CreatAC#StyleGuide汉译

CREAT A C# STYLE GUIDE

Contnets

Table of Contents

Introduction

Creativity can be messy .
创造力可能是混乱的。

A flash of inspiration becomes a flurry of code, which then spawns a working prototype . Success! Congratulations on passing the first hurdle . However, simply getting your code to work won’t be enough . There’s much more to game development.
一闪而过的灵感变成了一阵代码风暴，然后孕育出一个可用的原型。成功了！恭喜你通过了第一个难关。然而，仅仅让你的代码运行起来是不够的。游戏开发还有更多内容。

Once your logic is functional, then the process of refactoring and cleaning up begins.
一旦你的逻辑功能正常，重构和清理的过程就开始了。

This guide compiles advice from industry experts on how to create a code style guide. Establishing ssuch a guide for each member of your team to follow will help ensure your codebase can grow your project to a commercial-scale production.
本指南汇集了行业专家的建议，介绍了如何创建代码风格指南。为团队中的每个成员建立这样的指南，将有助于确保您的代码库能够将您的项目发展成商业规模的产品。

These tips and tricks will help your development process in the long term, even if they cost you extra effort up front. A cleaner, more scalable codebase also facilitates the efficient onboarding of new developers as you expand your team.
即使这些技巧在前期会花费你额外的精力，但它们将有助于你的长期开发过程。一个更干净、更可扩展的代码库也有助于在扩展团队时有效地吸引新开发人员。

Keep your code clean to make life easier for yourself and everyone involved in the project.
保持你的代码整洁，让你自己和项目中的每个人的生活更轻松。

Contributors

贡献者

This guide was written by Wilmer Lin, a 3D and visual effects artist with over 15 years of industry experience in film and television, who now works as an independent game developer and educator. Significant contributions were also made by senior technical content marketing manager Thomas Krogh-Jacobsen and senior Unity engineers Peter Andreasen, Scott Bilas, and Robert LaCruise.
本指南由Wilmer Lin撰写，他是一名3D和视觉特效艺术家，在电影和电视行业拥有超过15年的行业经验，现在是一名独立的游戏开发者和教育工作者。高级技术内容营销经理Thomas Krogh-Jacobsen和高级Unity工程师Peter Andreasen、Scott Bilas和Robert LaCruise也做出了重大贡献。

What is clean code, anyway?

Most game developers would agree that clean code is any code that’s easy to read and maintain.
大多数游戏开发者都会认为，干净的代码是任何易于阅读和维护的代码。

Clean code is elegant, efficient, and readable.
干净的代码是优雅的、高效的和可读的。

There’s good reasons for this congruence. Something that might be obvious to you as the original author might be less apparent to another developer. By the same token, when you implement some logic now, you might not remember what that same code snippet does three months later.
这种一致性有很好的理由。对你作为原始作者来说可能是显而易见的，但对另一个开发者来说可能就不那么明显了。同样，当你现在实现一些逻辑时，你可能不会记得三个月后同样的代码片段是做什么的。

Clean code aims to make development more scalable and conform to a set of production standards, including:
干净的代码旨在使开发更具可扩展性，并符合一组生产标准，包括：

— Follow consistent naming conventions
遵循一致的命名约定

— Format your code for legibility
格式化你的代码以便阅读

— Organize classes and methods to keep them small and readable
组织类和方法，使它们小而易读

— Comment on any code that isn’t self-explanatory
对任何不是自我解释的代码进行评论

Whether you’re building a puzzler for mobile or a massive MMORPG targeted at consoles, keeping your codebase clean reduces the total cost of software maintenance. You can then implement new features or patch your existing software more easily.
无论你是为移动设备构建一个解谜游戏，还是为主机构建一个大型MMORPG，保持你的代码库干净，都会降低软件维护的总成本。然后，你可以更容易地实现新功能或修补现有的软件。

Your future teammates – and your future self – will be thankful for that.
你未来的队友——和你未来的自己——会为此感激。

DEVELOPING AS A TEAM

> ANY FOOL CAN WRITE CODE THAT A COMPUTER CAN UNDERSTAND. GOOD PROGRAMMERS WRITE CODE THAT HUMANS CAN UNDERSTAND.” – Martin Fowler, author of Refactoring
> 任何傻瓜都可以写出计算机能理解的代码。好的程序员写出人类能理解的代码。——Martin Fowler，*Refactoring* 的作者

No developer is an island. As the technical needs of your game application grow, you’ll need help. Inevitably, you’ll add more team members with diverse skill sets. Clean code introduces coding standards for your ever-expanding team so everyone is on the same page. Now everybody can work on the same project with a more uniform set of guidelines.
没有开发者是孤岛。随着你的游戏应用程序的技术需求的增长，你将需要帮助。不可避免的是，你将增加更多具有不同技能的团队成员。干净的代码为你不断扩大的团队引入了编码标准，这样每个人都在同一个页面上。现在，每个人都可以在同一个项目上使用更统一的指南。

Before looking into how to create the style guide, let’s go over some general rules to help you scale up your Unity development.
在研究如何创建样式指南之前，让我们先看一些通用规则，以帮助你扩展你的Unity开发。

KISS (keep it simple, stupid) KISS原则（保持简单，愚蠢）

Let’s face it: Engineers and developers can overcomplicate things, even though computing and programming are hard enough. Use the KISS principle of “keep it simple, stupid” as a guide for finding the simplest solution to the problem at hand.
让我们面对现实吧：工程师和开发人员可能会过于复杂化事情，即使计算和编程已经够难了。使用“保持简单，愚蠢”的KISS原则作为指南，找到手头问题的最简单解决方案。

There’s no need to reinvent the wheel if a proven and simple technique solves your challenge. Why use a fancy new technology just for the sake of using it? Unity already includes numerous solutions in its Scripting API. For example, if the existing Hexagonal Tilemap works for your strategy game, skip writing your own. The best code you can write is no code at all.
如果一个经过验证的简单技术可以解决你的挑战，就没有必要重新发明轮子。为什么要为了使用而使用一个新的花哨的技术呢？Unity已经在其脚本API中包含了许多解决方案。例如，如果现有的六角形Tilemap适用于你的策略游戏，请跳过编写你自己的代码。你可以写的最好的代码就是没有代码。

The KISS principle KISS原则

The well-known KISS principle emphasizes simplicity in design, an idea that’s been popular throughout different times, as these quotes attest:
众所周知的KISS原则强调设计的简单性，这是一个在不同时代都很流行的想法，正如下面的引用所证明的那样：

“Simplicity is the ultimate sophistication.”
– Leonardo da Vinci
“简单是终极的复杂。” ——— Leonardo da Vinci

“Make simple tasks simple!”
– Bjarne Stroustrup
“让简单的任务变得简单！” ——— Bjarne Stroustrup

“Simplicity is a prerequisite for reliability.”
– Edsger W. Dijkstra
“简单是可靠的先决条件。” ——— Edsger W. Dijkstra

“Everything should be made as simple as possible, but no simpler.”
– Albert Einstein
“一切都应该尽可能简单，但不要太简单。” ——— Albert Einstein

In programming, that means keeping your code as streamlined as possible.
Avoid adding unnecessary complexity
在编程中，这意味着尽可能简化你的代码。避免增加不必要的复杂性。

The YAGNI principle YAGNI原则

> The related YAGNI principle (“you aren’t gonna need it”) instructs you to implement features only as you need them. Don’t worry about features that you might need once the stars align. Build the simplest thing that you need now and build it to work.
> 相关的YAGNI原则（“你不会需要它”）指示你只在需要时实现功能。不要担心你可能需要的功能，一旦星星排>成一条线。构建你现在需要的最简单的东西，并使其工作。

Don’t code around the problem 不要绕过问题编码

The first step of software development is to understand what you are trying to solve. This idea might seem like common sense, but too often developers get bogged down in implementing code without understanding the actual problem, or they'll modify the code until it works without fully grasping why.
软件开发的第一步是了解你要解决的问题。这个想法可能看起来像是常识，但是开发人员经常陷入实现代码而不了解实际问题的泥潭，或者他们会修改代码，直到它工作，而不完全掌握为什么。

What if, for example, you fixed a Null Reference Exception with a quick if-null statement at the top of a method. Are you sure that was the real culprit, or was the problem a call to another method deeper inside?
例如，如果你在一个方法的顶部用一个快速的if-null语句修复了一个空引用异常。你确定这是真正的罪魁祸首吗，还是问题出在了内部更深的另一个方法的调用上?

Instead of adding code to fix a problem, investigate the root cause. Ask yourself why it’s happening rather than applying a band-aid solution.
不要添加代码来解决问题，而是调查根本原因。问问自己为什么会发生这种情况，而不是应用一个临时解决方案。

Improve incrementally, every day 每天逐步改进

Making clean code is a fluid and ongoing process. Get the whole team into this mindset. Expect code cleanup to be part of your day-to-day life as a developer. Most people don’t intend to write broken code. It just evolves that way over time. Your codebase needs constant maintenance and upkeep. Budget time for that and make sure it happens.
制作干净的代码是一个流动的、持续的过程。让整个团队进入这种心态。期望代码清理成为你作为开发人员日常生活的一部分。大多数人并不打算写破损的代码。随着时间的推移，它就会这样发展。你的代码库需要不断的维护和保养。为此预算时间，并确保它发生。

Make it good, not perfect 做好，但不苛求完美

On the flip side, don’t strive for perfection. When your code meets production standards, it’s time to commit it and move on.
另一方面，不要追求完美。当你的代码符合生产标准时，就是时候提交它并继续前进了。

Ultimately your code needs to do something. Balance implementing new functionality with code cleanup. Don’t refactor for the sake of it. Refactor when you think it will provide a benefit to you or somebody else.
最终你的代码需要做一些事情。平衡实现新功能和代码清理。不要为了重构而重构。当你认为它会给你或其他人带来好处时，再重构。

Plan, but adapt 计划，但适应

In The Pragmatic Programmer, Andy Hunt and Dave Thomas write, “Rather than construction, programming is more like gardening.” Software engineering is an organic process. Be prepared if everything does not go according to plan.
在《务实的程序员》中，Andy Hunt和Dave Thomas写道：“与其说是建筑，不如说是园艺。”软件工程是一个有机的过程。如果一切不按计划进行，请做好准备。

Even if you make the most elaborate drawing, designing a garden on paper will not guarantee results. Your plants may bloom differently than you expected. You’ll need to prune, transplant, and replace parts of your code to make this garden successful.
即使你做出最精心的图纸，设计一座花园也不能保证结果。你的植物可能会开出你意想不到的花。你需要修剪、移植和替换你的代码的一部分，使这个花园成功。

Software design isn’t quite like an architect drawing blueprints because it's more malleable and less mechanical. You’ll need to react as your codebase grows.
软件设计不像建筑师绘制蓝图，因为它更加可塑，不那么机械。你需要随着你的代码库的增长而做出反应。

Be consistent 保持一致

Once you decide how to tackle a problem, approach similar things the same way. It’s not difficult but will take constant effort. Apply this principle to everything from naming (classes and methods, casing, etc.) to organizing project folders and resources.
一旦你决定如何解决一个问题，以相同的方式处理类似的事情。这并不困难，但需要不断的努力。将这个原则应用于从命名（类和方法、大小写等）到组织项目文件夹和资源的所有内容。

Above all, have your team agree on a style guide and then follow it.
最重要的是，让你的团队同意一个样式指南，然后遵循它。

It takes a village 一个村庄

Although keeping code clean and simple is in everyone’s best interest, “clean and simple” is not the same as “easy.” Clean and simple takes effort and is hard work for beginners and experienced developers alike.
尽管保持代码的整洁和简单符合每个人的最大利益，但“整洁和简单”并不等于“简单”。对于初学者和有经验的开发人员来说，整洁和简单需要努力和艰苦的工作。

Your project will become messy if left unchecked. It’s a natural consequence of so many people working on different parts of a project. Everyone is responsible for pitching in and preventing code clutter, and each team member will need to read and follow the style guide. Cleanup is a group effort.
如果不加控制，你的项目将变得混乱。这是许多人在项目的不同部分工作的自然结果。每个人都有责任投入并防止代码混乱，每个团队成员都需要阅读并遵循样式指南。清理是一个集体的努力。

> ### A style guide for you and your team 你和你的团队的样式指南
>
> This guide focuses on the most common coding conventions you’ll encounter during Unity development. These are a subset of the Microsoft Framework Design Guidelines, which include an extensive number of rules beyond what is presented here.
> 本指南重点介绍了Unity开发过程中遇到的最常见的编码约定。这些是Microsoft Framework Design Guidelines的一个子集，其中包括大量的规则，超出了本文介绍的范围。
>
> These guidelines are recommendations, not hard and fast rules. Customize them according to your team’s preferences. Pick a style that suits everyone, and ensure they apply it.
> 这些准则是建议，而不是硬性规定。根据你的团队的喜好来定制它们。选择一种适合每个人的风格，并确保他们应用它。
>
> Consistency is king. If you follow these suggestions and need to modify your style guide in the future, a few find-and-replace operations can migrate your codebase quickly.
> 一致性是王道。如果你遵循这些建议，并且需要在将来修改你的样式指南，一些查找和替换操作可以快速迁移你的代码库。
>
> When your style guide conflicts with this document or the Microsoft Framework Design Guidelines, it should take precedence over them because this will allow your team to maintain a uniform style throughout your project.
> 当你的样式指南与本文档或Microsoft Framework Design Guidelines冲突时，它应该优先于它们，因为这将允许你的团队在整个项目中保持统一的风格。

CREATE A C# STYLE GUIDE 创建一个C#样式指南

"THERE ARE ONLY TWO HARD THINGS IN COMPUTER SCIENCE: CACHE INVALIDATION & NAMING THINGS.”– Phil Karlton, software engineer
“计算机科学中只有两件难事：缓存失效和命名事物。”——Phil Karlton，软件工程师

Your application is the collective product of individuals who might think differently from one another. A style guide helps rein in those differences to create a cohesive final product. No matter how many contributors work on a Unity project, it should feel like it’s been developed by a single author.
你的应用程序是个体的集体产品，他们可能彼此思维不同。样式指南有助于控制这些差异，创造

一个有凝聚力的最终产品。无论有多少贡献者在Unity项目上工作，它都应该感觉像是由一个作者开发的。

Microsoft and Google both offer comprehensive example guides:
Microsoft和Google都提供了全面的示例指南：

— Microsoft C# Coding Conventions Microsoft C#编码约定

— C# at Google Style Guide Google样式指南

These are excellent starting points for managing your Unity development. Each guide offers solutions for naming, formatting, and commenting. If you're a solo developer, this might feel like a constraint at first, but following a style guide is essential when working in teams.
这些都是管理你的Unity开发的绝佳起点。每个指南都提供了命名、格式化和注释的解决方案。如果你是一个独立的开发者，这可能会一开始就感觉像是一个限制，但是在团队中工作时遵循一个样式指南是必不可少的。

Think of a style guide as an initial investment that will pay dividends later. Maintaining a single set of standards can reduce the time spent on relearning if you move anyone onto another project.
把样式指南看作是一项最初的投资，它将在以后产生回报。维护一套标准可以减少在将任何人移动到另一个项目时重新学习的时间。

Style guides take the guesswork out of coding conventions and formatting.
样式指南消除了编码约定和格式化的猜测。

Consistent style then becomes a matter of following directions.
一致的风格就成了遵循指示的问题。

We created an example C# style sheet that you can also use as a reference as you assemble your own guide. Feel free to copy and tweak it as needed.
我们创建了一个示例C#样式表，你也可以在组装你自己的指南时使用它作为参考。随时根据需要复制和调整它。

Let's dive in.
让我们深入研究一下。

Naming conventions 命名约定

There's a deep psychology involved in giving something a name. A name tells us how that entity fits into the world. What is it? Who is it? What can it do for us?
给某物命名涉及到深层的心理学。一个名字告诉我们这个实体如何适应这个世界。它是什么？它是谁？它能为我们做什么？

The names of your variables, classes, and methods aren’t mere labels. They carry weight and meaning. Good naming style impacts how someone reading your program can comprehend the idea you’re trying to convey.
你的变量、类和方法的名称不仅仅是标签。它们承载着重量和意义。良好的命名风格影响着阅读你的程序的人如何理解你试图传达的思想。

Here are some guidelines to follow for naming.
以下是一些命名的指导原则。

Identifier names 标识符名称

An identifier is any name you assign to a type (class, interface, struct, delegate, or enum), member, variable, or namespace. Identifiers must begin with a letter or an underscore (*).*
*标识符是你分配给类型（类、接口、结构、委托或枚举）、成员、变量或命名空间的任何名称。标识符必须以字母或下划线（）*开头。

Avoid special characters (backslashes, symbols, Unicode characters) in your identifiers, even though C# permits them. These can interfere with certain Unity command-line tools. Steer clear of unusual characters to ensure compatibility with most platforms.
避免在你的标识符中使用特殊字符（反斜杠、符号、Unicode字符），即使C#允许它们。这些可能会干扰某些Unity命令行工具。避免使用不寻常的字符，以确保与大多数平台的兼容性。

> ### Casing terminology 大小写术语
>
> You can’t define variables with spaces in the name because C# uses the space character to separate identifiers. Casing schemes can alleviate the problem of using compound names or phrases in source code. There are several wellknown naming and casing conventions.
> 你不能用空格来定义变量的名称，因为C#使用空格字符来分隔标识符。大小写方案可以缓解在源代码中使用复合名称或短语的问题。有几种众所周知的命名和大小写约定。
>
> ### Camel case 骆驼命名法
>
> Also known as camel caps, camel case is the practice of writing phrases without spaces or punctuation, separating words with a single capitalized letter. The very first letter is lowercase. Local variables and method parameters are camel case.
> 也称为骆驼大写，骆驼命名法是一种不使用空格或标点符号的短语写法，用一个大写字母分隔单词。第一个字母是小写的。局部变量和方法参数是骆驼命名法。
>
> For example:
> examplePlayerController
> maxHealthPoints
> endOfFile

Pascal case 帕斯卡命名法

Pascal case is a variation of camel case, where the initial letter is capitalized. Use this for class and method names in Unity development. Public fields can be pascal case as well. For example:
帕斯卡命名法是骆驼命名法的一种变体，其中初始字母是大写的。在Unity开发中使用这个类和方法名。公共字段也可以是帕斯卡命名法。例如：

  ExamplePlayerController
MaxHealthPoints
EndOfFile

Snake case 蛇命名法

In this case, spaces between words are replaced with an underscore character.
For example:
在这种情况下，单词之间的空格被下划线字符替换。例如：

  example_player_controller
max_health_points
end_of_file

Kebab case 烤肉串命名法

Here, spaces between words are replaced with dashes. The words appear on a “skewer” of dash characters. For example:
在这里，单词之间的空格被破折号替换。这些单词出现在一个破折号字符的“串”上。例如：

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

The problem with kebab case is that many programming languages use the dash as a minus sign. Some languages interpret numbers separated by dashes as calendar dates.
烤肉串命名法的问题在于，许多编程语言使用破折号作为减号。一些语言将用破折号分隔的数字解释为日历日期。

Hungarian notation 匈牙利命名法

The variable or function name often indicates its intention or type. For example:
变量或函数名通常表示其意图或类型。例如：

> int iCounter
> string strPlayerName
>
> Hungarian notation is an older convention and is not common in Unity development.
> 匈牙利命名法是一个较旧的约定，在Unity开发中并不常见。

Fields and variables 字段和变量

Consider these rules for your variables and fields:
考虑一下你的变量和字段的规则:
— Use nouns for variable names: Variable names must be descriptive, clear, and unambiguous because they represent a thing or state. So use a noun when naming them except when the variable is of the type bool (see below).
用名词来命名变量:变量名必须是描述性的、清晰的和不含糊的，因为它们代表了一个事物或状态。所以在命名它们时使用名词，除非变量是bool类型(见下文)。

— Prefix Booleans with a verb: These variables indicate a true or false value. Often they are the answer to a question, such as – is the player running? Is the game over? Prefix them with a verb to make their meaning more apparent. Often this is paired with a description or condition, e.g., isDead, isWalking, hasDamageMultiplier, etc.
用动词前缀布尔值:这些变量表示一个真或假的值。通常它们是一个问题的答案，比如——玩家在跑步吗？游戏结束了吗？用一个动词前缀来使它们的意义更明显。通常这与描述或条件配对，例如，isDead，isWalking，hasDamageMultiplier等。

— Use meaningful names. Don’t abbreviate (unless it’s math): Your variable names will reveal their intent. Choose names that are easy to pronounce and search for.
使用有意义的名字。不要缩写(除非是数学):你的变量名将揭示它们的意图。选择易于发音和搜索的名称。

— Single letter variables are fine for loops and math expressions, but otherwise, don’t abbreviate. Clarity is more important than any time saved from omitting a few vowels.
单字母变量对于循环和数学表达式来说是可以的，但是除此之外，不要缩写。清晰比省略几个元音节节省的时间更重要。

— When doing quick prototyping, you can use short “junk” names and then refactor to meaningful names later.
在快速原型开发时，你可以使用短的“垃圾”名称，然后再重构为有意义的名称。

Examples to avoid 错误示范	Use instead 正确示范	Notes 注意
int d	intelapsedTimeInDays	Avoid single letter abbreviations unless a counter or expression.

