Misconceptions about DOTS and Entities

False: DOTS, ECS, Unity Entities, and Data-oriented Design are all the same thing

- **Entity Component System (ECS)**: an architectural pattern that originated in some games of the 2000's. There are at least several variants of the architecture, but they are all built around two common core ideas:
    - A clear separation of data and code (entities and components are the data; systems are the code)
    - A composable data structure (entities composed of components)
- **Unity Entities**: a Unity package that provides a variant of ECS architecture. You'll often see "ECS" or "Unity ECS" used as synonyms for Unity Entities.
- **Data-oriented Design (DoD)**: a set of design principles about performance and problem solving that emerged in the 2000's, partly in conjunction with ECS. Not to be confused with "data-oriented programming", "data-driven design", or "data-driven programming", which are entirely separate concepts despite the similar names.
- **Unity's Data-Oriented Technology Stack (DOTS)**: a set of Unity features built on DoD principles and which includes:
    - The Entities package
    - The Collections package
    - The Mathematics package
    - The Burst compiler package
    - The C# job system
    - As well as feature packages built on top of the above:
        - The Entities Graphics package
        - The Unity Physics package
        - The Netcode for Entities package

False: Using DOTS requires using Entities

While some parts of DOTS depend upon Entities (Entities Graphics, Unity Physics, and Netcode for Entities), the other parts do not. In fact, games built just with GameObjects but no Entities can often make very effective use of Burst-compiled jobs along with the Mathematics and Collections packages.

Not only can Burst-compiled jobs often greatly alleviate key bottlenecks, they are usually relatively painless to retrofit into an existing codebase. In contrast, retrofitting code built around GameObjects to use Entities commonly requires an invasive rewrite.

> **Note:** Currently, work is under way to improve interop and integration between Entities and GameObjects, which should make Entities much easier to selectively retrofit late into a project. However, these features won't ship in Unity 6.x.

False: Using DOTS will make any Unity game significantly faster

Even among games with CPU bottlenecks, many games suffer from inefficiencies that DOTS does not directly address. For example, it's quite common for games to get bogged down unnecessarily by heavy-weight UI, thanks to slow UI layout or excessive overdraw. In such cases, the whole game might run slow even if the core game simulation is perfectly adequate to hit the game's performance targets. The only solution, then, is to fix the parts of your game that are actually too slow!

On the other hand, many games end up with excess costs spread throughout their code and throughout each frame, such that it's very hard to identify clear opportunities for optimization. This kind of 'death by a thousand cuts' easily arises in a codebase where performance was not a key concern from the very start and closely monitored throughout development. Building a project with DOTS can't guarantee that you'll avoid this pitfall, but it can greatly help.

False: Every new Unity project should use Entities

As discussed just above, there are many games where the potential performance benefits of Entities (or DOTS more broadly) won't necessarily make a big difference. Maybe your game is really only GPU-limited, or maybe, on the whole, your game just isn't that demanding for your target hardware. Also as mentioned above, Burst-compiled jobs are easier than Entities to retrofit late into your project.

The hard question, then, is whether to build the foundation of your game on Entities. In short, there are five major reasons to do so:

- **Your game will have an ambitious scale simulation.**
- **Your game will have ambitious scale environments.**
- **Your game needs netcode with client-side prediction.**
- **You want to use Unity Physics.**
- **You prefer ECS as a way of writing code.** *(See the next section.)*

> **Note:** Currently, only Netcode for Entities provides client-side prediction, which is important for certain kinds of fast-action netcode multiplayer games. However, client-side prediction for GameObjects is a feature currently in development, as part of a broader long-term initiative to unify Netcode for GameObjects and Netcode for Entities.

False: The benefits of ECS are just about performance

Though Object-Oriented Programming is what's familiar to most programmers today, the procedural style of ECS often makes code and data easier to design and reason about.

For elaboration, see the DOTS E-book appendix sections called "Data-oriented Design", "Costs of OOP", and "Structural costs of OOP".

