1. Алгоритмические структуры (циклы и массивы)
2. Action, Event, Delegate
3. ООП
   1. Наследование
   2. Инкапсуляция
   3. Полиморфизм
   4. Абстракция
4. Синглтон
5. Сериализация данных (JSON, PlayerPrefs)
6. LINQ
7. Интерфейсы, абстрактные классы
8. Архитектурные паттерны
9. Поведенческие и производственные паттерны
10. Порядок выполнения ивентов MonoBehaviour
11. ScriptableObject

# Алгоритмические структуры (циклы и массивы)

Массив – это набор данных с одинаковым типом. В многомерных массивах каждый элемент массива также массив. В C# зубчатый массив состоит из нескольких элементов, каждый из которых является массивом. Отличие от многомерных массивов заключается в том, что каждый массив может быть разного размера.

С сайта майков:

Коллекции предоставляют гибкий способ работы с группами объектов. Вы можете классифицировать различные коллекции по следующим характеристикам:

Доступ к элементам: каждая коллекция может быть перечислена для доступа к каждому элементу в порядке. Некоторые коллекции обращаются к элементам по индексу, позиция элемента в упорядоченной коллекции. Наиболее распространенным примером является System.Collections.Generic.List<T>. Другие коллекции обращаются к элементам по ключу, где значение связано с одним ключом. Наиболее распространенным примером является System.Collections.Generic.Dictionary<TKey,TValue>. Вы выбираете между этими типами коллекций в зависимости от способа доступа к элементам приложения.

Профиль производительности. Каждая коллекция имеет различные профили производительности для действий, таких как добавление элемента, поиск элемента или удаление элемента. Вы можете выбрать тип коллекции на основе операций, используемых большинством в приложении.

Динамическое увеличение и сжатие: большинство коллекций, поддерживающих добавление или удаление элементов динамически. В частности, Array, System.Span<T>и System.Memory<T> не.

Коллекции

Самым примитивным способом хранения объектов в C# является использование массивов. Одной из основных проблем, с которой столкнется разработчик следуя такому подходу, является то, что массивы не предоставляют инструментов для динамического изменения размера. В языке C# есть два пространства имен для работы со структурами данных:

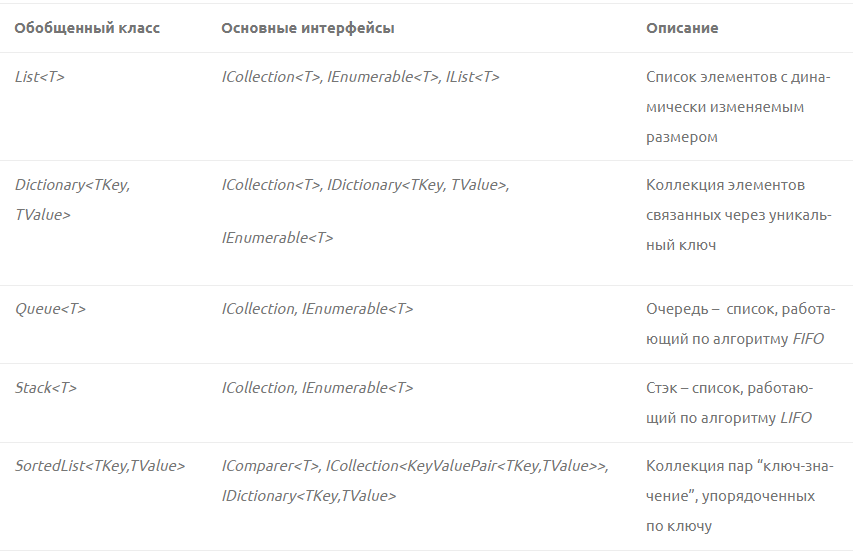
System.Collections;

System.Collections.Generic.

Первое из них – System.Collections предоставляет структуры данных для хранения объектов типа Object. У этого решения есть две основных проблемы – это производительность и безопасность типов. В настоящее время не рекомендуется использовать объекты классов из System.Collections.

Для решения указанных выше проблем Microsoft были разработаны коллекции с обобщенными типами (их ещё называют дженерики), они расположены в пространстве имен System.Collections.Generic. Суть их заключается в том, что вы не просто создает объект класса List, но и указываете, объекты какого типа будут в нем храниться, делается это так: List<T>, где T может быть int, string, double или какой-то ваш собственный класс.

В рамках данного урока мы не будем подробно останавливаться на особенностях обобщенных типов, на текущий момент можете их воспринимать как псевдонимы, для реальных типов данных.



Циклы используются для обхода массивов.

