**抽象类/抽象方法/接口**

具有抽象方法的类是抽象类，抽象类中可以有普通成员变量或方法，接口中没有普通成员变量，只能为抽象方法或属性；一个类可以继承多个接口，但只能继承一个类。

**const和readonly的区别？**

const是编译时常量，需在声明时初始化；readonly是运行时常量，可在声明时或构造函数中初始化，static不能用于修饰const。

const只能用于值类型和string类型，readonly都可以。

**引用类型与值类型/类和结构的区别是什么？**

当作为方法的局部变量时，值类型是在栈中分配的，而引用类型实例在堆中分配，其变量存储了地址在栈中分配。

引用类型赋值是复制内存地址，而值类型赋值是逐一字段复制。

值类型由System.ValueType派生，重写了Equals、GetHashCode和ToString。（调用struct的这三个函数可能会产生装箱）

值类型不可继承类或被继承，但可以继承接口。

**装箱与拆箱？**

装箱：将值类型字段复制到堆中的新分配的内存中并返回内存地址，从而使值类型对象变为引用类型对象。

拆箱：将引用类型对象的值拷贝到栈上的值类型实例中，从而使引用类型对象变为值类型对象。

避免值类型的LINQ表达式；若值类型没有重载Equals、GetHashCode和ToString函数，则避免调用它们；避免使用参数为object的方法（如Debug.Log）。

**反射**

可以使用反射从现有对象中获取类型，然后调用其方法或访问其字段、属性、特性；也可以动态地创建类型的实例，将类型绑定到现有对象。

反射的缺点：

①反射会造成编译时无法保证类型安全性

反射依赖字符串，所以会丧失编译时的类型安全性（即：编译成功，运行报错）。

②反射速度慢

反射依赖字符串，会执行字符串的搜索，会比较影响性能。

编译器不能做任何优化。

所有被调用、创建的东西都必须被发现。

参数需要通过装箱/拆箱等方式重新包装。

反射优化：属性赋值采用委托访问属性、定义接口使目标继承将其引用放到接口类变量中。

**委托**

一种回调函数机制。委托会确保类型安全，而且还允许顺序调用多个方法，并支持调用静态方法和实例方法。

无返回值的委托利用Action实现，有返回值的委托利用Func实现，predicate<T>与Func<T, bool>相同。

Unity对Action进行了封装，即UnityAction和UnityEvent，可以在编辑器上序列化显示和操作。

event关键字：用于修饰委托，事件是一种特殊的多播委托，仅可以从声明事件的类中对其进行调用。

lambda表达式：基于委托实现，相当于没有声明的方法，可实现闭包。

闭包：声明时捕获作用域中环境状态，会对变量的生命周期进行延长，提升到堆上。

**字符串**

string，一种特殊的引用类型。应采取StringBuilder实现。

创建string对象时会加入常量区的三种方式：

①利用字面量值创建

②利用string.Intern()创建

③字面量值+字面量值拼接创建（concat、内插$）

只有编译阶段的文本字符常量会被自动添加到暂存池。

运行时期动态创建的字符串不会被加入到暂存池中，而是托管堆。

**垃圾回收/GC**

一致内存管理的机制，指将不再被需要的、废弃的内存重新回收以供再次使用的过程。会定时执行，或在申请内存分配时发现内存不足时触发执行，也可强制调用。

**GC过程**

① GC准备阶段：暂停进程中的所有线程，避免在GC期间访问堆内存。

② GC标记阶段：GC会检查堆内存上的每个存储变量，对每个变量会检测其引用是否处于激活状态，如果变量的引用不再处于激活状态，则会被标记为可回收。（根节点引用链法、引用计数法。根节点可以为当前执行方法中的本地变量、静态变量、重写了Finalize的对象；引用计数法与智能指针相类似，难以解决循环引用的问题）