Examples to avoid 错误示范	Use instead 正确示范	Notes 注意
		避免使用单个字母的缩写，除非用于计数器或表达式。
int hp, string tName, int mvmtSpeed	int healthPoints, string teamName, int movementSpeed	Variable names reveal intent. Make names searchable and pronounceable. 变量名揭示意图。使名称易于搜索和发音。
int getMovemementSpeed	int movementSpeed	Use nouns. Reserve verbs for methods unless it's a bool (below). 使用名词。除非是布尔类型（见下文），否则将动词保留给方法。
bool dead	bool isDead bool isPlayerDead	Booleans ask a question that can be answered true or false. bool会问一个可以用是或否回答的问题

— **Use pascal case for public fields. Use camel case for private variables:**
For an alternative to public fields, use Properties with a public getter (see Formatting below).
对公共字段使用帕斯卡命名法。对私有变量使用骆驼命名法: 用公共getter的属性作为公共字段的替代方法(见下文的格式化)。

— **Avoid too many prefixes or special encoding:** You can prefix private member variables with an underscore () *to differentiate them from local variables.*
避免过多的前缀或特殊编码: *你可以用下划线()*作为前缀来区分私有成员变量和局部变量。

Alternatively, use the this keyword to distinguish between member and local variables in context and skip the prefix. Public fields and properties generally don't have prefixes.
或者，使用this关键字来区分上下文中的成员和局部变量，并跳过前缀。公共字段和属性通常没有前缀。

Some style guides use prefixes for private member variables (m*), constants (*k), or static variables (s*), so the name can reveal more about the variable at a glance.*
一些样式指南使用前缀来表示私有成员变量(m)、常量(k)*或静态变量(s*)，因此名称可以一目了然地揭示变量的更多信息。

Many developers eschew these and rely on the editor instead. However, not all IDEs support highlighting and color coding, and some tools can't show rich context at all. Consider this when deciding how (or if) you will apply prefixes together as a team.
许多开发人员避免使用这些，而是依赖于编辑器。然而，并不是所有的IDE都支持高亮和颜色编码，有些工具根本不能显示丰富的上下文。在决定如何（或是否）将前缀作为一个团队一起应用时，请考虑这一点。

— **Specify (or omit) access level modifiers consistently:** If you leave off the access modifier, the compiler will assume the access level to be private. This works well, but be consistent in how you omit the default access modifier. Remember that you'll need to use protected if you want this in a subclass later.
一致地指定(或省略)访问级别修饰符: 如果你省略了访问修饰符，编译器将假定访问级别为private。这很好用，但是在省略默认访问修饰符的方式上要保持一致。记住，如果你想在后面的子类中使用这个，你需要使用protected。

Example code snippets 示例代码片段

The code snippets in this guide are non-functional and abbreviated. They're presented here to show style and formatting.
本指南中的代码片段是非功能性的和缩写的。它们在这里被呈现出来以显示样式和格式。

You can also reference this example C# style sheet for Unity developers, based on a modified version of Microsoft's Framework Design Guidelines. This represents just one example of how you can set up your team's style guide.
你也可以参考这个基于微软框架设计指南修改版本的Unity开发人员的C#样式表示例。这只是一个例子，说明你如何设置你的团队的样式指南。

Note the specific style rules found in these code examples:
注意这些代码示例中的具体样式规则:

— The default private access modifier is not omitted.
默认的私有访问修饰符没有被省略。

— Public member variables use pascal case.
公共成员变量使用帕斯卡命名法。

— Private member variables are camel case and use underscores (*) as a prefix.*
*私有成员变量是骆驼命名法，并使用下划线()*作为前缀。

— Local variables and parameters use camel case with no prefix.
局部变量和参数使用骆驼命名法，没有前缀。

— Public and private member variables are grouped together.
公共和私有成员变量被分组在一起。

Review each rule in the example style guide and customize it to your team's preferences. The specifics of an individual rule are less important than having everyone agree to follow it consistently. When in doubt, rely on your team's own guide to settle any style disagreements.
审查每个规则在示例样式指南，并根据你的团队的偏好进行自定义。一个单独规则的具体内

> 容比起每个人都同意一致地遵循它来说并不重要。当你怀疑的时候，依靠你的团队自己的指南来解决任何风格上的分歧。

```
// EXAMPLE: public and private variables
public float DamageMultiplier = 1.5f;
public float MaxHealth;
public bool IsInvincible;

private bool _isDead;
private float _currentHealth;

// parameters
public void InflictDamage(float damage, bool isSpecialDamage)
{
  // local variable
   int totalDamage = damage;
  // local variable versus public member variable
  if (isSpecialDamage)
  {
    totalDamage *= DamageMultiplier;
  }
  // local variable versus private member variable
  if (totalDamage > _currentHealth)
  {
    /// ...
  }
}
```

— Use one variable declaration per line: It's less compact, but enhances readability.
每行使用一个变量声明:它不够紧凑，但增强了可读性。

— Avoid redundant names: If your class is called Player, you don't need to create member variables called PlayerScore or PlayerTarget. Trim them down to Score or Target.
避免冗余的名称:如果你的类被称为Player，你不需要创建名为PlayerScore或PlayerTarget的成员变量。把它们缩减到Score或Target。

— Avoid jokes or puns: While they might elicit a chuckle now, the infiniteMonkeys or dudeWheresMyChar variables won't hold up after a few dozen reads.
避免开玩笑或双关语:虽然它们现在可能会引起一阵笑声，但是在读了几十次之后，无限的猴子或者是我的角色在哪里的变量就不会坚持下去了。

— Use the var keyword for implicitly typed local variables if it helps readability and the type is obvious: Specify when to use var in your style guide. For example, many developers avoid var when it obscures the variable's type or with primitive types outside a loop.

如果它有助于可读性并且类型是显而易见的，使用var关键字来隐式地声明局部变量:在你的样式指南中指定何时使用var。例如，许多开发人员在它模糊了变量的类型或在循环外使用原始类型时避免使用var。

Generally, use var when it makes the code easier to read (e.g., with long type names) and the type is not ambiguous.
通常，当它使代码更容易阅读(例如，使用长类型名称)并且类型不是模糊的时候，使用var。

```
// EXAMPLE: good use of var
var powerUps = new List<PowerUps>();
var dictionary = new Dictionary<string, List<GameObject>>();
// AVOID: potential ambiguity
var powerUps = PowerUpManager.GetPowerUps();
```

Enums 枚举

Enums are special value types defined by a set of named constants. By default, the constants are integers, counting up from 0.
枚举是由一组命名常量定义的特殊值类型。默认情况下，常量是整数，从0开始计数。

Use pascal case for enum names and values. You can place public enums outside of a class to make them global. Use a singular noun for the enum name.
对枚举名称和值使用帕斯卡命名法。你可以将公共枚举放在类外面，使它们成为全局的。对枚举名称使用单数名词。

> Note: Bitwise enums marked with the System.FlagsAttribute attribute are the exception to this rule. You typically pluralize these as they represent more than one type.
> 注意:用System.FlagsAttribute属性标记的位枚举是这个规则的例外。你通常把它们变成复数，因为它们代表的不止一种类型。

```
// EXAMPLE: enums use singular nouns
public enum WeaponType
{
    Knife,
    Gun,
    RocketLauncher,
    BFG
}

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

// EXAMPLE: but a bitwise enum is plural
[Flags]
public enum AttackModes
{
    // Decimal // Binary
    None = 0, // 000000
    Melee = 1, // 000001
    Ranged = 2, // 000010
    Special = 4, // 000100
    // It seems like the code snippet is incomplete for MeleeAn
}
```

Classes and interfaces 类和接口

Follow these standard rules when naming your classes and interfaces:
在命名类和接口时，请遵循这些标准规则:

— **Use pascal case nouns for class names.**
对类名使用帕斯卡命名法。

— **If you have a Monobehaviour in a file, the source file name must match:** You may have other internal classes in the file, but only one Monobehaviour should exist per file.
如果你在一个文件中有一个Monobehaviour，源文件名必须匹配: 你可能在文件中有其他内部类，但是每个文件中只能有一个Monobehaviour。

— **Prefix interface names with a capital I:** Follow this with an adjective that describes the functionality.
用大写的I前缀接口名称: 用一个描述功能的形容词来跟随它。

```
// EXAMPLE: Class formatting
public class ExampleClass : MonoBehaviour
{
    public int PublicField;
    public static int MyStaticField;

    private int _packagePrivate;
    private int _myPrivate;

    private static int _myPrivate;
    protected int _myProtected;

    public void DoSomething()
    {

    }
}

// EXAMPLE: Interfaces
public interface IKillable
{
    void Kill();
}

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

Methods 方法

In C#, every executed instruction is performed in the context of a method.
在C#中，每个执行的指令都是在方法的上下文中执行的。

> **Note**: “function” and “method” are often used interchangeably in Unity development. However, because you can’t write a function without incorporating it into a class in C#, “method” is the accepted term.
> 注意:在Unity开发中，“function”和“method”经常可以互换使用。然而，因为你不能在C#中写一个不包含在类中的函数，“method”是被接受的术语。

Methods perform actions, so apply these rules to name them accordingly:
方法执行操作，所以根据这些规则来命名它们:

— **Start the name with a verb:** Add context if necessary. e.g., GetDirection, FindTarget, etc.
用动词开头: 如果需要，添加上下文。例如，GetDirection，FindTarget等。

— **Use camel case for parameters:** Format parameters passed into the
method like local variables.
对参数使用骆驼命名法: 对传递到方法中的参数进行格式化，就像对局部变量进行格式化一样。

— **Methods returning bool should ask questions:** Much like Boolean variables themselves, prefix methods with a verb if they return a true-false condition This phrases them in the form of a question, e.g., IsGameOver HasStartedTurn.

返回bool的方法应该提出问题: 就像布尔变量本身一样，如果方法返回一个真假条件，就用一个动词作为前缀。这样就可以用一个问题来表达它们，例如，IsGameOver，HasStartedTurn。

```
// EXAMPLE: Methods start with a verb
public void SetInitialPosition(float x, float y, float z)
{
transform.position = new Vector3(x, y, z);
}
// EXAMPLE: Methods ask a question when they return bool
public bool IsNewPosition(Vector3 currentPosition)
{
return (transform.position == newPosition);
}
```

Events and event handlers 事件和事件处理程序

Events in C# implement the Observer pattern. This software design pattern defines a relationship in which one object, the subject (or publisher), can notify a list of dependent objects called observers (or subscribers). Thus, the subject can broadcast state changes to its observers without tightly coupling the objects involved.
C#中的事件实现了观察者模式。这种软件设计模式定义了一种关系，其中一个对象，主题(或发布者)，可以通知一组称为观察者(或订阅者)的依赖对象。因此，主题可以向它的观察者广播状态变化，而不会紧密地耦合涉及的对象。

Several naming schemes exist for events and their related methods in the subject and observers. Try these practices:
存在许多命名方案来命名主题和观察者中的事件及其相关方法。试试这些做法:

— **Name the event with a verb phrase:*** Choose a name that communicates the state change accurately. Use the present or past participle to indicate events “before” or “after.” For example, specify “OpeningDoor” for an event before opening a door or “DoorOpened” for an event afterward.
用动词短语命名事件: 选择一个准确地传达状态变化的名称。使用现在时或过去分词来表示“之前”或“之后”的事件。例如，为打开门之前的事件指定“OpeningDoor”，为打开门之后的事件指定“DoorOpened”。

— **Use the System.Action delegate for events:** In most cases, the Action<T> delegate can handle the events needed for gameplay. You may pass
anywhere from 0 to 16 input parameters of different types with a return type of void. Using the predefined delegate saves code.
对事件使用System.Action委托: 在大多数情况下，Action<T>委托可以处理游戏过程中需要的事

件。你可以传递0到16个不同类型的输入参数，返回类型为void。使用预定义的委托可以节省代码。

> **Note**: You can also use the EventHandler or EventHandler<TEventArgs> delegates. Agree as a team on how everyone will implement events.
> 注意:你也可以使用EventHandler或EventHandler委托。作为一个团队，你们可以商定如何实现事件。

```
// EXAMPLE: Events

// using System.Action delegate

public event Action OpeningDoor; // event before
public event Action DoorOpened; // event after

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

— **Prefix the event raising method (in the subject) with "On":** The subject that invokes the event typically does so from a method prefixed with "On," e.g. "OnOpeningDoor" or "OnDoorOpened."
用"On"前缀事件引发方法(在主题中): 调用事件的主题通常是从一个以"On"为前缀的方法中调用的，例如"OnOpeningDoor"或"OnDoorOpened"。

```
// raises the Event if you have subscribers
public void OnDoorOpened()
{
  DoorOpened?.Invoke();
}
public void OnPointsScored(int points)
{
  PointsScored?.Invoke(points);
}
```

— **Prefix the event handling method (in the observer) with the subject's name and underscore (_):** If the subject is named "GameEvents," your observers can have a method called "GameEvents*OpeningDoor" or "GameEvents_DoorOpened."*
***用主题的名称和下划线(*)前缀事件处理方法(在观察者中):** 如果主题的名称是"GameEvents"，你的观察者可以有一个名为"GameEvents_OpeningDoor"或"GameEvents_DoorOpened"的方法。

Note that this is called the "event handling method", not to be confused with the EventHandler delegate.

注意，这被称为“事件处理方法”，不要与EventHandler委托混淆。

Decide a consistent naming scheme for your team and implement those rules in your style guide.
为你的团队决定一个一致的命名方案，并在你的样式指南中实现这些规则。

— **Create custom EventArgs only as necessary:** If you need to pass custom data to your Event, create a new type of EventArgs, either inherited from System.EventArgs or from a custom struct.
只在必要时创建自定义的EventArgs: 如果你需要向你的事件传递自定义数据，请创建一个新的EventArgs类型，它要么继承自System.EventArgs，要么继承自自定义结构。

```
// define an EventArgs if needed
// EXAMPLE: read-only, custom struct used to pass an ID and Color
public struct CustomEventArgs
{
  public int ObjectID { get; }
  public Color Color { get; }

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

Namespaces 命名空间

Use a namespaces to ensure that your classes, interfaces, enums, and so on won’t conflict with existing ones from other namespaces or the global namespace. Namespaces can also prevent conflicts with third-party assets from the Asset Store.
使用命名空间来确保你的类、接口、枚举等不会与其他命名空间或全局命名空间中的现有命名空间发生冲突。命名空间也可以防止与来自资产商店的第三方资产发生冲突。

When applying namespaces:
在应用命名空间时:

— Use pascal case without special symbols or underscores.
使用帕斯卡命名法，不使用特殊符号或下划线。

— Add a using directive at the top of the file to avoid repeated typing of the namespace prefix.
在文件顶部添加一个using指令，以避免重复输入命名空间前缀。

— Create sub-namespaces as well. Use the dot(.) operator to delimit the name levels, allowing you to organize your scripts into hierarchical categories. For example, you can create MyApplication.GameFlow, MyApplication.AI, MyApplication.UI, and so on to hold different logical components of your game.
也可以创建子命名空间。使用点(.)运算符来分隔名称级别，允许你将你的脚本组织成分层的类别。例如，你可以创建MyApplication.GameFlow，MyApplication.AI，MyApplication.UI等来保存游戏的不同逻辑组件。

```
namespace Enemy
{
  public class Controller1 : MonoBehaviour
  {
    ...
  }
  public class Controller2 : MonoBehaviour
  {
    ...
  }
}
```

In code, these classes are referred to as Enemy.Controller1 and Enemy. Controller2, respectively. Add a using line to save typing out the prefix:
在代码中，这些类分别被称为Enemy.Controller1和Enemy.Controller2。添加一个using行来保存输入前缀:

When the compiler finds the class names Controller1 and Controller2, it understands you mean Enemy.Controller1 and Enemy.Controller2.
当编译器找到类名Controller1和Controller2时，它会理解你的意思是Enemy.Controller1和Enemy.Controller2。