Циклы являются управляющими конструкциями, позволяя в зависимости от определенных условий выполнять некоторое действие множество раз. В C# имеются следующие виды циклов:

For

for ([действия\_до\_выполнения\_цикла]; [условие]; [действия\_после\_выполнения])

{

// действия

}

Foreach

foreach(тип\_данных переменная in коллекция)

{

// действия цикла

}

While

while (условие)

{

действия цикла

}

Do...while

do

{

действия цикла

}

while (условие)

# Action, Event, Delegate

Делегаты представляют такие объекты, которые указывают на методы. То есть делегаты - это указатели на методы и с помощью делегатов мы можем вызвать данные методы.

Message message1 = Welcome.Print;

Message message2 = new Hello().Display;

message1(); // Welcome

message2(); // Привет

delegate void Message();

class Welcome

{

public static void Print() => Console.WriteLine("Welcome");

}

class Hello

{

public void Display() => Console.WriteLine("Привет");

}

Ключевое слово event используется для объявления события. Событие, это не что иное, как ситуация, при возникновении которой, произойдет действие или несколько действий. Говоря языком программного моделирования, Событие — это именованный делегат, при вызове которого, будут запущены все подписавшиеся на момент вызова события методы заданной сигнатуры.

В .NET есть несколько встроенных делегатов, которые используются в различных ситуациях. И наиболее используемыми, с которыми часто приходится сталкиваться, являются Action, Predicate и Func.

Делегат Action представляет некоторое действие, которое ничего не возвращает, то есть в качестве возвращаемого типа имеет тип void

Делегат Predicate<T> принимает один параметр и возвращает значение типа bool

Еще одним распространенным делегатом является Func. Он возвращает результат действия и может принимать параметры. Он также имеет различные формы: от Func<out T>(), где T - тип возвращаемого значения, до Func<in T1, in T2,...in T16, out TResult>(), то есть может принимать до 16 параметров.

# ООП

## Наследование

Наследование — это возможность порождать один класс от другого с сохранением всех свойств и методов класса-предка (суперкласса), добавляя при необходимости новые свойства и

методы.

Наследоваться можно только от одного класса. Класс наследник получает все свойства и методы класса родителя, а также конструктор.

Интерфейс – класс, который содержит методы без их реализации. Реализуют их классы наследники.

## Инкапсуляция

Инкапсуляция – это принцип, согласно которому любой класс и в более широком смысле – любая часть системы должны рассматриваться как «черный ящик»: пользователь класса или подсистемы должен видеть только интерфейс (т.е. список декларируемых свойств и методов) и не вникать во внутреннюю реализацию.

Класс – это капсула.

Модификаторы доступа и свойства гет-сет позволяют защитить класс от нежелательного внешнего воздействия

Апкастеры и даункастеры

## Полиморфизм

Полиморфизм позволяет одному и тому же методу или классу работать по-разному в разных частях кода и обрабатывать разные типы данных.

Сюда относятся перегрузки методов, дженерик методы и классы, тип данных var, переопределение методов

## Абстракция

Следует отделять абстракцию от конкретной реализации с помощью интерфейсов и абстрактных классов.

Интерфейс представляет ссылочный тип, который может определять некоторый функционал - набор методов и свойств без реализации. Затем этот функционал реализуют классы и структуры, которые применяют данные интерфейсы.

# Синглтон

Singleton (Синглтон) — некий менеджер, через который производится управление игровыми скриптами. Как правило, синглтоны сохраняются от сцены к сцене без повторной реинициализации (наподобие глобального объекта).

public class GameManager : MonoBehaviour

{

public static GameManager instance { get; private set; }

private void Awake()

{

if (instance == null)

{

instance = this;

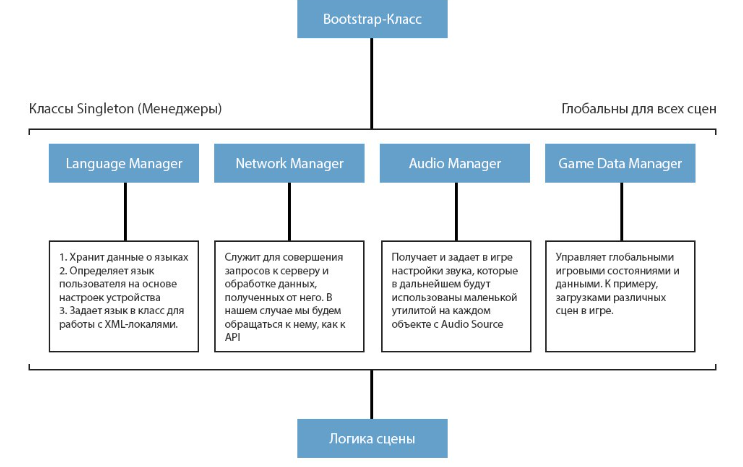
DontDestroyOnLoad(this.gameObject);

return;