③ 垃圾回收阶段：被标记的变量会被移除，其所占有的内存会被回收到堆内存上。

**GC带来的问题**

① 需要大量时间来运行，可能会使得游戏运行缓慢。

② 会产生很多内存碎片，如果不整理会导致游戏占用内存越来越大，GC也会更频繁触发。

**降低GC影响的方法**

减少GC的运行次数。

减少单次GC运行时间。

控制GC运行时间，避免在关键时候触发。

核心在于减少内存垃圾的数量。

缓存

反复调用造成堆内存的函数，可以将返回结果缓存为变量来重复利用（比如GetComponentsInChildren）。

避免在频繁调用的函数中反复进行堆内存分配

若造成堆内存的函数结果可能变化，仅当造成变化的参数产生变化时进行调用，或采用计时器使其固定周期调用一次而不是每帧调用。

清除容器

对容器进行反复分配时，可以采用Clear函数来清空容器替代。

对象池

对于游戏中大量需要产生和销毁的对象采用对象池用SetActive代替产生和销毁。

字符串优化

字符串的值在创建后不可变更，可减少不必要的字符串构建，如果一个字符串的值被多次利用，创建并缓存该字符串。

减少不必要的字符串操作，对Text组件可分为不变部分和变化部分。

不要使用String的+，用StringBuilder代替。

比较tag用CompareTag()。

减少拆箱与装箱、避免值类型的LINQ表达式

协程优化

yield return null而不是yield return 0。

避免每次返回都new同一个变量，提前缓存。

减少函数引用，避免将方法作为参数传递，优先采用匿名方法

代码重构

减少对不必检查对象的检查，如避免struct中包含引用类型对象。

减少引用，比如用索引代替。

手动调用GC

定时进行GC操作，比如在场景切换时手动调用System.GC.Collect()。

增量式GC

将GC的操作跨多个帧来操作，来降低GC的峰值。

分代垃圾回收

对象越新，生命周期越短；回收堆的一部分，性能更优

基本分代：0代：从未被标记为回收的新分配对象；1代：上一次垃圾回收中没有被回收的对象；2代：在一次以上的垃圾回收后依然未被回收的对象。一般0:1:2的频率为100:10:1。

Unity中的GC检测：Window->Profiler可以检测每一帧的堆内存分配

**C#编译的过程？**

基于托管环境的编译。在词法分析和语法分析后，编译器不直接编译成机器码，而是转换为中间语言（CIL/MSIL）。而后在运行时通过CLR使用JIT编译器将中间语言翻译成机器码。

**容器与C++ STL比较**

C#→C++：

List→vector、

LinkedList→list、

Dictionary→unordered\_map、

SortedDictionary（二叉搜索树）→map（红黑树）、

HashSet→unordered\_set、

SortedSet→set。

Dictionary底部通过哈希表实现，支持泛型，但是是非线程安全的。

HashTable不支持泛型，在存储和检索值类型时会导致装箱和拆箱操作。

ArrayList不支持泛型，基于object，会进行装箱和拆箱，其余与List类似。

**浅拷贝与深拷贝**

浅拷贝：将原对象或原数组的引用直接赋给新对象。MemberwiseClone()。

深拷贝：完全复制一个对象，可以通过自定义对象的拷贝构造函数、利用反射、或序列化和反序列化（json或二进制）来实现深拷贝。

**强引用与弱引用**

强引用：正常的引用， 对GC产生影响。

弱引用：WeakReference，不会对GC产生影响。

**C#泛型与C++模板的区别**

C#泛型不支持非类型参数（值参数），C++模板支持。

C#泛型不支持特性类型模板自定义实现，C++模板支持。

C#泛型不支持默认（类型）参数，C++模板支持。

C#泛型类型参数本身不能是泛型， C++模板支持使用模板参数。

C#泛型不支持调用算术运算符，C++模板支持。

C#泛型支持类型约束（where T : struct/class/class?/new()/U/U?），C++11模板可以使用静态断言来约束。