False: The performance benefit of Entities / ECS is all about memory efficiency and cache utilization

Memory efficiency is definitely a key consideration for high-performance code, and good cache utilization is an important part of that: the more the CPU has to go out to main memory instead of finding what it needs in the cache, the more cycles the CPU is going to spend waiting instead of doing work.

Unity Entities stores entity components in blocks of memory called "chunks", which are organized into sets called "archetypes" (details here). This layout lends itself to good memory efficiency and cache utilization, but Unity Entities also has other performance advantages over GameObjects and typical OOP code:

- Instantiation and destruction of entities is considerably cheaper than for GameObjects: entities have much lower memory footprints and require much less bookkeeping, and so can be created and destroyed much more efficiently, especially *en masse*. Unlike with GameObjects, pooling entities is never really necessary.
- Whereas GameObjects and their components are always managed objects, entities and their components are unmanaged (meaning they are manually allocated rather than managed by the garbage collector). So entities not only minimize allocation costs, they avoid garbage collection overhead.
- Because entity components are unmanaged, they can be accessed in Burst-compiled code and jobs, which often provide enormous performance gains.
- Entity queries allow for easy, fast processing *en masse*. They help avoid the one-at-a-time anti-pattern common in OOP that can create bottlenecks and be very difficult to optimize.

So yes, the potential performance gains of Entities are partly about better cache utilization, but cache is far from the only performance consideration. In fact, for many typical game scenarios that involve only dozens or hundreds of things to process (rather than thousands or millions), these other factors are often more significant.

False: Entities are the ultimate, optimal data structure for everything

A core principle of Data-oriented Design is that different problems require different solutions. The corollary of this is that overly abstract, generalized solutions are sub-optimal, especially for performance.

Entities, in a sense, are a very generic data structure, and despite being flexible, they cannot be optimal for all possible use cases:

- While hierarchical relationships, such as in transform hierarchies, *can* be expressed with entities, doing so is neither convenient nor particularly efficient.
- Similarly, ordered relationships between entities cannot be conveniently or efficiently expressed.
- While entities are much more compact and cheaper to instantiate than GameObjects, it wouldn't make sense to represent, say for example, every vertex of a mesh as an individual entity, just as it wouldn't make sense to represent them as individual GameObjects. The only sensible way to store mesh vertices is in tightly-packed arrays, and the same is true for many other kinds of data.

So the best way to think of entities is that they are a good *default*, but not ideal for everything.

In the context of a general-purpose game engine:

- entities can serve as a common data protocol between various parts of the engine
- entities are uniformly inspectable and editable in the editor
- entities lend themselves to easy, fast serialization

In your own game code, you generally won't go too wrong modeling your data as entities. On the other hand, you should keep an eye out for cases that warrant just simple arrays or lists. Likewise, you should consider using trees, hashmaps, lists, graphs, or other more specialized structures for data with complex inter-relationships.

Keep in mind that these alternative data structures can always be stored indirectly in components of your entities. For example:

- A tree can be stored as a BlobAsset that is referenced from an entity.

- An array or list can be stored as a DynamicBuffer component of an entity.
- A managed C# object can be referenced from an entity as a UnityObjectRef field of a component.

Referencing all of your data structures from entities in this way will generally make them easier to work with in your ECS code. For example, instead of having to somehow pass a hashmap around to your systems, your systems can just grab the reference to the hashmap from a queried entity.

False: Multi-threaded programming is too hard for most programmers

Multii-threaded programming is an intimidating topic, and even experienced programmers struggle to get it right. There are two main challenges:

- **Thread synchronization.** The access of data and other resources which are shared between threads must be carefully coordinated, usually by means of spinlocks, semaphores, and other synchronization primitives. Failure to do this coordination correctly can produce race conditions, deadlocks, and other bugs.
- **Thread management.** The threads themselves must be created and destroyed, and work must be farmed out to the threads. Failure to do this intelligently can undermine the potential performance gains of multi-threading.