}

Destroy(gameObject);

}



В данном случае помимо объектов Singleton у нас будут присутствовать следующие элементы:  
  
Объекты для подгрузки Singleton (Так называемый Bootstrap-класс)

1. Объекты игровой логики (объекты управления сценариями)
2. Контроллеры (Например: контроллер игрока)
3. Модели данных (объекты для сериализации данных, получаемых с сервера)
4. Объекты интерфейса
5. Прочие статические игровые объекты

# Сериализация данных (JSON, PlayerPrefs)

PlayerPrefs - это класс, для работы с нашими данными, которые мы хотим сохранять или переносить. Данные, которые мы заносим в PlayerPrefs хранятся в разных местах для разных устройств. Например, в Web приложениях, которые мы запускаем на Windows они хранятся в %APPDATA%\Unity\WebPlayerPrefs, а обычные приложения сохраняют себя в реестр (HKEY\_CURRENT\_USER\Software\[Название компании]\[Название продукта]). PlayerPrefs довольно небезопасный способ хранения данных, поэтому их в основном применяют для сохранения настроек и чего-то неважного, чтоб это не влияло на игру.

Но можно модифицировать данные, которые мы заносим в PlayerPrefs для того, чтоб их изменение видилось и не применялось. Например, можно использовать шифрование, и тогда все данные будут представляться результатами шифрования. После этого чтоб их получить, нужно расшифровать их в коде игры.

Сериализация представляет собой процесс автоматического преобразования структур данных либо состояний объекта в иной формат. Когда используем этот термин относительно Unity, мы подразумеваем упрощение хранения данных и их реконструкцию. Это один из приемов повышения продуктивности разработки.

Итак, структуру и класс мы можем пометить как сериализуемые. Для этого потребуется указать [Serializable] над именем.

Форматы:

- Xml-сериализация

+ При больших объемах данных\* быстрее всех сериализуется и десериализуется

- Json-сериализация

+ при малых объемах данных\* быстрее всех сериализуется и десериализуется

+ исходный файл меньше весит при больших и малых объемах данных

Идеально подходит для кратковременной сериализации объектов Unity.

- Бинарная-сериализация

+ нечитабельное содержимое, позволяющее скрыть информацию от сторонних глаз

+ поддерживает больше типов для де/сериализации

Подходит к долговременной сериализации объектов Unity, где подразумевается сокрытие данных от "взломщиков".

Большим объемом считается количество более 1500 объектов. Малым объемом считается количество менее 500 объектов.

XML сериализация сериализует только публичные поля и свойства. XML сериализация должна должна на этапе компиляции располагать информацией о типах, которые сериализует. Сериализуемые объекты должны иметь беспараметрический конструктор. Свойства с модификатором readonly не сериализуются.

Рассмотрим распространенные атрибуты XML-сериализации:

[XmlElement]: поле будет сериализовано в качестве элемента

[XmlAttribute]: поле будет сериализовано в качестве атрибута

[XmlIgnore]: поле будет пропущено во время сериализации

[XmlRoot]: задает корневой элемент при сериализации

Перечислю, пожалуй, основные атрибуты для сериализации в json, которые вам могут пригодиться в дальнейшем:

[JsonObjectAttribute] - атрибут, который используется для задания поведения класса при сериализации

[JsonPropertyAttribute] - атрибут, который используется для задания поведения свойств и полей при сериализации

[JsonIgnore] - атрибут, позволяющий игнорировать поле или свойство при сериализации

# LINQ

LINQ (Language-Integrated Query) представляет простой и удобный язык запросов к источнику данных. В качестве источника данных может выступать объект, реализующий интерфейс IEnumerable (например, стандартные коллекции, массивы), набор данных DataSet, документ XML. Но вне зависимости от типа источника LINQ позволяет применить ко всем один и тот же подход для выборки данных.

Существует несколько разновидностей LINQ:

LINQ to Objects: применяется для работы с массивами и коллекциями

LINQ to Entities: используется при обращении к базам данных через технологию Entity Framework

LINQ to XML: применяется при работе с файлами XML

LINQ to DataSet: применяется при работе с объектом DataSet

Parallel LINQ (PLINQ): используется для выполнения параллельных запросов

Основная часть функциональности LINQ сосредоточена в пространстве имен System.LINQ.