```
using Enemy;
```

If the script needs to refer to classes with the same name from different namespaces, use the prefix to differentiate them. For instance, if you have a Controller1 and Controller2 class in the Player namespace, you can write out Player.Controller1 and Player.Controller2 to avoid any conflicts. Otherwise, the compiler will report an error.
如果脚本需要引用来自不同命名空间的相同名称的类，请使用前缀来区分它们。例如，如果你在Player命名空间中有一个Controller1和Controller2类，你可以写出Player.Controller1和Player.Controller2来避免任何冲突。否则，编译器将报告一个错误。

Formatting 格式化

"IF YOU WANT YOUR CODE TO BE EASY TO WRITE, MAKE IT EASY TO READ.”
– Robert C. Martin, author of Clean Code and Agile Software Development
“如果你想让你的代码易于编写，那就让它易于阅读。”——《代码整洁之道》和《敏捷软件开发》的作者罗伯特·C·马丁

Along with naming, formatting helps reduce guesswork and improves code clarity. By following a standardized style guide, code reviews become less about how the code looks and more about what it does.
除了命名，格式化还有助于减少猜测，提高代码的清晰度。通过遵循标准化的样式指南，代码审查变得不再关注代码的外观，而是关注代码的功能。

When constructing a style guide, personalize how your team will format your code. Consider each of the following code formatting suggestions when setting up your Unity dev style guide. Omit, expand, or modify these example rules to fit your team’s needs.
在构建样式指南时，个性化你的团队将如何格式化你的代码。在设置Unity开发样式指南时，请考虑以下每个代码格式化建议。省略、扩展或修改这些示例规则以适应你的团队的需求。

In all cases, consider how your team will implement each formatting rule and then have everyone apply it uniformly. Refer back to your team’s style to resolve any discrepancies. The less you think about formatting, the more you can work on something else.
在所有情况下，考虑你的团队将如何实现每个格式化规则，然后让每个人都统一地应用它。回顾你团队的风格来解决任何不一致。你越少考虑格式，你就越能在其他事情上工作。

Let’s take a look at formatting guidelines.
让我们来看看格式化指南。

Properties 属性

A property provides a flexible mechanism to read, write, or compute class values. Properties behave as if they were public member variables, but in fact they’re special methods called accessors. Each property has a get and set method to access a private field, called a backing field.
属性提供了一种灵活的机制来读取、写入或计算类值。属性的行为就像它们是公共成员变量一样，但实际上它们是称为accessors的特殊方法。每个属性都有一个get和set方法来访问一个私有字段，称为backing field。

In this way, the property [encapsulates](https://en.wikipedia.org/wiki/Encapsulation_(computer_programming #Information_hiding) the data, hiding it from unwanted changes by the user or external objects. The getter and setter each have their own access modifier, allowing your property to be read-write, read-only, or write-only.

这样，属性封装了数据，将其隐藏在用户或外部对象的不需要的更改中。getter和setter各自有自己的访问修饰符，允许你的属性是读写、只读或只写的。

You can also use the accessors to validate or convert the data (e.g., verify that the data fits your preferred format or change a value to a particular unit).
你还可以使用访问器来验证或转换数据(例如，验证数据是否符合你的首选格式或将值更改为特定的单位)。

The syntax for properties can vary, so your style guide should define how to format them. Use these tips to keep properties consistent in your code:
属性的语法可能会有所不同，所以你的样式指南应该定义如何格式化它们。使用这些提示来保持代码中的属性一致:

— **Use expression-bodied properties for single line read-only properties (=>)**: This returns the private backing field.
对单行只读属性使用表达式主体属性(=>): 这将返回私有的backing field。

```
// EXAMPLE: expression-bodied properties
public class PlayerHealth
{
    // the private backing field
    private int maxHealth;

    // read-only, returns backing field
    public int MaxHealth => maxHealth;

    // equivalent to:
    // public int MaxHealth { get; private set; }
}
```

— **Everything else uses the older { get; set; } syntax:** If you just want to expose a public property without specifying a backing field, use the Auto-Implemented Property.
其他所有的都使用旧的{ get; set; }语法: 如果你只想公开一个属性而不指定一个backing field，请使用Auto-Implemented Property。

Apply the expression-bodied syntax for the set and get accessors.
对set和get访问器应用表达式主体语法。

Remember to make the setter private if you don't want to give write access. Align the closing with the opening brace for multi-line code blocks.
如果你不想给写入访问权限，请记住将setter设置为私有。将关闭与打开大括号对齐以进行多行代码块。

```
// EXAMPLE: expression bodied properties
public class PlayerHealth
{
    // backing field
    private int _maxHealth;

    // explicitly implementing getter and setter
    public int MaxHealth
    {
        get => _maxHealth;
        set => _maxHealth = value;
    }

    // write-only (not using backing field)
    public int Health { private get; set; }

    // write-only, without an explicit setter
    public void SetMaxHealth(int newMaxValue) => _maxHealth = newMaxValue;
}
```

Serialization 序列化

Script serialization is the automatic process of transforming data structures or object states into a format that Unity can store and reconstruct later. For performance reasons, Unity handles serialization differently than in other programming environments.
脚本序列化是将数据结构或对象状态自动转换为Unity可以存储和重建的格式的过程。出于性能的考虑，Unity处理序列化的方式与其他编程环境不同。

Serialized fields appear in the Inspector, but you cannot serialize static, constant, or read-only fields. They must be either public or tagged with the [SerializeField] attribute. Unity only serializes certain field types, so refer to the documentation page for the complete set of serialization rules.
序列化字段出现在检查器中，但你不能序列化静态、常量或只读字段。它们必须是公共的，或者用[SerializeField]属性标记。Unity只序列化某些字段类型，所以请参考文档页面以获取完整的序列化规则集。

Observe a few basic guidelines when working with serialized fields:
在使用序列化字段时，请遵循一些基本准则:

— **Use the [SerializeField] attribute:** The SerializeField attribute can work with private or protected variables to make them appear in the Inspector. This encapsulates the data better than marking the variable public and prevents an external object from overwriting its values.

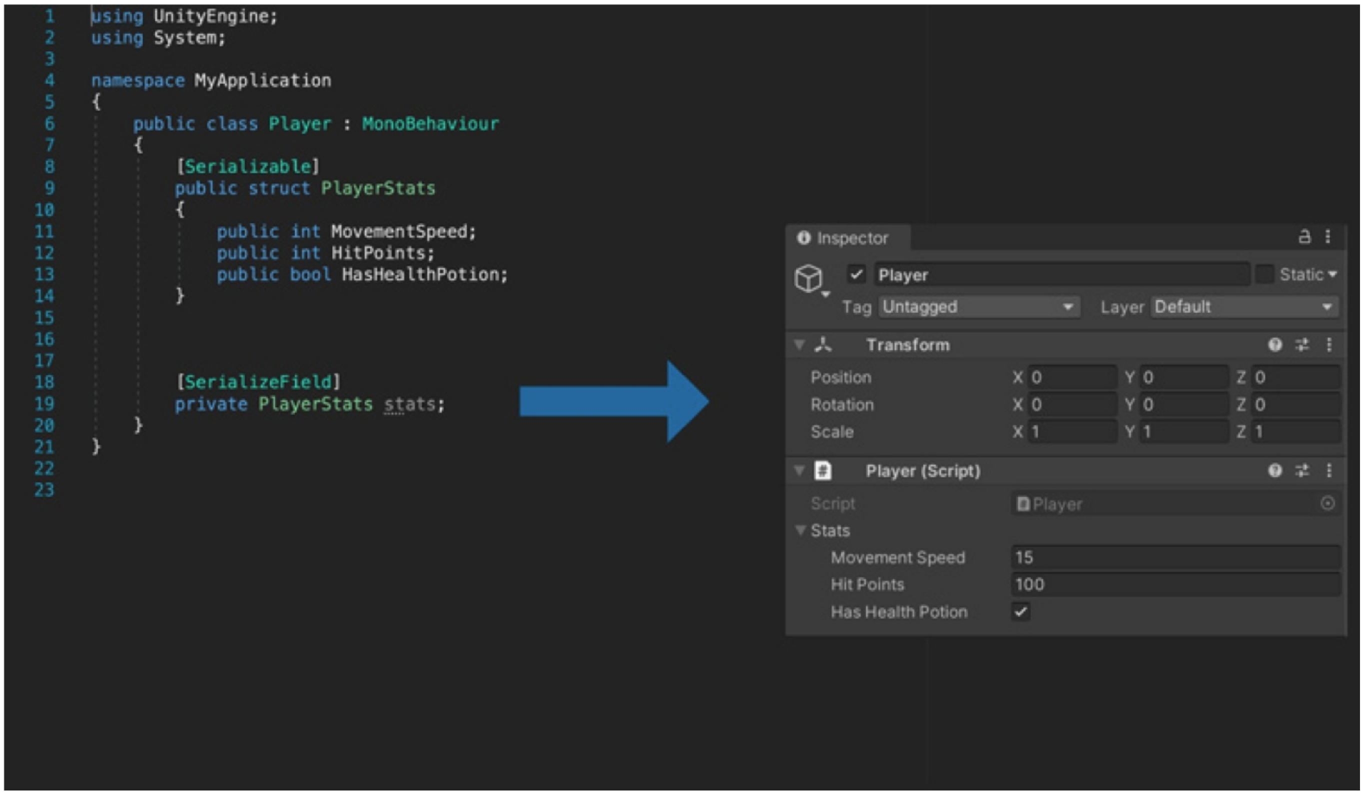
使用[SerializeFiled]属性: SerializeField属性可以与私有或受保护的变量一起使用，使它们出现在检查器中。这比将变量标记为public更好地封装了数据，并防止外部对象覆盖其值。

— **Use the Range attribute to set minimum and maximum values:** The [Range(min, max)] attribute is handy if you want to limit what the user can assign to a numeric field. It also conveniently represents the field as a
slider in the Inspector.
使用Range属性来设置最小值和最大值: 如果你想限制用户可以分配给数字字段的内容，[Range(min, max)]属性是很方便的。它还可以方便地在检查器中将字段表示为滑块。

— **Group data in serializable classes or structs to clean up the Inspector:** Define a public class or struct and mark it with the [Serializable] attribute. Define public variables for each type you want to expose in the Inspector.
在可序列化的类或结构中对数据进行分组，以清理检查器: 定义一个公共类或结构，并用[Serializable]属性标记它。为你想在检查器中公开的每种类型定义一个公共变量。

```
// EXAMPLE: a serializable class for PlayerStats
using System;
using UnityEngine;

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

    // EXAMPLE: The private field is visible in the Inspector
    [SerializeField]
    private PlayerStats _stats;
}
```

Reference this serializable class from another class. The resulting variables appear within collapsible units in the Inspector.
从另一个类引用这个可序列化的类。结果变量出现在检查器中的可折叠单元中。



```
using UnityEngine;
using System;

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


        [SerializeField]
        private PlayerStats stats;
    }
}
```

Inspector			
✓ Player			Static
Tag Untagged		Layer Default	
Transform			
Position	X 0	Y 0	Z 0
Rotation	X 0	Y 0	Z 0
Scale	X 1	Y 1	Z 1
Player (Script)			
Script	Player		
Stats			
Movement Speed	15		
Hit Points	100		
Has Health Potion	✓		

A serializable class or struct can help organize the Inspector.
可序列化的类或结构可以帮助组织inspector。

Brace or indentation style 大括号或缩进样式

There are two common indentation styles in C#:
C#中有两种常见的缩进样式:

— The Allman style places the opening curly braces on a new line, also known as the BSD style (from BSD Unix).
Allman style将左大括号放在新行上，也称为BSD样式(来自BSD Unix)。

— The K&R style, or “one true brace style,” keeps the opening brace on the same line as the previous header.
K&R style，或者“one true brace style”，将左大括号保持在与前一个头相同的行上。

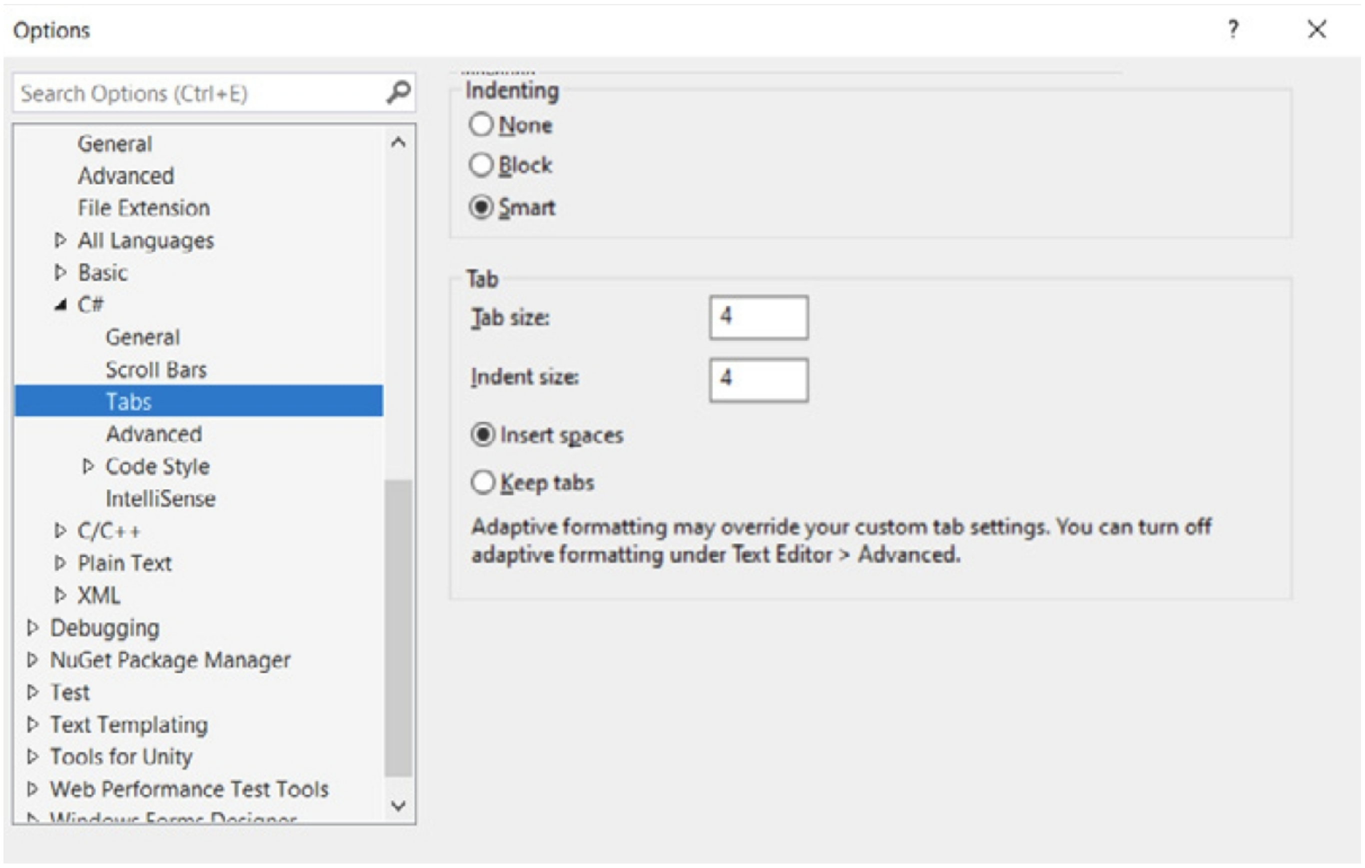
```
// EXAMPLE: Allman or BSD style puts opening brace on a new line.
void DisplayMouseCursor(bool showMouse)
{
    if (!showMouse)
    {
        Cursor.lockState = CursorLockMode.Locked;
        Cursor.visible = false;
    }
    else
    {
        Cursor.lockState = CursorLockMode.None;
        Cursor.visible = true;
    }
}

// EXAMPLE: K&R style puts opening brace on the previous line.
void DisplayMouseCursor(bool showMouse){
    if (!showMouse) {
        Cursor.lockState = CursorLockMode.Locked;
        Cursor.visible = false;
    }
    else {
        Cursor.lockState = CursorLockMode.None;
        Cursor.visible = true;
    }
}
```

There are variations on these indentation styles as well. The examples in this guide use the Allman style from the Microsoft Framework Design Guidelines.Regardless of which one you choose as a team, make sure everyone follows the same indentation and brace style.
这些缩进样式也有变化。本指南中的示例使用了Microsoft Framework Design Guidelines中的Allman样式。无论你的团队选择哪种样式，都要确保每个人都遵循相同的缩进和大括号样式。

Try these tips:
试试这些提示:

— **Decide on a uniform indentation:** This is typically four or two spaces. Get everyone on your team to agree on a setting in your Editor preferences without igniting a tabs versus spaces flame war. Note that Visual Studio provides the option to convert tabs to spaces.
决定统一的缩进: 通常是四个或两个空格。在不引发制表符与空格的战争的情况下，让你的团队中的每个人都同意在你的编辑器首选项中设置一个选项。注意，Visual Studio提供了将制表符转换为空格的选项。

In Visual Studio (Windows), navigate to **Tools > Options > Text Editor > C# > Tabs**.
在Visual Studio(Windows)中，导航到**Tools > Options > Text Editor > C# > Tabs**。



Indenting

None

Block

[✓ Smart]

Tab

Tab size: 4

Indent size: 4

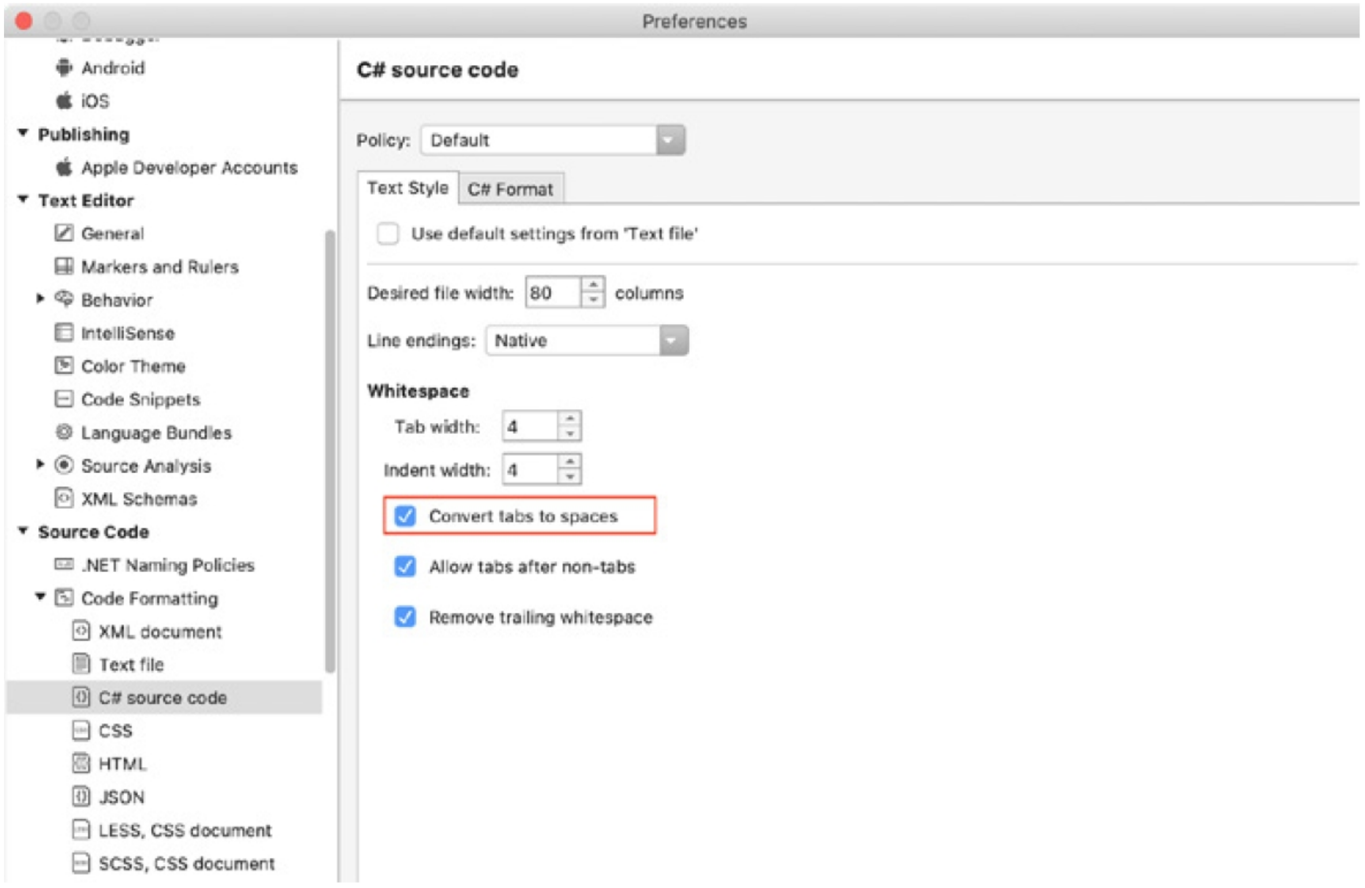
[✓ Insert spaces]

Keep tabs

Adaptive formatting may override your custom tab settings. You can turn off adaptive formatting under Text Editor > Advanced.

Tabs settings in Visual Studio
Visual Studio中的制表符设置

On Visual Studio for Mac, navigate to **Preferences > Source Code > C# Source Code**. Select the Text Style to adjust the settings.
在Visual Studio for Mac上，导航到**Preferences > Source Code > C# Source Code**。选择文本样式来调整设置。



C# source code

Policy: Default

[ ] Use default settings from 'Text file'

Desired file width: 80 columns

Line endings: Native

Whitespace

Tab width: 4

Indent width: 4

[✓] Convert tabs to spaces

[✓] Allow tabs after non-tabs

[✓] Remove trailing whitespace

Convert tabs to spaces to make indentation uniform.
将制表符转换为空格，使缩进统一。

— **Where possible, don't omit braces, even for single-line statements:** This increases consistency, keeping your code easier to read and maintain. In this example, the braces clearly separate the action, DoSomething, from
the loop.
在可能的情况下，不要省略大括号，即使是单行语句: 这增加了一致性，使你的代码更容易阅读和维护。在这个例子中，大括号清楚地将动作DoSomething与循环分开。

If later you need to add a Debug line or to run DoSomethingElse, the braces will already be in place. Keeping the clause on a separate line allows you to add a breakpoint easily.
如果以后你需要添加一个Debug行或运行DoSomethingElse，大括号将已经就位。将子句保持在单独的一行上可以让你轻松地添加断点。

