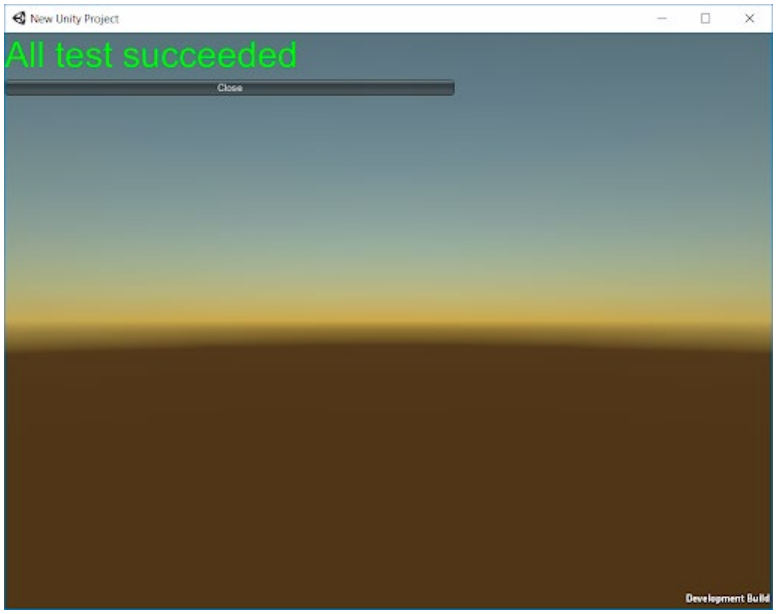
Unity Test Framework의 일반적인 워크플로는 다음과 같습니다.

— **Test Assembly**라는 새로운 테스트 모음을 생성합니다. 테스트 러너 UI는 이 프로세스를 단순화하며 프로젝트에 폴더를 생성합니다.

— **테스트를 생성합니다.** 테스트 러너 UI를 사용하면 단위 테스트로 생성할 C# 스크립트를 더 쉽게 관리할 수 있습니다. Test Assembly 폴더를 선택하고 **Assets > Create > Testing > C# Test Script로 이동합니다.** 이 스크립트를 편집하고 테스트 로직을 추가합니다.

— **테스트를 실행합니다.** 테스트 러너 UI를 사용하여 모든 단위 테스트 또는 선택된 하나의 단위 테스트를 실행합니다. JetBrains Rider를 사용하여 스크립트 에디터에서 UTF를 바로 실행할 수도 있습니다.

— **에디터나 스탠드얼론으로 플레이 모드 테스트를 추가합니다.** 기본 Test Assembly는 편집 모드에서 작동합니다. 런타임 시 단위 테스트를 수행하려면 플레이 모드에서 별도 어셈블리를 생성합니다. 스탠드얼론 빌드도 동일하게 설정합니다(에디터에 표시되는 테스트 결과 포함).



에디터 내에서 스탠드얼론 빌드의 결과를 표시하는 Test Framework.

UTF에 대한 더 많은 정보와 Unity 프로젝트의 일반적인 테스트 및 디버깅 팁을 살펴보려면 다음 리소스를 참고하세요.

— Unity Test Framework 마이크로사이트

— Unity Test Framework로 게임에 대한 자동화된 테스트를 실행하는 방법

— Roslyn 분석기로 코드 품질과 유지 관리성을 보장하는 방법

— Unity 프로젝트를 위한 테스트 및 품질 보증 팁

— Microsoft Visual Studio Code로 디버깅 속도 향상

— Microsoft Visual Studio 2022로 코드를 디버깅하는 방법

unity.com/kr