Позволяет в одну строчку совершать сложные операции с коллекциями и массивами.

# Интерфейсы, абстрактные классы

Интерфейс представляет ссылочный тип, который может определять некоторый функционал - набор методов и свойств без реализации. Затем этот функционал реализуют классы и структуры, которые применяют данные интерфейсы.

Для определения интерфейса используется ключевое слово interface. Как правило, названия интерфейсов в C# начинаются с заглавной буквы I, например, IComparable, IEnumerable (так называемая венгерская нотация), однако это не обязательное требование, а больше стиль программирования.

Что может определять интерфейс? В целом интерфейсы могут определять следующие сущности:

Методы

Свойства

Индексаторы

События

Статические поля и константы (начиная с версии C# 8.0)

Абстрактные классы полезны для описания некоторого общего функционала, который могут наследовать и использовать производные классы

Кроме обычных свойств и методов абстрактный класс может иметь абстрактные члены классов, которые определяются с помощью ключевого слова **abstract** и не имеют никакого функционала. В частности, абстрактными могут быть:

* Методы
* Свойства
* Индексаторы
* События

# Архитектурные паттерны

## Singleton (одиночка)

Одиночка (Singleton, Синглтон) — порождающий паттерн, который гарантирует, что для определенного класса будет создан только один объект, а также предоставит к этому объекту точку доступа.

Когда надо использовать Синглтон? Когда необходимо, чтобы для класса существовал только один экземпляр

Синглтон позволяет создать объект только при его необходимости. Если объект не нужен, то он не будет создан. В этом отличие синглтона от глобальных переменных.

## Object pool (пул объектов)

Улучшение производительности и эффективности использования памяти за счет повторного использования объектов из фиксированного пула, вместо их индивидуального выделения и освобождения.

Пул нужен для оптимизации в тех играх, где часто создаются и уничтожаются какие-нибудь объекты. Например, пули или враги.

При создании нового объекта, программа пробегается по оперативной памяти, выискивая свободное местечко куда бы этот объект запихнуть. Это долго. А потом надо еще записать в найденное место байты этого объекта, запустить конструктор, всякие события в движке, происходящие при создании объекта, и так далее. Всё это жрёт большое количество процессорного времени и может приводить к тормозам. Особенно на слабых компьютерах и на телефонах.

При частом удалении объектов тоже могут быть лаги, связанные с работой сборщика мусора. Сборщик мусора обычно запускается когда приложению нужно больше памяти, чем оно уже занимало раньше. Сборщик пробегается по уже занятой памяти, и пытается определить, какие объекты уже не будут использоваться в игре, потому что ни одному другому объекту они больше не нужны. Это занимает очень много процессорного времени.

## Command (команда)

Паттерн «Команда» (Command) позволяет инкапсулировать запрос на выполнение определенного действия в виде отдельного объекта. Этот объект запроса на действие и называется командой. При этом объекты, инициирующие запросы на выполнение действия, отделяются от объектов, которые выполняют это действие.

Команды могут использовать параметры, которые передают ассоциированную с командой информацию. Кроме того, команды могут ставиться в очередь и также могут быть отменены.

Пример применения: инкапсулировать функциональность игры в виде команд, которые можно вводить в консоль, сохранять и воспроизводить, или даже создавать сценарии для помощи в тестировании игры.

Вы задавались когда-нибудь вопросом, как в играх наподобие Super Meat Boy реализована функция реплея? Один из способов её реализации — выполнять ввод точно так же, как это делал игрок, что, в свою очередь, означает, что ввод нужно как-то хранить. Для этого и многого другого можно использовать шаблон Command.

Шаблон Command («Команда») также полезен для создания функций «Отменить» (Undo) и «Повторить» (Redo) в стратегической игре.

## State (состояние)

Состояние (State) — шаблон проектирования, который позволяет объекту изменять свое поведение в зависимости от внутреннего состояния.

Когда применяется данный паттерн?