```
// EXAMPLE: keep braces for clarity...
for (int i = 0; i < 100; i++) { DoSomething(i); }

// … and/or keep the clause on a separate line.
for (int i = 0; i < 100; i++)
{
    DoSomething(i);
}
```

```
// AVOID: omitting braces
for (int i = 0; i < 100; i++) DoSomething(i);
```

— **Don’t remove braces from nested multi-line statements:** Removing braces in this case won’t throw an error, but can be confusing. Apply braces for clarity, even if they are optional.
不要从嵌套的多行语句中删除大括号: 在这种情况下删除大括号不会抛出错误，但可能会令人困惑。即使可选，也要应用大括号来保持清晰。

```
// EXAMPLE: keep braces for clarity
for (int i = 0; i < 10; i++)
{
    for (int j = 0; j < 10; j++)
    {
        ExampleAction();
    }
}

// AVOID: removing braces from nested multi-line statements
for (int i = 0; i < 10; i++)
    for (int j = 0; j < 10; j++)
      ExampleAction();
```

— **Standardize your switch statements:** Formatting can vary, so document your team preference in your style guide. Here is one example where you indent the case statements.
标准化你的switch语句: 格式可以有所不同，所以在你的样式指南中记录你的团队偏好。这是一个例子，你可以缩进case语句。

```
// EXAMPLE: indent cases from the switch statement
switch (someExpression)
{
    case 0:
        DoSomething();
        break;
    case 1:
        DoSomethingElse();
        break;
    case 2:
        int n = 1;
        DoAnotherThing(n);
        break;
}
```

> ## What is EditorConfig? 什么是EditorConfig?
>
> Do you have multiple developers working on the same project with different editors and IDEs? Consider using an EditorConfig file.
> 你是否有多个开发人员在同一个项目上使用不同的编辑器和IDE?考虑使用EditorConfig文件。
>
> The EditorConfig file can help you define a coding style that works across your entire team. Many IDEs, like Visual Studio and Rider, come bundled with native support and do not require a separate plugin.
> EditorConfig 文件可以帮助你定义一个在整个团队中都有效的编码风格。许多IDE，如Visual Studio和Rider，都内置了本地支持，不需要单独的插件。
>
> EditorConfig files are easily readable and work with version control systems. You can see an example file here. The code styling from EditorConfig travels with your code and can enforce coding styles even outside of Visual Studio.
> EditorConfig文件易于阅读，并且可以与版本控制系统一起使用。你可以在这里看到一个示例文件。来自EditorConfig的代码样式与你的代码一起传播，甚至可以在Visual Studio之外强制执行代码样式。
>
> EditorConfig settings take precedence over the global Visual Studio text editor settings. Your personal editor preferences still apply whenever you’re working in a codebase without a .editorconfig file, or when the .editorconfig file doesn’t override a particular setting.
> EditorConfig设置优先于全局Visual Studio文本编辑器设置。只要你在没有.editorconfig文件的代码库中工作，或者.editorconfig文件没有覆盖特定的设置，你的个人编辑器首选项仍然适用。
>
> See the GitHub repo for some real-world samples.
> 请参阅GitHub repo中的一些真实样本。

orizontal spacing 水平间距

Something as simple as spacing can enhance your code’s appearance onscreen. Your personal formatting preferences can vary, but try the following suggestions to improve readability:
简单的间距可以增强你的代码在屏幕上的外观。你个人的格式化偏好可能会有所不同，但是尝试以下建议来提高可读性:

— **Add spaces to decrease code density:** The extra whitespace can give a sense of visual separation between parts of a line.
添加空格来减少代码密度: 额外的空白可以在一行的各个部分之间产生视觉分离的感觉。

```
// EXAMPLE: add spaces to make lines easier to read
for (int i = 0; i < 100; i++) { DoSomething(i); }

// AVOID: no spaces
for(inti=0;i<100;i++){DoSomething(i);}
```

— Use a single space after a comma between function arguments.
在函数参数之间的逗号后使用一个空格。

```
// EXAMPLE: single space after comma between arguments
CollectItem(myObject, 0, 1);

// AVOID:
CollectItem(myObject,0,1);
```

— Don’t add a space after the parenthesis and function arguments.
在括号和函数参数之间不要添加空格。

```
// EXAMPLE: no space after the parenthesis and function arguments
DropPowerUp(myPrefab, 0, 1);

//AVOID:
DropPowerUp( myPrefab, 0, 1 );
```

— Don’t use spaces between a function name and parenthesis.
不要在函数名和括号之间使用空格。

```
// EXAMPLE: omit spaces between a function name and parenthesis.
DoSomething()

// AVOID
DoSomething ()
```

— Avoid spaces inside brackets.
避免在括号内使用空格。

```
// EXAMPLE: omit spaces inside brackets
x = dataArray[index];

// AVOID
x = dataArray[ index ];
```

— **Use a single space before flow control conditions:** Add a space between the flow comparison operator and the parentheses.
在流控制条件之前使用一个空格: 在流比较运算符和括号之间添加一个空格。

```
// EXAMPLE: space before condition; separate parentheses with a
space.
while (x == y)

// AVOID
while(x==y)
```

— **Use a single space before and after comparison operators.**
在比较运算符之前和之后使用一个空格。

```
/ EXAMPLE: space before condition; separate parentheses
with a space.
if (x == y)

// AVOID
if (x==y)
```

— **Keep lines short. Consider horizontal whitespace:** Decide on a standard line width (80–120 characters). Break a long line into smaller statements rather than letting it overflow.
保持行短。考虑水平空白: 决定一个标准的行宽(80-120个字符)。将一行长语句分成几个较小的语句，而不是让它溢出。

— **Maintain indentation/hierarchy:** Indent your code to increase legibility.
保持缩进/层次结构: 缩进你的代码以增加可读性。

— **Don't use column alignment unless needed for readability:** This type of spacing aligns the variables but can make it difficult to pair the type with the name.
除非需要可读性，否则不要使用列对齐: 这种类型的间距对齐变量，但可能会使类型与名称配对变得困难。

Column alignment, however, can be useful for bitwise expressions or structs with a lot of data. Just be aware that it may create more work for you to maintain the column alignment as you add more items. Some autoformatters
might also change which part of the column gets aligned.
然而，列对齐对于位表达式或具有大量数据的结构可能是有用的。只要注意，随着你添加更多的项目，它可能会为你创建更多的工作来维护列对齐。一些自动格式化程序也可能会改变列对齐的哪一部分。

```
// EXAMPLE: One space between type and name
public float Speed = 12f;
public float Gravity = -10f;
public float JumpHeight = 2f;

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

// AVOID: column alignment

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

Vertical spacing 垂直间距

You can use the vertical spacing to your advantage as well. Keep related parts of the script together and use blank lines to your advantage. Try these suggestions to organize your code from top to bottom:
你也可以利用垂直间距。将脚本的相关部分放在一起，并利用空白行。尝试以下建议，从上到下组织你的代码:

— **Group dependent and/or similar methods together:** Code needs to be logical and coherent. Keep methods that do the same thing next to one another, so someone reading your logic doesn't have to jump around the file.
将依赖和/或相似的方法分组在一起: 代码需要逻辑和连贯。将做同样事情的方法放在一起，这样阅读你的逻辑的人就不必在文件中跳来跳去。

— **Use the vertical whitespace to your advantage to separate distinct parts of your class:** For example, you can add two blank lines between:
利用垂直空白来分隔类的不同部分: 例如，你可以在以下两个空行之间添加两个空行:

— Variable declarations and methods
变量声明和方法

— Classes and Interfaces
类和接口

— if-then-else blocks (if it helps readability)
if-then-else块(如果它有助于可读性)

Keep this to a minimum and note on your style guide where applicable.
将其最小化，并在适用的地方在你的样式指南中注明。

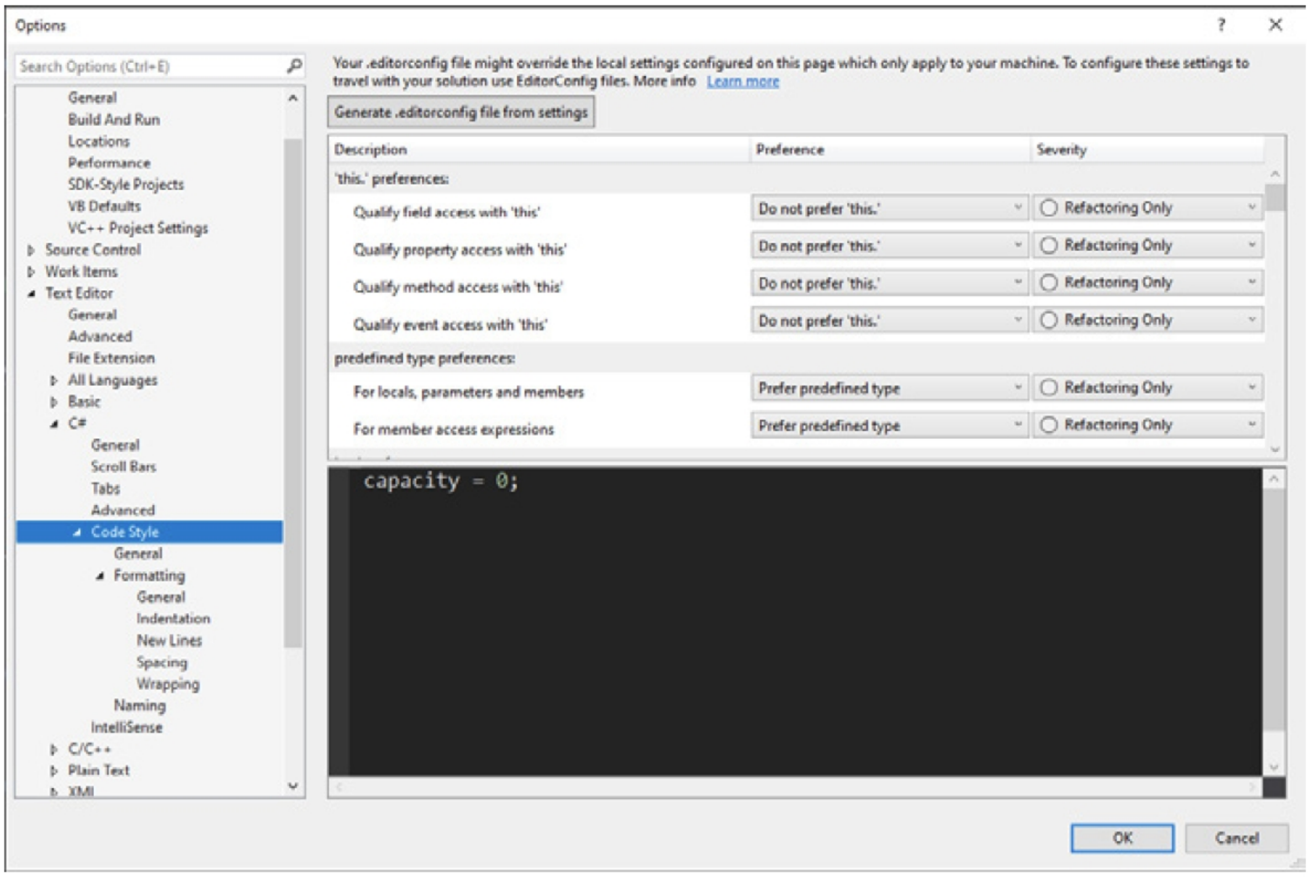
Regions 区域

The `#region` directive enables you to collapse and hide sections of code in C# files, making large files more manageable and easier to read.
`#region指令使你能够折叠和隐藏C` `#文件中的代码部分，使大文件更易于管理和阅读。`

However, if you follow the general advice for Classes from this guide, your class size should be manageable and the `#region` directive superfluous. Break your code into smaller classes instead of hiding code blocks behind regions. You will be less inclined to add a region if the source file is short.
然而，如果你遵循本指南中类的一般建议，你的类大小应该是可管理的，#region指令是多余的。将你的代码分成较小的类，而不是将代码块隐藏在区域后面。如果源文件很短，你就不太可能添加一个区域。

> **Note:** Many developers consider regions to be code smells or anti-patterns. Decide as a team on which side of the debate you fall.
> **注意:** 许多开发人员认为区域是代码气味或反模式。作为一个团队决定你在辩论的哪一边。

> ### Code formatting in Visual Studio for Mac 在Visual Studio for Mac中的代码格式化
>
> Don't despair if these formatting rules seem overwhelming. Modern IDEs make it efficient to set up and enforce them. You can create a template of formatting rules and then convert your project files at once.
> 如果这些格式化规则看起来令人不知所措，不要绝望。现代IDE使得设置和强制执行它们变得高效。你可以创建一个格式化规则的模板，然后一次转换你的项目文件。
>
> To set up formatting rules for the script editor:
> 要为脚本编辑器设置格式化规则:
>
> — In Visual Studio (Windows), navigate to **Tools > Options**. Locate **Text Editor > C# > Code Style Formatting**.
> 在Visual Studio(Windows)中，导航到**Tools > Options**。找到**Text Editor > C# > Code Style Formatting**。
>
> Use the settings to modify the General, Indentation, New Lines, Spacing, and Wrapping options.
> 使用这些设置来修改General、Indentation、New Lines、Spacing和Wrapping选项。



Your .editorconfig file might override the local settings configured on this page which only apply to your machine. To configure these settings to travel with your solution use EditorConfig files. More info Learn more

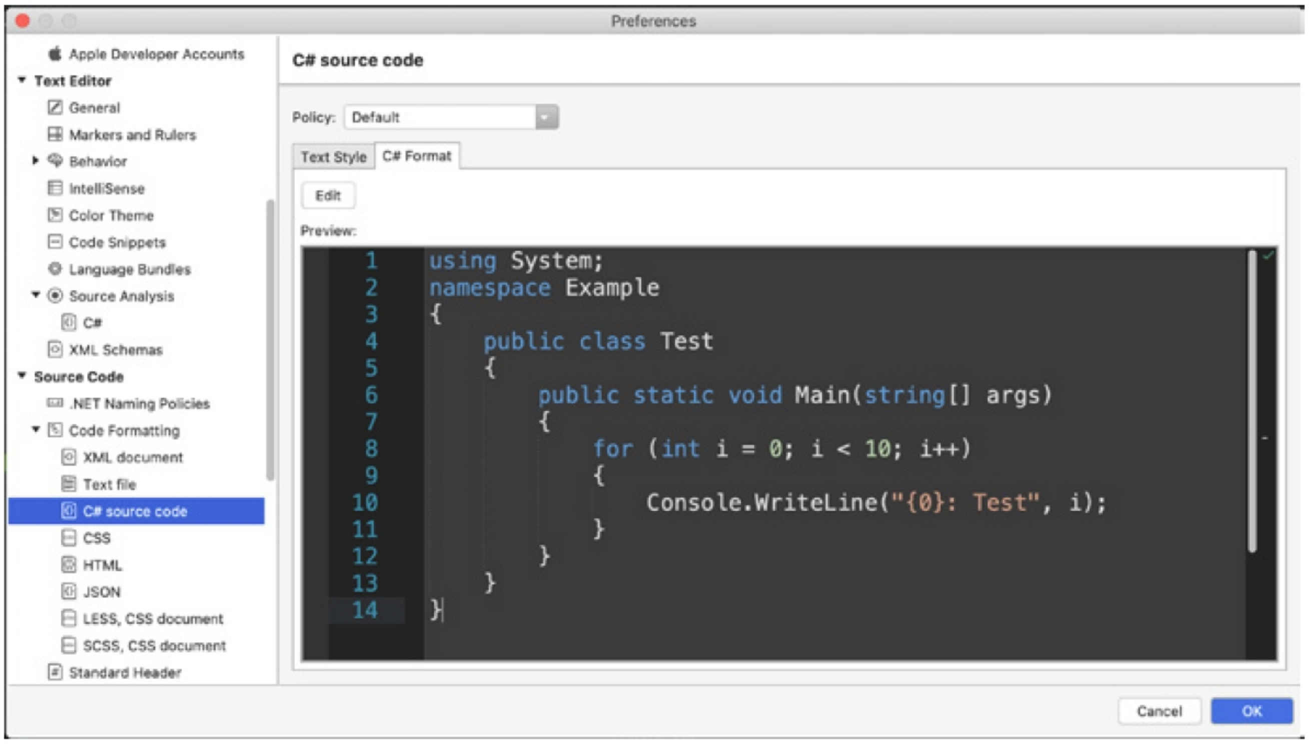
Description	Preference	Severity
'this.' preferences:		
Qualify field access with 'this'	Do not prefer 'this.'	Refactoring Only
Qualify property access with 'this'	Do not prefer 'this.'	Refactoring Only
Qualify method access with 'this'	Do not prefer 'this.'	Refactoring Only
Qualify event access with 'this'	Do not prefer 'this.'	Refactoring Only
predefined type preferences:		
For locals, parameters and members	Prefer predefined type	Refactoring Only
For member access expressions	Prefer predefined type	Refactoring Only

```
capacity = 0;
```