Unity's job system greatly simplifies these concerns: the job system itself is responsible for managing threads, and the job safety checks allow even inexperienced programmers to easily write correct, multi-threaded code without manual synchronization.

False: Manual memory management is too hard for most programmers

If, like many programmers these days, you only have experience in garbage collected languages, having to manually manage memory may seem like a daunting prospect. This fear, however, is mostly misplaced, as manual memory management is very tractable once you get the hang of a few key patterns. Conveniently, DOTS provides a safe environment for you to code without a garbage collector for the first time:

For starters, when using Unity Entities, the entities are manually allocated, but the allocations and deallocations are handled for you when you create entities and destroy entities or add components and remove components. You are still responsible for destroying the entities which

are no longer needed, but generally this is a natural part of your game logic. Although keeping entities alive too long technically counts as a memory leak, in practice, such mistakes manifest as bugs in your game logic, so they tend to be very visible and relatively easy to track down.

Aside from entities, DOTS also provides unmanaged collection types and a set of arena allocators. For a large majority of cases, proper allocation and deallocation of these collections is handled by just picking the right allocator. Does the data in question need to live for the full duration of the game? Use the Persistent allocator. Does the data just need to live for the duration of the current frame? Then use the Temp or WorldUpdate allocators. *Etc.*

Lastly, the DOTS disposal safety checks will catch most cases where you fail to properly allocate or deallocate a collection. So even when you make a mistake, the source of the problem is usually very easy to track down and rectify.

False: GameObjects and Entities cannot be used together (Mostly) False: Entities cannot animate, emit sound, or do UI, etc…

Adding Unity Entities or any other DOTS package to a project does nothing to change how GameObjects and other Unity features operate, and in fact, most any “Entities/ECS project” will rely upon GameObjects for various aspects of Unity functionality. For example, Unity’s UI solutions all depend upon GameObjects, so an Entities project with UI will still need some GameObjects.

There is currently no Unity-provided solution for directly animating entities or having them emit sounds, but the desired end result can generally be achieved indirectly by coordinating your entities with GameObjects. For example, an animated character can be represented in your simulation as an entity that has an associated GameObject which does the actual animation and rendering. This requires you to sync the transform and animation state between the entity and GameObject, which incurs some hassle and overhead, but the cost is usually not a big deal as long as you don’t have hundreds or thousands of characters.

> **Note:** A new Unity animation system is in development which will support both entities and GameObjects. With this new system, entities will be directly animatable without use of GameObjects.

More generally, use of some GameObjects isn’t necessarily a major performance problem as long as the GameObjects stay numbered in the tens or hundreds rather than thousands or beyond, and as long as the computationally heavy lifting is handled by entities and Burst-compiled jobs. Typically, you should prefer using entities for your core simulation logic, physics, and most rendering, but then use GameObjects for a few parts of presentation, such as animation, sound, and UI.

(Mostly) False: DOTS code cannot use managed objects

As described in the previous section, there are cases where you'll need to coordinate between entities and GameObjects or other managed objects. However, the key restriction of jobs and Burst-compiled code is that they cannot access managed objects. Consequently, coordination of GameObjects and Entities requires you to either:

- access entities from GameObject code
- or access GameObjects from an ECS system that isn't Burst-compiled.

Generally, accessing entities from GameObject code is discouraged because GameObject updates do not coordinate with the ECS job safety checks. Besides, the alternative is generally a cleaner code pattern.

> **Note:** Technically, a managed object *can* be accessed in a job, but doing so is only safe if you 'pin' the object for the duration of the job. Doing this is generally not worth the hassle, as the whole point of jobs is to maximize CPU performance, which means you basically always want your jobs to be Burst-compiled: because Burst-compiled code cannot access managed objects, there's rarely a good reason to access managed objects in a job.

> **Note:** For storing managed objects in entity components, you can use managed components or the more recent feature, UnityObjectRef (which is generally the preferred option going forward).