Когда поведение объекта должно зависеть от его состояния и может изменяться динамически во время выполнения

Когда в коде методов объекта используются многочисленные условные конструкции, выбор которых зависит от текущего состояния объекта

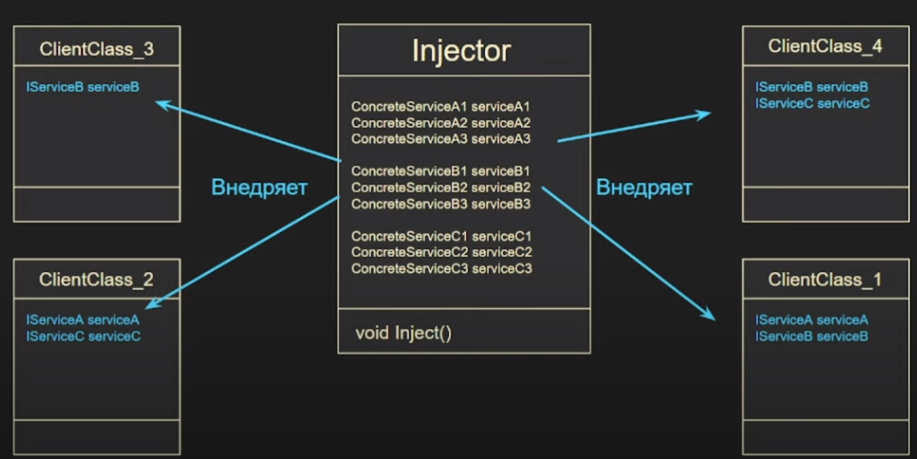
## Dependency injection (внедрение зависимости)

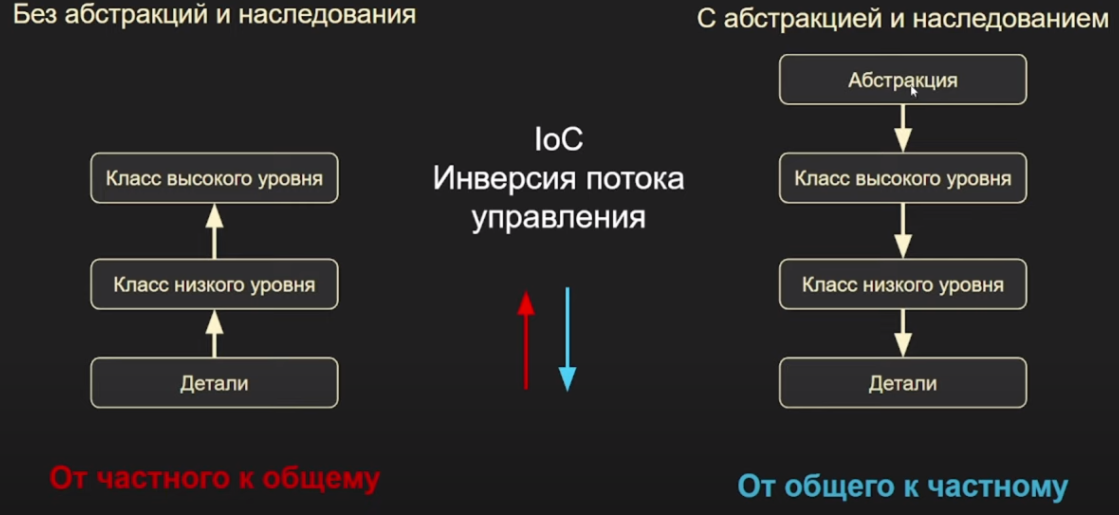
it’s just a way for you class to reference other classes. So instead of creating instances of others classes directly or use GetComponent to obtain references to them, we receive them from outside, usually through the constructor in plain classes or using the SerializeField attribute.

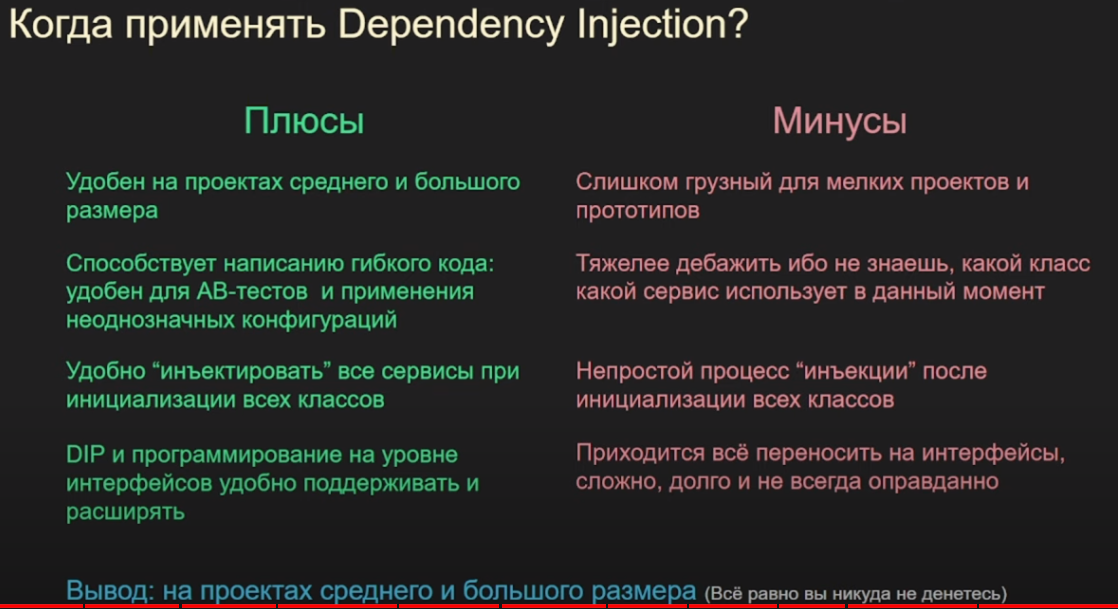
Yes, you read it correctly, just by using SerializeField to obtain your references to other MonoBehaviour, you are already using dependency injection.