Code style formatting options 代码样式格式选项

— In Visual Studio for Mac, select **Visual Studio > Preferences, then navigate to Source Code > Code Formatting > C# source code**.
在Visual Studio for Mac中，选择**Visual Studio > Preferences**，然后导航到**Source Code > Code Formatting > C# source code。**

Select the Policy at the top. Then set your spacing and indentation in the Text Style tab. In the C# Format tab, adjust the Indentation, New Lines, Spacing, and Wrapping settings.
在顶部选择Policy。然后在Text Style选项卡中设置你的间距和缩进。在C# Format选项卡中，调整Indentation、New Lines、Spacing和Wrapping设置。



Policy: Default

```
using System;
namespace Example
{
    public class Test
    {
        public static void Main(string[] args)
        {
            for (int i = 0; i < 10; i++)
            {
                Console.WriteLine("{0}: Test", i);
            }
        }
    }
}
```

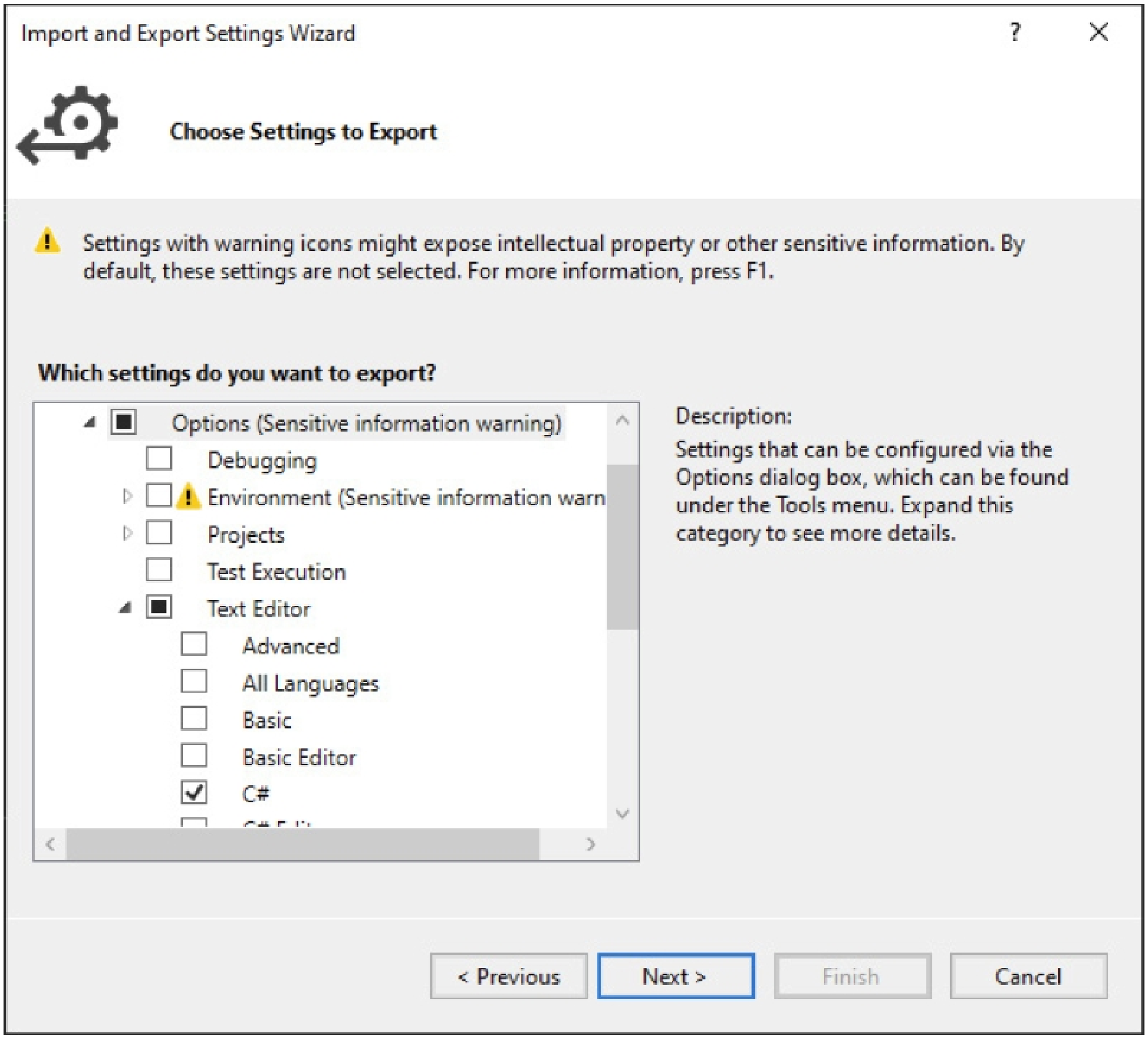
The Preview window shows off your style guide choices. 预览窗口展示了你的样式指南选择。

If at any time you want to force your script file to conform to the style guide:
如果你想随时强制你的脚本文件符合样式指南:

— In Visual Studio (Windows), go to **Edit > Advanced > Format Document (Ctrl + K, Ctrl + D** hotkey chord). If you want only to format white spaces and tab alignment, you can also use Run Code Cleanup (**Ctrl + K , Ctrl + E**) at the bottom of the editor.
在Visual Studio(Windows)中，转到**Edit > Advanced > Format Document(Ctrl + K, Ctrl + D**热键和弦)。如果你只想格式化空格和制表符对齐，你也可以在编辑器底部使用Run Code Cleanup(**Ctrl + K , Ctrl + E**)。

— In Visual Studio for Mac, go to **Edit > Format Document (Ctrl + I** hotkey)
在Visual Studio for Mac中，转到**Edit > Format Document (Ctrl + I**

On Windows, you can also share your editor settings from **Tools > Import and Export Settings**. Export a file with the style guide's C# code formatting and then have every team member import that file.
在Windows上，你还可以从**Tools > Import and Export Settings**共享你的编辑器设置。导出一个带有样式指南的C#代码格式化的文件，然后让每个团队成员导入该文件。


Import and Export Settings Wizard

Choose Settings to Export

Settings with warning icons might expose intellectual property or other sensitive information. By default, these settings are not selected. For more information, press F1.

Which settings do you want to export?

- [■] Options (Sensitive information warning)
  - [ ] Debugging
  - [ ] Environment (Sensitive information warn
  - [ ] Projects
  - [ ] Test Execution
  - [■] Text Editor
    - [ ] Advanced
    - [ ] All Languages
    - [ ] Basic
    - [ ] Basic Editor
    - [✓] C#

Description:
Settings that can be configured via the Options dialog box, which can be found under the Tools menu. Expand this category to see more details.

Exporting the C# code formatting to share 导出C#代码格式以共享

Visual Studio makes it easy to follow the style guide. Formatting then becomes as simple as using a hotkey.
Visual Studio使得遵循样式指南变得很容易。然后，格式化就像使用热键一样简单。

Note: You can configure an EditorConfig file (see above) instead of importing and exporting Visual Studio settings. Doing this allows you to share formatting more easily across different IDEs, and it has the added benefit of working with version control. See the .NET code style rule options for more information.
注意: 你可以配置一个EditorConfig文件(见上文)，而不是导入和导出Visual Studio设置。这样做可以更轻松地在不同的IDE之间共享格式，而且它还有一个额外的好处，就是可以与版本控制一起使用。有关更多信息，请参阅.NET代码样式规则选项。

Though this isn't specific to clean code, be sure to check out 10 ways to speed up your programming workflow in Unity with Visual Studio. Clean code is much easier to format and

> refactor if you apply these productivity tips.
> 虽然这不是特定于清洁代码的，但一定要查看10 ways to speed up your programming workflow in Unity with Visual Studio。如果你应用这些生产力技巧，清洁代码就更容易格式化和重构。

CLASSSES 类

"NO ONE IN THE BRIEF HISTORY OF COMPUTING HAS EVER WRITTEN A PIECE OF PERFECT SOFTWARE. IT’S UNLIKELY THAT YOU’LL BE THE FIRST.” – Andy Hunt, author of The Pragmatic Programmer
“在计算机的简短历史上，没有人写过一段完美的软件。你不太可能是第一个。”——《The Pragmatic Programmer》的作者Andy Hunt

According to Robert C. Martin’s Clean Code, the first rule of classes is that they should be small. The second rule is they should be even smaller than that.
根据Robert C. Martin的《Clean Code》，类的第一条规则是它们应该很小。第二条规则是它们应该比这还要小。

Limiting the size of each class makes it more focused and cohesive. It’s easy to keep adding on top of an existing class until it overextends with functionality. Instead make a conscious effort to keep the classes short. Big, bloated classes become difficult to read and troubleshoot.
限制每个类的大小使其更加专注和内聚。很容易在现有类的基础上添加功能，直到它过度扩展。相反，要有意识地保持类的简短。大而臃肿的类变得难以阅读和排除故障。

The newspaper metaphor 报纸的隐喻

Imagine the source code of a class as a news article. You start reading from the top, where the headline and byline catch your eye. The lead-in paragraph gives you a rough summary, then you glean more details as you continue downward.
想象一下一个类的源代码就像一篇新闻文章。你从头开始阅读，标题和署名吸引了你的注意。引语段给你一个粗略的摘要，然后你在继续向下阅读时获得更多的细节。

Journalists call this the inverted pyramid. The broad strokes of most newsworthy items appear at the beginning. You only get the story’s nuances as you read to the end.
记者称之为倒金字塔。大多数新闻事件的大致轮廓出现在开头。只有当你读到最后，你才能得到这个故事的细微之处。

Your class should also follow this basic pattern. Organize top-down and think of your functions as forming a hierarchy. Some methods serve a higher-level and lay the groundwork for the big picture. Put these first, then, place lower-level functions with implementation details later.
你的类也应该遵循这个基本模式。自上而下地组织，把你的函数看作是一个层次结构。一些方法服务于更高层次，为大局奠定基础。首先放置这些，然后，稍后放置具有实现细节的低级函数。

For example, you might make a method called ThrowBall that references other methods, SetInitialVelocity and CalculateTrajectory. Keep ThrowBall first, since that describes the main action. Then, add the supporting methods below it.
例如，你可以制作一个名为ThrowBall的方法，该方法引用其他方法，SetInitialVelocity和CalculateTrajectory。保持ThrowBall在前，因为它描述了主要动作。然后，在它下面添加支持方法。

Though each news article is short, a newspaper or news website will have many such collected stories. When taken together, the articles comprise a unified, functional whole. Think of your Unity project in the same way. It has numerous classes that must come together to form a larger, yet coherent, application.
虽然每篇新闻文章都很短，但报纸或新闻网站会有许多这样的文章。当这些文章放在一起时，它们构成了一个统一的、功能完整的整体。以同样的方式来思考你的Unity项目。它有许多类必须结合在一起，形成一个更大的、但连贯的应用程序。

Class organization 类组织

Each class will need some standardization. Group class members into sections to organize them:
每个类都需要一些标准化。将类成员分组到部分中以组织它们:

— Fields 字段
— Properties 属性
— Events / Delegates 事件/委托
— Monobehaviour Methods (Awake, Start, OnEnable, OnDisable, OnDestroy, etc.)
Monobehaviour方法(Awake, Start, OnEnable, OnDisable, OnDestroy,等等)
— Public Methods 公共方法
— Private Methods 私有方法

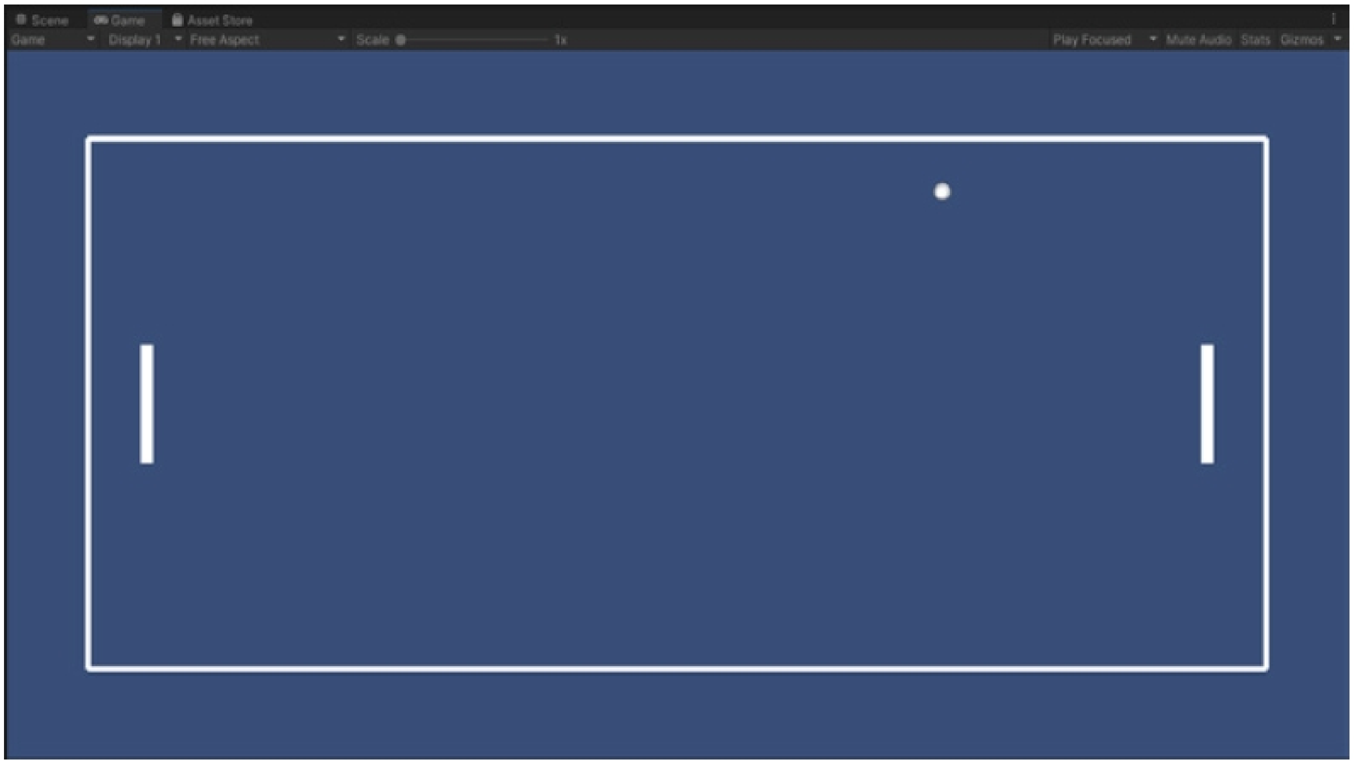
Recall the recommended class naming rules in Unity: The source file name must match the name of the Monobehaviour in the file. You might have other internal classes in the file, but only one Monobehaviour should exist per file.
回想一下Unity中推荐的类命名规则:源文件名必须与文件中Monobehaviour的名字匹配。你可能在文件中有其他内部类，但每个文件只能有一个Monobehaviour。

Single-responsibility principle 单一职责原则

Remember the goal is to keep each class short. In software design, the singleresponsibility principle guides you toward simplicity.
记住目标是保持每个类的长度短。在软件设计中，单一职责原则指导你走向简单。

The idea is that each module, class, or function is responsible for one thing. Suppose you want to build a game of Pong. You might start with classes for a paddle, a ball, and a wall.

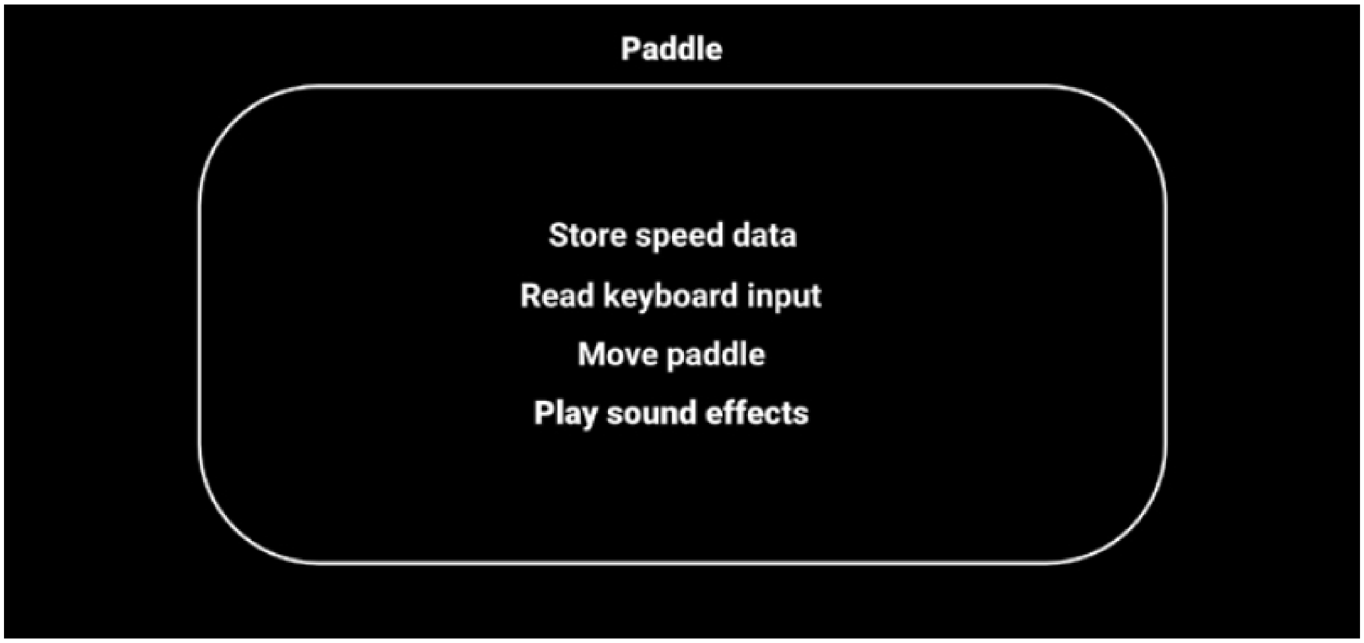
这个想法是每个模块、类或函数都负责一件事。假设你想建一个乒乓球游戏。你可以从球拍、球和墙的类开始。



Fancy a game of Pong? 想玩乒乓球吗?