Dependency injection is a technique for automatically wiring together your components. It will help bring order to your work as the complexity of your game code explodes out of control. I can't recommend it enough.









## Factory method (фабричный метод)

Фабричный метод (Factory Method) - это паттерн, который определяет интерфейс для создания объектов некоторого класса, но непосредственное решение о том, объект какого класса создавать происходит в подклассах. То есть паттерн предполагает, что базовый класс делегирует создание объектов классам-наследникам.

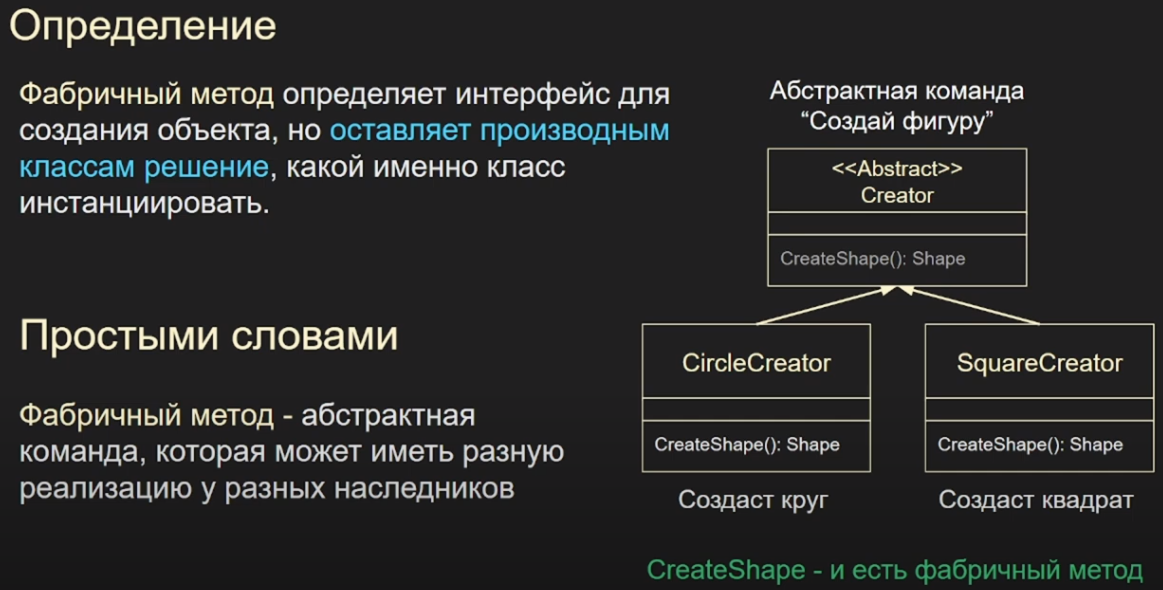
Когда надо применять паттерн

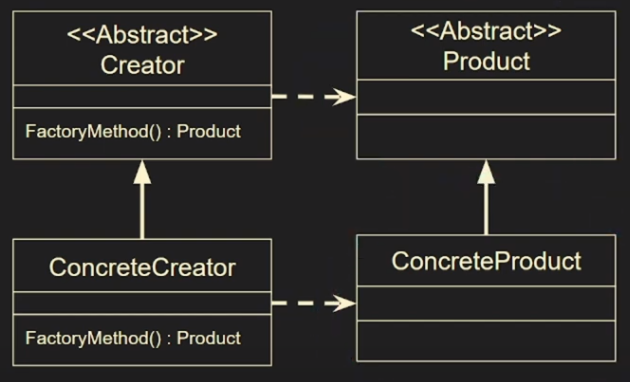
Когда заранее неизвестно, объекты каких типов необходимо создавать

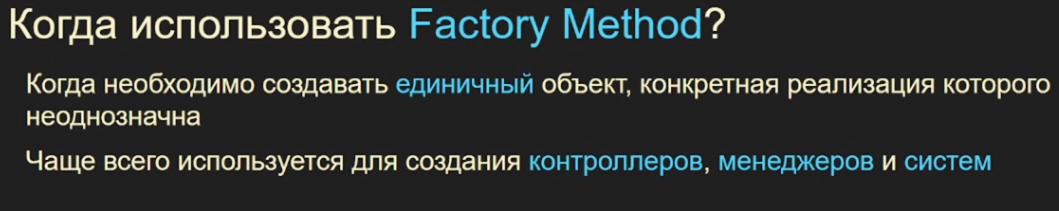
Когда система должна быть независимой от процесса создания новых объектов и расширяемой: в нее можно легко вводить новые классы, объекты которых система должна создавать.

Когда создание новых объектов необходимо делегировать из базового класса классам наследникам

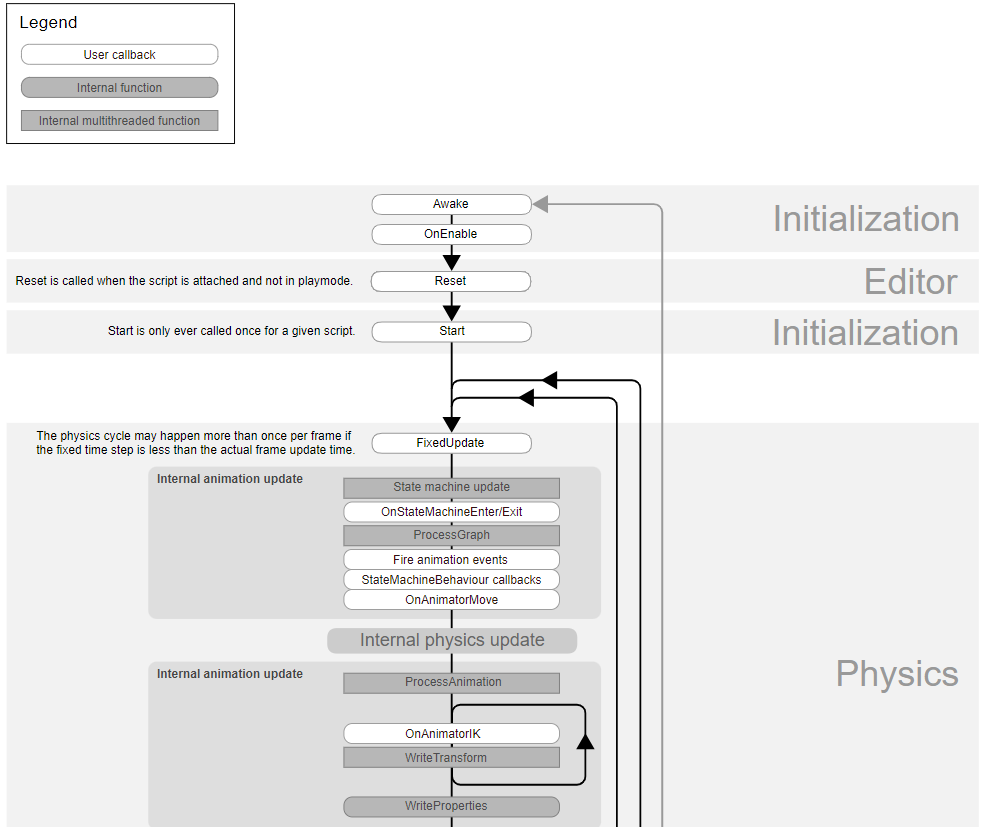


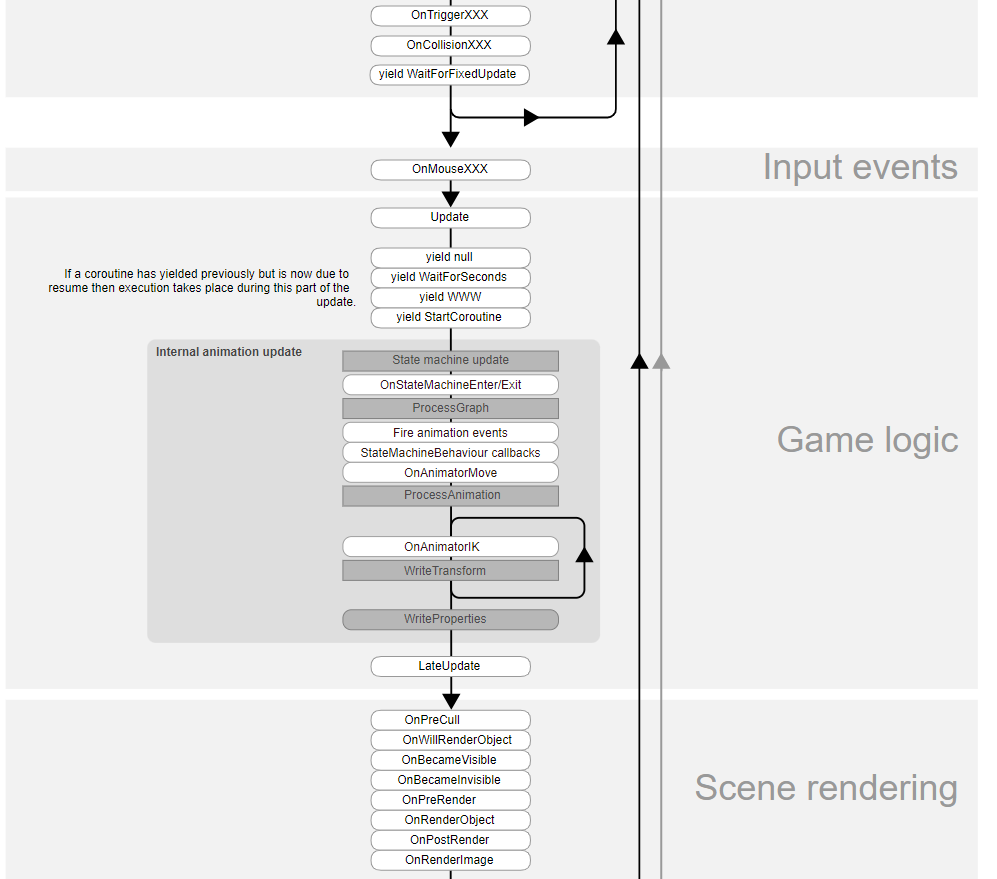


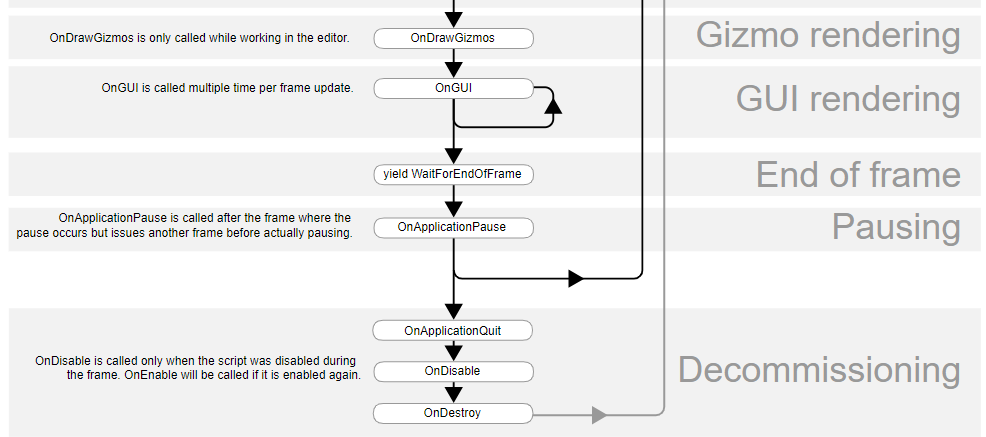




# Порядок выполнения ивентов MonoBehaviour







# ScriptableObject

Согласно документации Unity, ScriptableObject — это код класса, позволяющий создавать в игре Scriptable Objects для хранения больших объёмов общих данных, не зависящих от экземпляров скриптов.

Существует множество причин для использования Scriptable Objects в Unity. Они могут снизить объём используемой под каждый дополнительный префаб памяти, потому что по своей сути Scriptable Object следуют паттерну разработки Flyweight.

[CreateAssetMenu(fileName = "ChestInfo", menuName = "Gameplay/New ChestInfo")]

public class ChestInfo : ScriptableObject

{

[SerializeField] private string \_id;

[SerializeField] private GameObject \_prefab;

[SerializeField] private Sprite \_spriteIcon;

public string id => \_id;

public GameObject prefab => \_prefab;

public Sprite spriteIcon => \_spriteIcon;

}

public class Example : MonoBehaviour

{

void Start()

{

var allChestInfos = Resources.LoadAll<ChestInfo>("");

foreach (var chestInfo in allChestInfos)

{

Debug.Log(chestInfo.id);

}

}

}