For example, a Paddle class might need to:
例如，一个Paddle类可能需要:
—Store basic data about how fast it can move 存储关于它可以移动的速度的基本数据
— Check keyboard input 检查键盘输入
— Move the paddle in response 响应移动挡板
— Play a sound when colliding with a ball 碰撞球时播放声音

Because the game design is simple, you can incorporate all of these things into a basic Paddle class. In fact, it’s entirely possible to create one Monobehaviour that does everything you need.
因为游戏设计很简单，你可以将所有这些东西都纳入一个基本的Paddle类。事实上，完全有可能创建一个Monobehaviour来做你需要的一切。

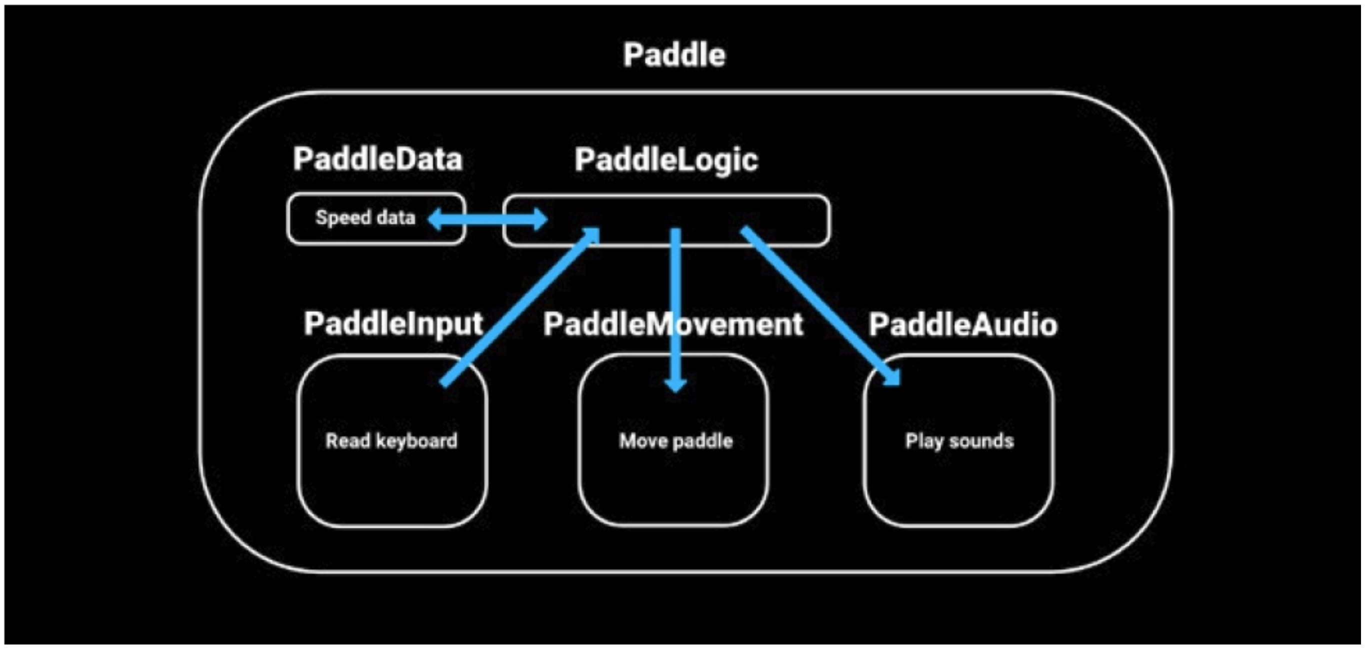


One Monobehaviour doing everything 一个Monobehaviour做一切

However, keeping everything as part of one class, even a small one, complicates the design by mixing responsibilities. The data intertwines with the input, while the class needs to apply logic to both. Contrary to the KISS principle, you've taken a few simple things and entangled them.
然而，将所有东西都作为一个类的一部分，即使是一个小类，也会通过混合职责来复杂化设计。数据与输入交织在一起，而类需要对两者都应用逻辑。与KISS原则相反，你已经把一些简单的东西纠缠在一起了。

Instead, break your Paddle class into smaller classes, each with a single responsibility. Separate data into its own PaddleData class or use a ScriptableObject. Then refactor everything else into a PaddleInput class, a PaddleMovement class, and a PaddleAudio class.
相反，将你的Paddle类分成更小的类，每个类都有一个单一的职责。将数据分离到它自己的PaddleData类中，或者使用一个ScriptableObject。然后将其他所有内容重构为PaddleInput类、PaddleMovement类和PaddleAudio类。

A PaddleLogic class can process the input from the PaddleInput. Applying the speed information from the PaddleData, it can shift the paddle using the PaddleMovement. Finally, the PaddleLogic can notify the PaddleAudio to play a sound when the ball collides with the paddle.
PaddleLogic类可以处理来自PaddleInput的输入。应用来自PaddleData的速度信息，它可以使用PaddleMovement移动挡板。最后，当球与球拍碰撞时，PaddleLogic可以通知PaddleAudio播放声音。



Refactor a Paddle class into single responsibilities 重构一个Paddle类到单一职责

Each class does one thing in this redesign and fits into small, digestible pieces. You don't need to scroll through several screens to follow the code.
在这个重新设计中，每个类只做一件事，适合小而易于消化的东西。你不需要滚动几个屏幕来跟踪代码。

You'll still require a Paddle script but its sole job is to tie these other classes together. The bulk of the functionality is split into the other classes.
你仍然需要一个Paddle脚本，但它的唯一工作是将这些其他类联系在一起。大部分功能都分成了其他类。

Note that clean code is not always the most compact code. Even when you use shorter classes, the total number of lines may increase during refactoring. However, each individual class becomes easier to read. When the time comes to debug or add new features, this simplified structure helps keep everything in its place.
注意，清洁代码并不总是最紧凑的代码。即使你使用更短的类，在重构过程中，总行数也可能会增加。然而，每个单独的类变得更容易阅读。当时机成熟时，调试或添加新功能，这种简化的结构有助于保持一切井然有序。

Refactoring example 重构示例

For a more in-depth look at refactoring a simple project, see How to architect code as your project scales. This article demonstrates how to break down larger Monobehaviours into smaller pieces using the single-responsibility principle.
有关重构一个简单项目的更深入的了解，请参阅How to architect code as your project scales。本文演示了如何使用单一职责原则将较大的Monobehaviour分解成较小的部分。

> You can also watch Mikael Kalms's original presentation, "From Pong to 15-person project," from Unite Berlin.

METHODS 方法

"YOU KNOW YOU ARE WORKING ON CLEAN CODE WHEN EACH ROUTINE YOU READ TURNS OUT TO BE PRETTY MUCH WHAT YOU EXPECTED." – Ward Cunningham, inventor of Wiki and cofounder of eXtreme Programming
"当你读到的每一个例程都变成了你所期望的东西时，你就知道你正在处理干净的代码。"——Ward Cunningham，Wiki的发明者和eXtreme Programming的联合创始人

Like classes, methods should be small with a single responsibility. Each method should describe one action or answer one question. It shouldn't do both.
像类一样，方法应该是小的，只有一个职责。每个方法应该描述一个动作或回答一个问题。它不应该两者都做。

A good name for a method reflects what it does. For example, GetDistanceToTarget is a name that clarifies its intended purpose.
一个好的方法名反映了它的作用。例如，GetDistanceToTarget是一个可以澄清其预期目的的名字。

Try the following suggestions when you create methods for your custom classes:
当你为你的自定义类创建方法时，请尝试以下建议:

— **Use fewer arguments**: Arguments can increase the complexity of your method. Reduce their number to make your methods easier to read and test.
使用更少的参数: 参数可以增加你的方法的复杂性。减少它们的数量，使你的方法更容易阅读和测试。

— **Avoid excessive overloading**: You can generate an endless permutation of method overloads. Select the few that reflect how you will call the method and implement those. If you do overload a method, prevent confusion by making sure each method signature has a distinct number of arguments.
避免过度重载: 你可以生成一个无限的方法重载排列。选择反映你将如何调用方法的少数方法，并实现这些方法。如果你确实重载了一个方法，请确保每个方法签名都有一个不同数量的参数，以防止混淆。

— **Avoid side effects**: A method only needs to do what its name advertises. Avoid modifying anything outside of its scope. Pass in arguments by value instead of by reference when possible. If sending back results via the out or ref keyword, make sure that's the one thing you intend the method to accomplish.
避免副作用: 一个方法只需要做它的名字所说的。避免修改其范围之外的任何东西。如果可能的

话，通过值而不是通过引用传递参数。如果通过out或ref关键字发送结果，请确保这是你打算完成的唯一事情。

Though side effects are useful for certain tasks, they can lead to unintended consequences. Write a method without side effects to cut down on unexpected behavior.
虽然副作用对于某些任务是有用的，但它们可能会导致意想不到的后果。编写一个没有副作用的方法来减少意外行为。

— **Instead of passing in a flag, make another method**: Don't set up your method to work in two different modes based on a flag. Make two methods with distinct names. For example, don't make a GetAngle method that returns degrees or radians based on a flag setting. Instead make methods for GetAngleInDegrees and GetAngleInRadians.
不要传递一个标志，而是创建另一个方法: 不要根据标志设置使你的方法在两种不同的模式下工作。使用不同的名称创建两个方法。例如，不要创建一个GetAngle方法，它根据标志设置返回度或弧度。而是为GetAngleInDegrees和GetAngleInRadians创建方法。

While the Boolean flag as an argument seems innocuous, it can lead to tangled implementation or broken single-responsibility.
虽然布尔标志作为参数似乎是无害的，但它可能会导致纠缠的实现或破坏单一职责。

Extension methods 扩展方法

Extension methods offer a way to add additional functionality to classes that might otherwise be sealed and can be a clean way to extend the UnityEngine API.
扩展方法提供了一种方法来为可能被封装的类添加额外的功能，并且可以是扩展UnityEngine API的一种干净的方法。

To create an extension method, make a static method and use the this keyword before the first argument, which will be the type you want to extend.
要创建一个扩展方法，创建一个静态方法，并在第一个参数之前使用this关键字，这将是你想要扩展的类型。

For example, suppose you want to make a method called ResetTransformation to remove any scaling, rotation, or translation from a GameObject.
You can create a static method passing in a Transform for the first argument with the this keyword:
例如，假设你想创建一个名为ResetTransformation的方法，以从GameObject中删除任何缩放、旋转或平移。

```
// EXAMPLE: Define an extension method
public static class TransformExtensions
{
    public static void ResetTransformation(this Transform transform)
    {
        transform.position = Vector3.zero;
        transform.localRotation = Quaternion.identity;
        transform.localScale = Vector3.one;
    }
}
```

Then, when you want to use it, invoke the ResetTransformation method. The ResetOnStart class calls it on the current Transform during Start.
然后，当你想使用它时，调用ResetTransformation方法。ResetOnStart类在Start期间调用它的当前Transform。

```
// EXAMPLE: Calling the extension method

public class ResetOnStart : MonoBehaviour
{
    void Start()
     {
        transform.ResetTransformation();
     }
}
```

For organization purposes, define your extension methods in a static class. For example, you create a class called TransformExtensions for methods that extend Transforms, Vector3Extensions for extending Vector3s, and so on.
出于组织目的，在一个静态类中定义你的扩展方法。例如，你创建一个名为TransformExtensions的类，用于扩展Transforms、Vector3Extensions用于扩展Vector3s，等等。

Extension methods can build many useful utilities without the need to create more Monobehaviours. See Unity Learn: Extension Methods to add them to your gamedev bag of tricks.
扩展方法可以构建许多有用的实用程序，而无需创建更多的Monobehaviours。请参阅Unity Learn: Extension Methods将它们添加到你的gamedev技巧包中。

The DRY principle: Don’t repeat yourself DRY原则:不要重复自己

In The Pragmatic Programmer, Andy Hunt and Dave Thomas formulated the DRY principle, or, “don’t repeat yourself.” This oft-spoken mantra in software engineering advises programmers to avoid duplicate or repetitious logic.
在《The Pragmatic Programmer》中，Andy Hunt和Dave Thomas制定了DRY原则，即“不要重复自己”。这个在软件工程中经常被提及的口头禅建议程序员避免重复或重复的逻辑。

In doing so, you can ease bug fixing and maintenance costs. If you follow the single-responsibility principle, you shouldn't need to change an unrelated piece of code whenever you modify a class or a method. Quashing a logical bug in a DRY program stops it everywhere.
这样做可以减轻错误修复和维护成本。如果你遵循单一职责原则，当你修改一个类或一个方法时，你不应该需要改变一个不相关的代码片段。在DRY程序中消除一个逻辑错误会使它在任何地方停止。

The opposite of DRY is WET ("we enjoy typing" or "write everything twice"). Programming is WET when there are unnecessary repetitions in the code.
DRY的反义词是WET("我们喜欢打字"或"写两遍")。当代码中有不必要的重复时，编程是湿的。

Imagine there are two ParticleSystems (explosionA and explosionB) and two AudioClips (soundA and soundB). Each ParticleSystem needs to play with its respective sound, which you can achieve with simple methods like this.
想象一下，有两个ParticleSystem(explosionA和explosionB)和两个AudioClips(soundA和soundB)。每个ParticleSystem都需要与其各自的声音一起播放，你可以用这样简单的方法来实现。

```
// EXAMPLE: 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);
}
```

Here each method takes a Vector3 position to move the ParticleSystem into place for playback. First, stop the particles (in case they are already playing) and play the simulation. The AudioSource's static PlayClipAtPoint method then creates a sound effect at the same location.

> 这里每个方法都需要一个Vector3位置来移动ParticleSystem到播放位置。首先，停止粒子(以防它们已经在播放)，然后播放模拟。AudioSource的静态PlayClipAtPoint方法在同一位置创建一个声音效果。
>
> One method is a cut-and-paste version of the other with a little text replacement. Though this works, you need to make a new method – with duplicate logic – every time you want to create an explosion.
> 一个方法是另一个方法的剪切和粘贴版本，只是有一点文本替换。虽然这样做可以，但每次你想创建一个爆炸时，你都需要创建一个新的方法——有重复逻辑。
>
> Instead, refactor it into one PlayFXWithSound method like this:
> 相反，将它重构为一个PlayFXWithSound方法，如下所示:

```
// EXAMPLE: Refactored

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

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

> Add more ParticleSystems and AudioClips, and you can continue using this same method to play them in concert.
> 添加更多的ParticleSystem和AudioClips，你可以继续使用这个相同的方法来演奏它们。
>
> Note that it’s possible to duplicate code without violating the DRY principle. It’s more important that you don’t duplicate logic.
> 注意，可以在不违反DRY原则的情况下复制代码。更重要的是，你不要重复逻辑。
>
> Here, we’ve extracted the core functionality into the PlayFXWithSound method. If you need to adjust the logic, you only need to change it in one method rather than in both PlayExplosionA and PlayExplosionB.
> 在这里，我们将核心功能提取到PlayFXWithSound方法中。如果你需要调整逻辑，你只需要在一个方法中改变它，而不是在PlayExplosionA和PlayExplosionB中都改变它

COMMENTS 注释

"CODE IS LIKE HUMOR. IF YOU HAVE TO EXPLAIN IT, IT’S BAD.”– Cory House, software architect and author 代码就像幽默。如果你不得不解释它，那就是坏的。——Cory House，软

件架构师和作者

Well-placed comments enhance the readability of your code. Excessive or frivolous comments can have the opposite effect. Like all things, strike a balance when using them.
适当的注释可以增强代码的可读性。过多或轻浮的评论可能会产生相反的效果。像所有的事情一样，在使用它们时要保持平衡.

Most of your code won't need comments if you follow KISS principles and break your code into easy-to-digest logical parts. Well-named variables and functions will explain themselves.
如果你遵循KISS原则并将你的代码分解成易于消化的逻辑部分，那么大多数代码都不需要注释。良好命名的变量和函数将解释它们自己。

Rather than answering "what," useful comments fill in the gaps and tell you "why." Did you make specific decisions that are not immediately obvious? Is there a tricky bit of logic that needs clarification? Useful comments reveal information not gleaned from the code itself.
有用的注释不是回答"什么"，而是填补空白，告诉你"为什么"。你做了一些不明显的具体决定吗?有没有需要澄清的棘手的逻辑?有用的注释揭示了从代码本身中无法获得的信息。

Here are some dos and don'ts for comments:
下面是一些注释的dos和don'ts:

— **Don't add comments to replace bad code**: If you need to add a comment to explain a convoluted tangle of logic, restructure your code to be more obvious. Then you won't need the comment.
不要添加注释来替换坏代码: 如果你需要添加一个注释来解释一个复杂的逻辑纠缠，重构你的代码更明显。然后你就不需要评论了。

— **A properly named class, variable, or method serves in place of a comment**: Is the code self-explanatory? Then reduce noise and skip the comment.
适当命名的类、变量或方法可以代替注释: 代码是不是不言自明的?那么减少噪音，跳过评论。

```
// AVOID: noisy, redundant comments

// the target to shoot
Transform targetToShoot;
```

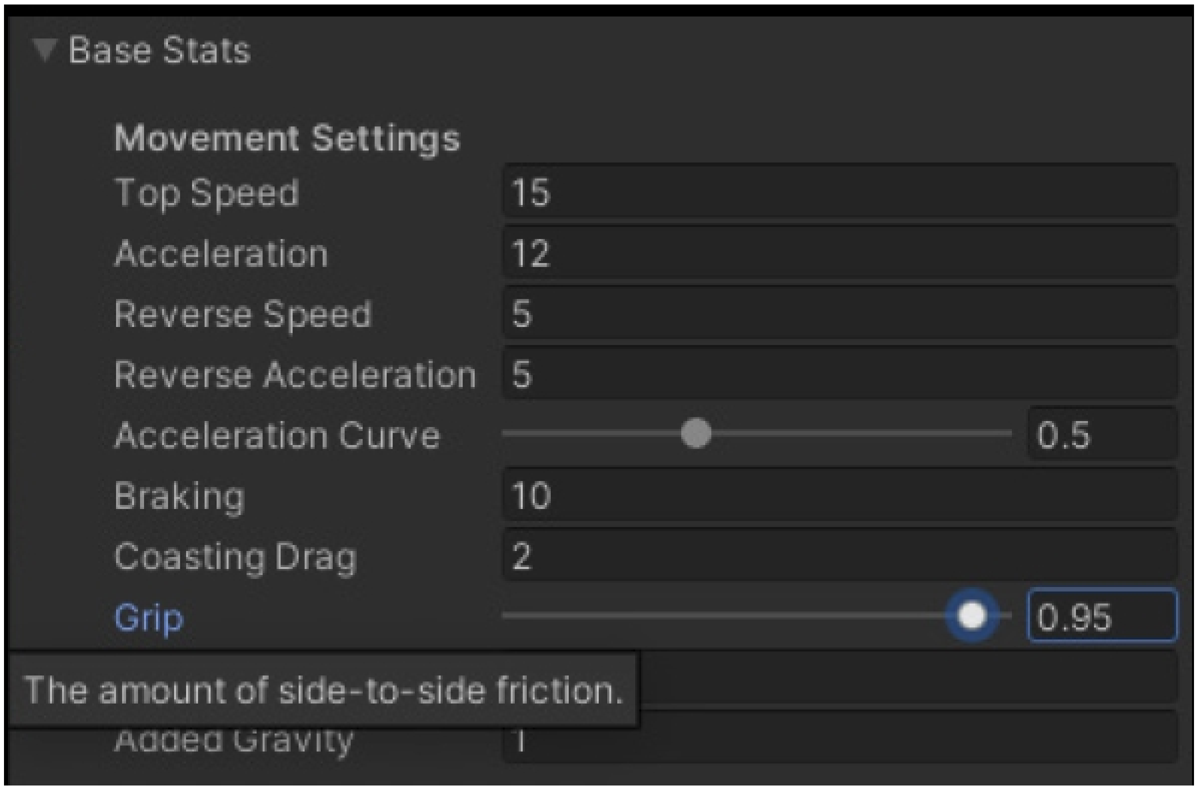
— **Place the comment on a separate line when possible, not at the end of a line of code**: In most cases, keep each one on its own line for clarity.
尽可能将注释放在单独的一行上，而不是在代码行的末尾: 在大多数情况下，为了清晰起见，将每个注释放在自己的一行上。

— **Use the double slash (//) comment tag in most situations**: Keep the comment near the code that it explains rather than using a large multiline at the beginning. Keeping it close helps

the reader connect the explanation with the logic.
在大多数情况下使用双斜杠(//)注释标签: 将注释保持在它解释的代码附近，而不是在开头使用一个大的多行。保持它的接近有助于读者将解释与逻辑联系起来。

— **Use a tooltip instead of a comment for serialized fields**: If your fields in the Inspector need explanation, add a tooltip attribute and skip the separate comment. The tooltip will do double duty.
使用工具提示而不是注释进行序列化字段: 如果你的检查器中的字段需要解释，添加一个工具提示属性，跳过单独的注释。工具提示将承担双重职责。

```
// EXAMPLE: Tooltip replaces comment

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



Tooltip in the Inspector. 工具提示在检查器中。

— **You can also use a summary [XML tag](#) in front of public methods or functions**: Visual Studio can provide IntelliSense for many common XMLstyle comments.
你也可以在公共方法或函数前面使用一个摘要[XML标签](#): Visual Studio可以为许多常见的XML样式注释提供智能感知。

```
// This is a common comment.
// Use them to show intent, logical flow, and approach.
// You can also use a summary XML tag.
```

```
//
/// <summary>
/// Fire the weapon
/// </summary>
public void Fire()
{
    ...
}
```

— **Insert one space between the comment delimiter (//) and the comment text.**
在注释分隔符(//)和注释文本之间插入一个空格。

— **Add legal disclaimers**: A comment is appropriate for the license or copyright information. However, avoid inserting an entire legal brief into your code. Link instead to an external page with the full legal information.
添加法律声明: 注释适用于许可证或版权信息。然而，避免在你的代码中插入整个法律简报。相反，链接到一个包含完整法律信息的外部页面。

— **Style your comments**: Maintain a uniform appearance for your comments, e.g., begin each comment with an uppercase letter and end with a period. Whatever your team decides, make it part of the style guide and follow it.
样式你的评论: 保持你的评论的统一外观，例如，每个评论都以大写字母开头，以句号结尾。无论你的团队决定什么，都要把它作为样式指南的一部分，并遵循它。

— **Don't create formatted blocks of asterisks or special characters around comments**: This reduces readability and contributes to the general malaise of code clutter.
不要在注释周围创建格式化的星号块或特殊字符: 这会降低可读性，并有助于代码混乱的普遍病态。

— **Remove commented out code**: Though commenting out statements may be normal during testing and development, don't leave commented code lying around. Rely on your source control. Then have the courage to delete those two lines of code.
删除注释掉的代码: 虽然在测试和开发过程中注释掉语句可能是正常的，但不要留下注释掉的代码。依靠你的源代码控制。然后有勇气删除这两行代码。

— **Keep your TODO comments up-to-date**: As you complete tasks, make sure you scrub the TODO comments you've left as a reminder. Outdated comments are distractions.
保持你的TODO注释最新: 当你完成任务时，确保你清除你留下的TODO注释作为提醒。过时的评论是分心的。

You can add a name and date to a TODO for more accountability and context.
你可以为TODO添加一个名字和日期，以增加责任和上下文。

Also, be realistic. That TODO you left in the code five years ago? You're never going to get to it. Remember YAGNI. Delete the TODO comment until you need to implement it.
另外，要现实一点。你五年前在代码中留下的TODO?你永远不会去做它。记住YAGNI。删除TODO注释，直到你需要实现它。

— **Avoid journals**: The comments are not a place for your dev diary. There's no need to log everything you're doing in a comment when you start a new class. Proper use of source control makes this redundant.
避免日志: 注释不是你的开发日记的地方。当你开始一个新的类时，在注释中记录你所做的一切是没有必要的。正确使用源代码控制使这个多余。

— **Avoid attributions**: You don't need to add bylines, e.g., // added by devA or devB, especially if you use source control.
避免归因: 你不需要添加署名，例如，// added by devA or devB，特别是如果你使用源代码控制。

COMMON PIFALLLS 常见陷阱

"IF DEBUGGING IS THE PROCESS OF REMOVING SOFTWARE BUGS, THEN PROGRAMMING MUST BE THE PROCESS OF PUTTING THEM IN. — Edsger W. Dijkstra, computer science pioneer
如果调试是消除软件错误的过程，那么编程必须是把它们放进去的过程。——Edsger W. Dijkstra，计算机科学先驱

Clean code isn't an accident. It's the deliberate work of individuals trying to think and code like a team.
干净的代码不是偶然的。它是个人有意识地努力去思考和编码的团队。

Not everything goes to plan, of course. Unclean code inevitably happens, no matter how hard you might try. You'll need to be on the hunt for it.
当然，并不是所有的事情都按计划进行。不管你多么努力，不干净的代码总是会发生的。你需要去寻找它。

A code smell is a telltale sign you might have troublesome code lurking in the project. Though the following symptoms don't necessarily point to underlying problems, they are worth investigating when they appear:
代码气味是一个暗示，你可能有麻烦的代码潜伏在项目中。虽然下面的症状不一定指向潜在的问题，但当它们出现时，它们是值得调查的:

— **Enigmatic naming**: Everyone loves a good mystery, except in their coding standards. Classes, methods, and variables need straightforward, nononsense names.
谜一般的命名: 除了在他们的编码标准中，每个人都喜欢一个好的谜题。类、方法和变量需要直截了当、非常规的名称。

— **Needless complexity**: Over-engineering happens when you try to anticipate every possible need for a class. This can manifest itself as a God object with long methods or large classes that try to do too much. Break up a large class into smaller dedicated parts, each with its own responsibility.
不必要的复杂性: 过度工程化发生在你试图预测一个类的每一个可能的需要时。这可能表现为一个具有长方法的上帝对象，或者试图做太多事情的大类。将一个大类分解成更小的专用部分，每个部分都有自己的职责。

— **Inflexibility**: A small change should not require you to make multiple changes elsewhere. Double-check that you aren't breaking the single responsibility principle if that's the case.
缺乏灵活性: 一个小的改变不应该要求你在其他地方做出多次改变。如果是这样的话，请仔细检查你是否违反了单一职责原则。

When you give something more than one responsibility, it breaks more easily because it's harder to anticipate everything. If you update a method that is doing one thing, and the updated logic still works, you expect the
rest of your code to continue to work afterward.
当你给一个东西超过一个职责时，它更容易被打破，因为它更难以预料到一切。如果你更新一个正在做一件事的方法，更新后的逻辑仍然有效，你希望你的其余代码在此之后继续工作。

— **Fragility**: If you make a minor change and everything stops working, this often indicates a problem.
— **脆弱性**: 如果你做了一个小的改变，一切都停止工作，这通常表明有问题。

— **Immobility**: You'll often write code that is reusable in a different context. If it requires many dependencies to deploy elsewhere, then decouple how the logic works.
— **不动性**: 你经常会写一些在不同上下文中可重用的代码。如果它需要许多依赖项才能在其他地方部署，那么解耦逻辑的工作方式。

— **Duplicate code**: If it's noticeable that you've cut and pasted code, it's time to refactor. Extract the core logic into its own function and call that from the other functions. Copy-and-paste code is difficult to maintain because you need to update the logic in multiple locations each time there is a change.
— **重复代码**: 如果你注意到你剪切和粘贴的代码，那么现在是重构的时候了。将核心逻辑提取到它自己的函数中，并从其他函数中调用它。复制和粘贴代码很难维护，因为每次有变化时，你都需要在多个位置更新逻辑。

— **Excessive commentary**: Comments can help explain code that isn't intuitive. However, developers can overuse them. A running commentary for every variable or statement is unnecessary. Remember that the best comment is a well-named method or class. If you split your logic into smaller pieces, the shorter code snippets require less explanation.
— **过多的评论**: 注释可以帮助解释不直观的代码。然而，开发人员可能会过度使用它们。每个变

量或语句的实时评论是不必要的。记住，最好的评论是一个良好命名的方法或类。如果你将你的逻辑分成更小的片段，较短的代码片段需要较少的解释。

CONCLUSION 结论

"PROGRAMMING IS NOT A ZEROSUM GAME. TEACHING SOMETHING TO A FELLOW PROGRAMMER DOESN'T TAKE IT AWAY FROM YOU."
— John Carmack, cofounder of id Software
“编程不是一个零和游戏。教给一个程序员一些东西并不会把它从你身边带走。”——id Software 的联合创始人John Carmack

We hope you enjoyed this gentle introduction to the principles of clean coding.
我们希望你喜欢这个对清洁编码原则的温和介绍。

The techniques presented here are less a specific set of rules than a set of habits, and like all habits, you'll need to discover them yourself through daily application.
这里介绍的技术不是一套具体的规则，而是一套习惯，就像所有的习惯一样，你需要通过日常应用来发现它们。

As mentioned earlier in the guide, feel free to copy this C# style sheet for Unity developers, to use as a starting point for your own guide.
如前所述，在这个指南中，你可以随意复制这个C#样式表，作为你自己指南的起点。

Prepare your code to be scalable by breaking it into small, modular pieces. As the marathon of development unfolds, expect to rewrite your code over and over again. Production can be a trying process with changing requirements. Fortunately, you won't have to go it alone.
通过将代码分解成小的模块化部分，为你的代码做好可扩展的准备。随着开发的马拉松展开，期望你一遍又一遍地重写你的代码。生产过程可能是一个令人沮丧的过程，因为需求在不断变化。幸运的是，你不必独自面对。

When you code as a group, game development becomes less of a long solo race and more akin to a relay. You have teammates to share the workload with and split up the entire course.
当你作为一个团队编码时，游戏开发就不再是一场漫长的个人比赛，而更像是接力赛。你有队友来分享工作量，并分担整个过程。

Remember to stay in your lane and pass the baton and together, you will make it across the finish line.
记住要留在你的车道上，传递接力棒，一起，你们将跨过终点线。

If you're looking for help on how to clean up your code, reach out to Unity's professional services team, Accelerate Solutions. The team is made up of Unity's most senior software developers. Specializing in performance optimization, development acceleration, game planning, innovation, and much more, Accelerate Solutions offers custom consulting and

development solutions for game studios of all sizes.
如果你正在寻找如何清理你的代码的帮助，请联系Unity的专业服务团队Accelerate Solutions。该团队由Unity最资深的软件开发人员组成。专门从事性能优化、开发加速、游戏规划、创新等工作，Accelerate Solutions为各种规模的游戏工作室提供定制的咨询和开发解决方案。

One of the services offered by Accelerate Solutions is CAP (Code, Assets and Performance). This two-week consulting engagement begins with a three-day deep dive into your code and assets to uncover the root causes of performance issues. This will come with an actionable and detailed report with best practice recommendations. To learn more about this or other services Unity Accelerate Solutions offers, speak to a Unity representative today.
Accelerate Solutions提供的服务之一是CAP(代码、资产和性能)。这个为期两周的咨询项目从为期三天的深入研究你的代码和资产开始，以发现性能问题的根本原因。这将提供一个可行的和详细的报告，其中包含最佳实践建议。要了解更多关于Unity Accelerate Solutions提供的这项服务或其他服务，请今天与Unity代表交谈。

References 参考资料

This guide is a short list of best practices used in computing. For more information, refer to the Microsoft Framework Design Guideline, which serve as an overarching style guide for this document.
本指南是计算机中使用的最佳实践的简短列表。有关更多信息，请参阅Microsoft Framework Design Guideline，它们作为本文档的总体样式指南。

You can also learn more from the comprehensive volumes already written about clean code. Here are a few of our favorite books to consider to further your understanding:
你也可以从已经写过的关于清洁代码的综合卷中学到更多的东西。下面是我们最喜欢的一些书，可以帮助你进一步了解:

Clean Code: A Handbook of Agile Software Craftsmanship. Robert C. Martin, 2008. Prentice Hall. ISBN 978-0132350884.

The Pragmatic Programmer, 20th Anniversary Edition. David Thomas and Andrew Hunt, 2019, Addison Wesley, ISBN 978-0135957059.

APPENDIX: SCRIPTT TEMPLATES 附录:脚本模板

"TALK IS CHEAP. SHOW ME THE CODE." — Linus Torvalds, creator of Linux and Git
“说话是廉价的。给我看代码。”——Linux和Git的创造者Linus Torvalds

Once you establish formatting rules for your style guide, configure your script templates. These templates generate the blank starting files for scripted assets like C# scripts, shaders, or materials.

一旦你为你的样式指南建立了格式规则，就配置你的脚本模板。这些模板生成脚本化资产的空白起始文件，如C#脚本、着色器或材质。

Locate Unity’s preconfigured script templates here:
在这里找到Unity的预配置脚本模板:

Windows: C:\Program Files\Unity\Editor\Data\Resources\ScriptTemplates

Mac: /Applications/Unity/Unity.app/Contents/Resources/ScriptTemplates

On macOS, reveal the Unity.app package contents to show the Resources subdirectory.
在macOS上，显示Unity.app包内容以显示资源子目录。

Inside this path, you’ll see the default templates.
在这个路径里，你会看到默认的模板。

81-C# Script-NewBehaviourScript.cs.txt

82-Javascript-NewBehaviourScript.js.txt

83-Shader__Standard Surface Shader-NewSurfaceShader.shader.txt

84-Shader__Unlit Shader-NewUnlitShader.shader.txt

Whenever you make a new scripted asset in the Project window from the Create menu, Unity uses one of these templates.
每当你从创建菜单中的项目窗口创建一个新的脚本化资产时，Unity就会使用其中一个模板。

If you open the file named 81-C# Script-NewBehaviourScript.cs.txt with a text editor, you will see the following:
如果你用文本编辑器打开名为81-C# Script-NewBehaviourScript.cs.txt的文件，你会看到以下内容:

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

public class #SCRIPTNAME# : MonoBehaviour
{
    // Start is called before the first frame update
    void Start()
    {
        #NOTRIM#
    }

    // Update is called once per frame
    void Update()
    {
        #NOTRIM#
    }
}
```

Note the keywords: 注释关键字:

— #SCRIPTNAME #: This is the name you've specified for the script. If you don't customize the name, it uses the default name, e.g., NewBehaviourScript.
#SCRIPTNAME #: 这是你为脚本指定的名称。如果你不自定义名称，它将使用默认名称，例如，NewBehaviourScript。

— #NOTRIM #:This guarantees whitespace, making sure one line appears between the curly braces.
#NOTRIM #: 这保证了空白，确保花括号之间有一行。

Script templates are customizable. For example, you can add a namespace or remove the default Update method. Modifying the template can save you a few keystrokes every time you create one of these scripted assets.
脚本模板是可定制的。例如，你可以添加一个命名空间或删除默认的Update方法。修改模板可以节省你每次创建这些脚本化资产时的几个按键。

The script template filename follows this pattern:
脚本模板文件名遵循这种模式:

PriorityNumber–MenuPath–DefaultName.FileExtension.txt

A dash (-) character separates the different parts of the name:
破折号(-)字符分隔名称的不同部分:

— **PriorityNumber** is the order that the script appears in, in the Create menu. Lower numbers have higher priority.
— **PriorityNumber**是脚本在创建菜单中出现的顺序。较低的数字优先级较高。

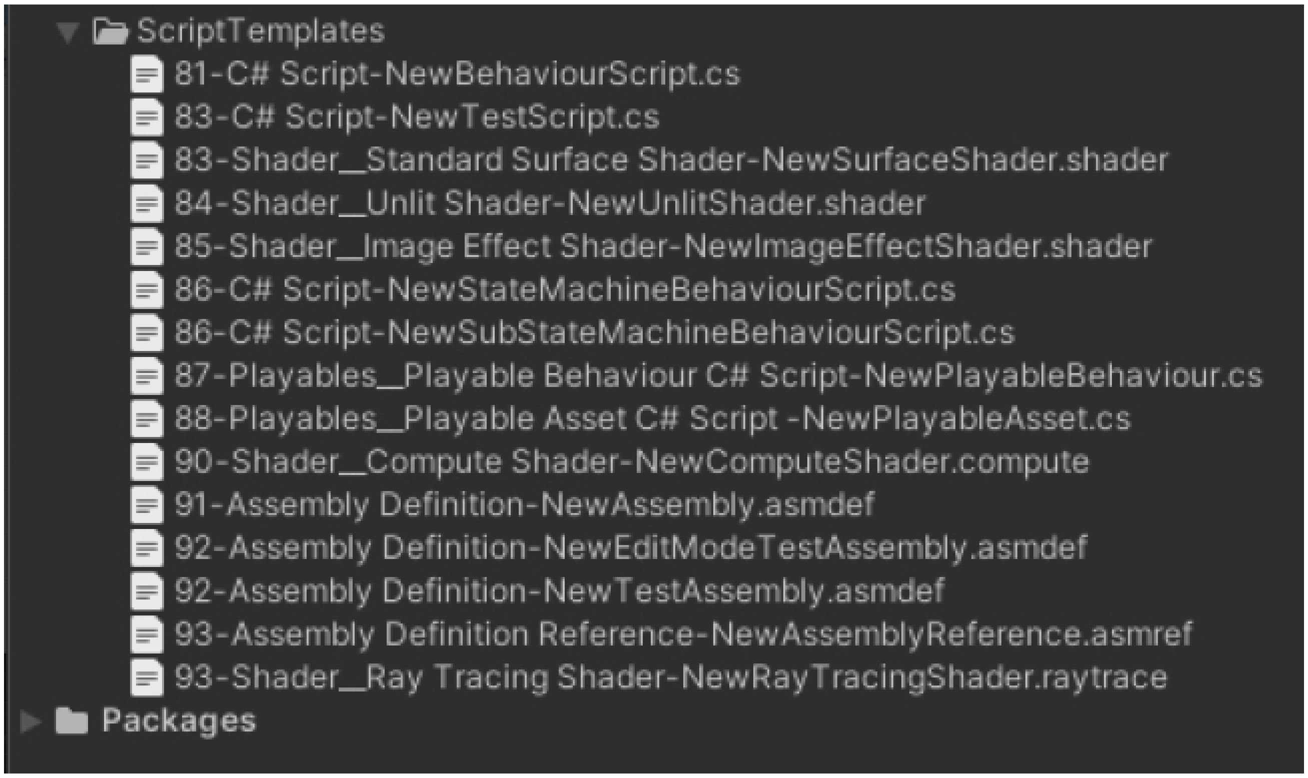
— **MenuPath** allows you to customize how the file appears in the Create menu. You can create categories with the double underscore**().**
— MenuPath允许你自定义文件在创建菜单中的显示方式。你可以用双下划线()创建类别。

— For example, "CustomScript**Misc**ScriptableObject" creates the menu item ScriptableObject under the **Create > CustomScript > Misc menu**.
— 例如，"CustomScript**Misc**ScriptableObject"在**Create > CustomScript > Misc菜单**下创建菜单项ScriptableObject。

— **DefaultName** is the default name given to the asset if you don't specify one.
— **DefaultName**是给资产的默认名称，如果你不指定一个。

— **FileExtension** is the file extension appended to the asset name. Also, note that each script template also has a .txt appended to the FileExtension.
— **FileExtension**是附加到资产名称的文件扩展名。另外，请注意，每个脚本模板也有一个.txt附加到FileExtension。

If you want to apply a script template to a specific Unity project, copy and paste the entire ScriptTemplates folder directly under the project's Assets.
如果你想将脚本模板应用到特定的Unity项目中，请将整个ScriptTemplates文件夹复制并粘贴到项目的Assets下。


```
ScriptTemplates
    81-C# Script-NewBehaviourScript.cs
    83-C# Script-NewTestScript.cs
    83-Shader__Standard Surface Shader-NewSurfaceShader.shader
    84-Shader__Unlit Shader-NewUnlitShader.shader
    85-Shader__Image Effect Shader-NewImageEffectShader.shader
    86-C# Script-NewStateMachineBehaviourScript.cs
    86-C# Script-NewSubStateMachineBehaviourScript.cs
    87-Playables__Playable Behaviour C# Script-NewPlayableBehaviour.cs
    88-Playables__Playable Asset C# Script -NewPlayableAsset.cs
    90-Shader__Compute Shader-NewComputeShader.compute
    91-Assembly Definition-NewAssembly.asmdef
    92-Assembly Definition-NewEditModeTestAssembly.asmdef
    92-Assembly Definition-NewTestAssembly.asmdef
    93-Assembly Definition Reference-NewAssemblyReference.asmref
    93-Shader__Ray Tracing Shader-NewRayTracingShader.raytrace
Packages
```
The ScriptTemplates copied to the Unity project. 脚本模板复制到Unity项目。

Next, create new script templates or modify the originals to fit your preferences. Delete any script templates from the project if you don't plan on changing them.
接下来，创建新的脚本模板或修改原始的脚本模板以适应你的偏好。如果你不打算更改它们，请从项目中删除任何脚本模板。

For example, you could create a blank script template for ScriptableObjects. Make a new text file under the ScriptTemplates folder called:
例如，你可以为ScriptableObjects创建一个空白的脚本模板。在ScriptTemplates文件夹下创建一个名为:

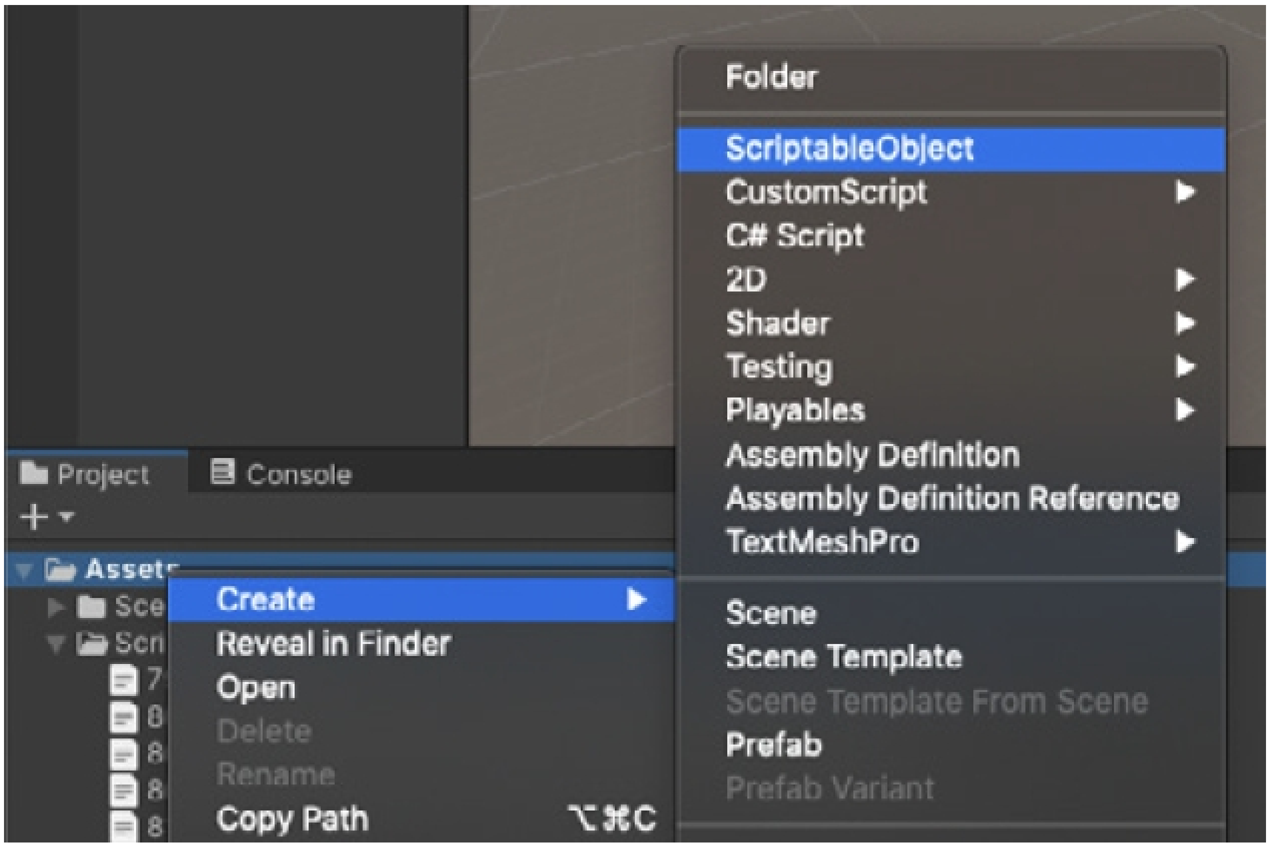
80-ScriptableObject-NewScriptableObject.cs.txt

Edit the text to read:
编辑文本以阅读:

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

[CreateAssetMenu(fileName = “#SCRIPTNAME#”, menuName = “#SCRIPTNAME#”)]
public class #SCRIPTNAME# : ScriptableObject
{
    #NOTRIM#
}
```

This creates a blank ScriptableObject script, complete with the CreateAssetMenu attribute.
这将创建一个空的ScriptableObject脚本，包括CreateAssetMenu属性。

Restart the Editor after you save the script template. Next time you should see an extra option in the Create menu.
在保存脚本模板后重新启动编辑器。下次你应该在创建菜单中看到一个额外的选项。



A custom script template adds a new menu item in the Create menu. 自定义脚本模板在创建菜单中添加了一个新的菜单项。
Create a new ScriptableObject script (and a corresponding ScriptableObject asset) from the

Create menu.
从创建菜单中创建一个新的ScriptableObject脚本(以及相应的ScriptableObject资产)。

Be sure to back up both the customized script templates and the originals. You will need to restore any files if Unity fails to recognize a modified template.
一定要备份定制的脚本模板和原始的脚本模板。如果Unity无法识别修改后的模板，你需要恢复任何文件。

Once you have a set of script templates you like, copy your ScriptTemplates folder to a new project and customize them to your specific needs. You can also change the original script templates in the application resources but exercise caution. That affects all projects using that version of Unity.
一旦你有了一套你喜欢的脚本模板，将你的ScriptTemplates文件夹复制到一个新项目中，并根据你的特定需求进行自定义。你也可以改变应用程序资源中的原始脚本模板，但要小心。这影响使用该版本Unity的所有项目。

See this support article for more information about customizing your script templates. Also, check the attached project for a few additional script template examples.
有关自定义脚本模板的更多信息，请参阅支持文章。此外，检查附加的项目，了解一些额外的脚本模板示例。

APPENDIX: TESTING AND DEBUGGERRING 附录:测试和调试

"EBUGGING IS LIKE BEING A DETECTIVE IN A CRIME MOVIE WHERE YOU ARE ALSO THE MURDERER." — Filipe Fortes
"调试就像是在一部犯罪电影中扮演侦探，而你也是凶手。" —— Filipe Fortes

Automated testing is an effective tool for improving the quality of your code and reducing the time spent on bug fixes. Test-driven development (TDD) is a development methodology where you create unit tests while you develop the software. In fact, you'll routinely write each test case before making a specific feature function.

As you develop the software, you'll repeatedly run it against this whole test suite of automated processes. This is in stark contrast to writing the software first and building the test cases later. In TDD, coding, testing, and refactoring are interwoven.

Here's the basic idea, presented in Kent Beck's Test-Driven Development by Example:

1. **Add a single unit test**: This describes one new feature you want to add to your application; spec out what needs to be done, either from your team or your user base.
   添加一个单元测试:这描述了你想要添加到你的应用程序的一个新功能;从你的团队或用户群中列出需要做的事情。

2. **Run the test**: The test should fail since you haven't implemented the new feature into your program. Additionally, this verifies whether or not the test itself is valid. It should not always pass by default.
   运行测试:测试应该失败，因为你还没有将新功能实现到你的程序中。此外，这还验证了测试本身是否有效。它不应该总是默认通过。
3. **Write the simplest code that passes the new test**: Write just enough logic to make it pass the new unit test. This doesn't have to be clean code at this point. It can use inelegant structure, hard-coded magic numbers, and so on, as long as it passes the unit test.
   编写最简单的代码，通过新的测试:只编写足够的逻辑来使它通过新的单元测试。这一点上的代码不一定是干净的。它可以使用不优雅的结构、硬编码的魔术数字等，只要它通过单元测试。
4. **Confirm that all tests pass**: Run the full automated test suite. Your previous unit tests should all pass. The new code meets your new testing requirements and the old requirements as well. If not, modify your new code – and only your new code – until all tests pass.
   确认所有测试都通过:运行完整的自动化测试套件。你以前的单元测试应该都通过。新代码满足你的新测试要求和旧要求。如果没有，修改你的新代码——只修改你的新代码——直到所有的测试都通过。
5. **Refactor**: Go back and clean up your new code. Use your style guide and make sure everything conforms.
   Move the code, so it is logically organized. Keep similar classes and methods together, etc.
   Remove duplicate code, and rename any identifiers to minimize the need for comments.
   Split methods or classes that are too long.
   Run the automated testing suite after each refactor.
   重构:回过头来清理你的新代码。使用你的样式指南，确保一切符合规范。移动代码，使其逻辑上有组织。保持相似的类和方法在一起，等等。删除重复的代码，并重命名任何标识符，以最小化对注释的需求。分割太长的方法或类。每次重构后运行自动化测试套件。
6. **Repeat**: Go through this process every time you add a new feature. Each step is a small, incremental change. Make frequent commits under source control. When debugging, you only have to examine a small amount of new code for each unit test. This simplifies the scope of your work. If all else fails, roll back to the previous commit and begin again.
   重复:每次添加一个新功能时都要经历这个过程。每一步都是一个小的、增量的改变。在源代码控制下频繁提交。当调试时，你只需要检查每个单元测试的一小部分新代码。这简化了你的工作范围。如果其他方法都失败了，回滚到上一个提交，重新开始。

That's the gist of it. If you develop software using this methodology, you tend to follow the KISS principle by necessity. Add one feature at a time, testing as you go. Refactor continuously with each test, so cleaning your code becomes a constant ritual.
这就是它的要点。如果你使用这种方法开发软件，你往往会按照KISS原则的必要性。一次只添加一个功能，测试时进行。每次测试都进行重构，这样清理你的代码就成了一个

Like most of the tenets of clean code, TDD takes extra work in the short-term but often results in the improvement of long-term maintenance and readability.
像大多数清洁代码的原则一样，TDD需要额外的短期工作，但通常会导致长期维护和可读性的改进。

Unity Test Framework Unity测试框架

The Unity Test Framework (UTF), formerly known as the Unity Test Runner, provides a standard test framework for Unity developers. UTF uses NUnit, an open-source testing library for .NET languages.
Unity测试框架(UTF)，以前称为Unity测试运行器，为Unity开发人员提供了一个标准的测试框架。UTF使用NUnit，一个用于.NET语言的开源测试库。

The Unity Test Framework can perform unit tests in the Editor (either using **Edit Mode** or **Play Mode**) and on target platforms (e.g., Standalone, Android, iOS). Install UTF via the Package Manager. The online documentation will help you get started.
Unity测试框架可以在编辑器中执行单元测试(使用编辑模式或播放模式)和目标平台(例如，独立、Android、iOS)。通过包管理器安装UTF。在线文档将帮助你入门。

The general workflow of the Unity Test Framework is to:
Unity测试框架的一般工作流程是:

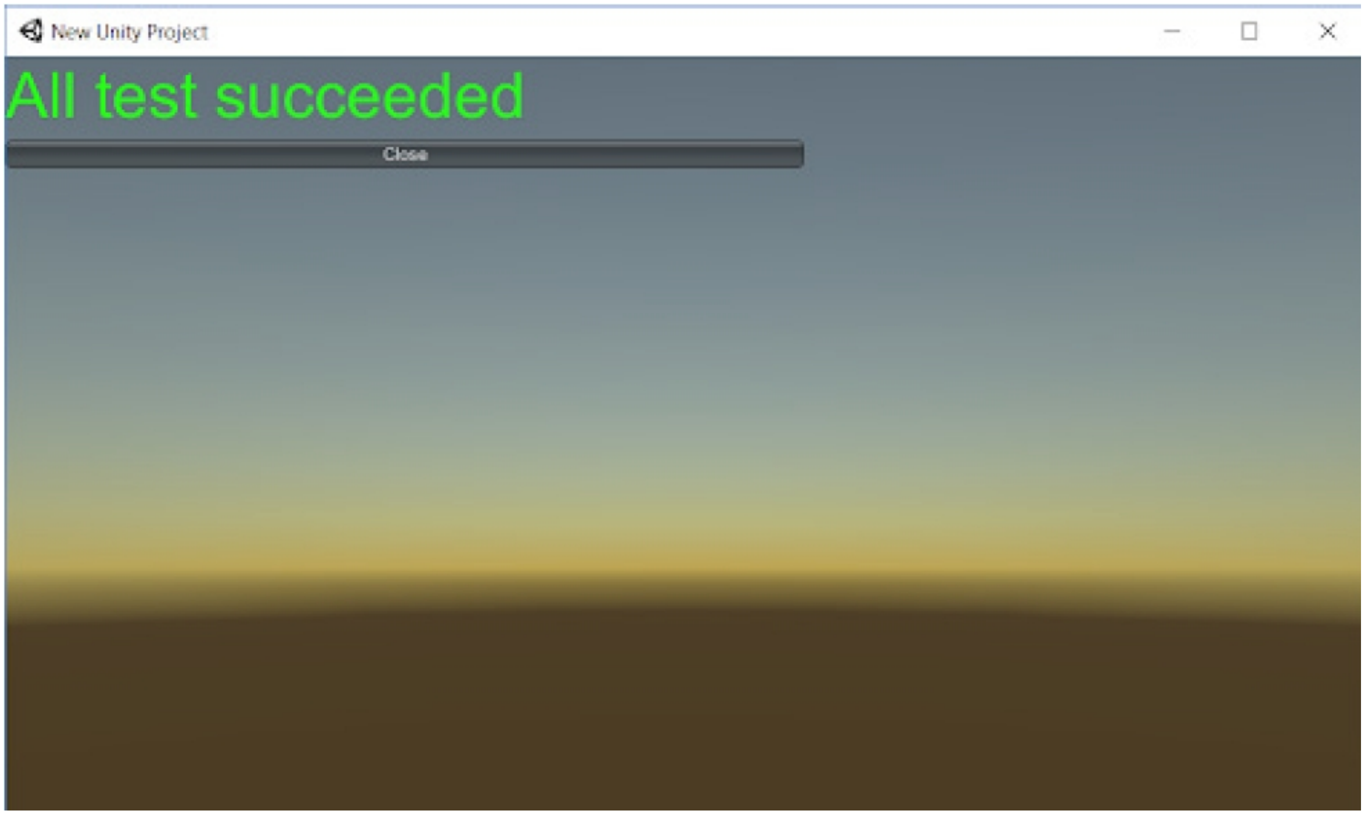
— **Create a new test suite, called a Test Assembly**: The Test Runner UI simplifies this process and creates a folder in your project.
— **创建一个新的测试套件，称为测试程序集**:测试运行器UI简化了这个过程，并在你的项目中创建了一个文件夹。

— **Create a test**: The Test Runner UI helps you manage the C# scripts that you will create as unit tests. Select aTest Assembly folder and navigate to **Assets > Create > Testing > C# Test Script**. Edit this script and add logic for your test.
— **创建一个测试**:测试运行器UI帮助你管理你将创建的C#脚本作为单元测试。选择一个Test Assembly文件夹，然后导航到**Assets > Create > Testing > C# Test Script**。编辑这个脚本，并为你的测试添加逻辑。

— **Run a test**: Use the Test Runner UI to run all of the unit tests or run a selected one. Using JetBrains Rider, you can also run UTF directly from the script editor.
— **运行一个测试**:使用测试运行器UI运行所有的单元测试或运行一个选定的单元测试。使用JetBrains Rider，你还可以直接从脚本编辑器运行UTF。

— **Add Play mode tests in the Editor or as standalone**: The default Test Assembly works in Edit Mode. If you want unit tests to work at runtime, create a separate assembly in Play Mode. Configure this for your standalone build as well (with the results of the test displayed in the Editor).

— **在编辑器中或作为独立的播放模式添加播放模式测试**:默认的Test Assembly在编辑模式下工作。如果你想让单元测试在运行时工作，在播放模式下创建一个单独的程序集。同样，也要为你的独立构建进行配置(在编辑器中显示测试结果)。



The Test Framework displays the results of a standalone build within the Editor. 测试框架在编辑器中显示独立构建的结果。
See the Test Framework microsite for more information about getting up and running with UTF. 有关使用UTF启动和运行的更多信息，请参阅测试框架微站。