**Abstraction** is the process of removing complex code from the scripts where other programmers will see it, and only exposing the functionality other programmers really need. When you “abstract out” the details, you reduce duplicate code and provide easy access to the most useful methods. The goal of abstraction is to keep your code as clean as possible, and simple for other programmers (or yourself!) to use. In our Rabbit example above, the Hop method would be a good example of abstraction. Without the Hop method, if a programmer wanted the Rabbit to move, they would have to write code that would access its RigidBody and set its velocity, hop height, and so on. With the Hop method, all of those aspects are abstracted away, allowing the programmer to focus on when the Rabbit should hop rather than how.

**Inheritance** is the process of creating a primary class from which other classes, referred to as child classes, can be created. A child class takes on all of the features of the primary, or parent class, automatically. This reduces the need to rewrite code that both classes would need to make use of. As an example, let’s say that we wanted to create a new class called Bunny. Like the Rabbit class, it has ears that can be floppy or standing, has fur that can be a specific color, and can hop. Without inheritance, you’d essentially have to copy all of the code you already wrote and paste it into the new class. With inheritance, you simply extend the Rabbit class, and that functionality is already there and accessible for the Bunny class. The Bunny class could then go on to feature Bunny-specific functionality, such as eating Bunny chow.

**Polymorphism** is one of the most useful aspects of using inheritance. It allows you to create alternative functionality for code that's been inherited from a parent class. As an example, our child Bunny class represents a domesticated Rabbit. It’s able to hop just as a Rabbit is able to, but a Bunny should probably be a little bit slower than its wild parent counterpart. With polymorphism, you can override the contents of the Hop method and write custom code that’s unique to the Bunny. The method call remains the same, but the correct code will be called based on which entity it was called on.

**Encapsulation** is similar to abstraction in that its overarching purpose is to separate the programmer from code complexity, but the focus here is more on code safety in the form of accessibility. Encapsulation gives you tools to code for other programmers, and make sure that they only use your variables and methods as intended. In encapsulated code, other programmers can’t easily change the values of variables or the properties of objects. It’s impossible to account for all of the different ways that other scripts might access your code, so it's far better to encapsulate what you’ve created so it can only perform as intended. As an example, let’s say that our Rabbit’s ear type affects its hearing ability. Once that value is set, it shouldn’t be changed later on. To ensure that the value is protected, you would set it as private, preventing any outside scripts from accessing it.

**What is data persistence?**

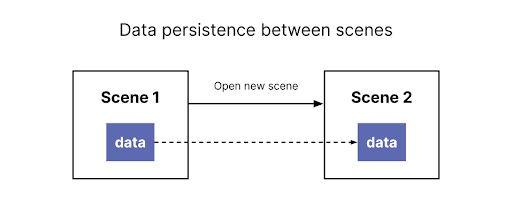
Q&A (0)

**Data persistence** means making data last longer than the process used to create that data. Some examples of data persistence are:

* The specific player icon you select in a one-off multiplayer game (for example, a quiz party game)
* Your name input at the start of an application and then displayed throughout the session (for example, in a polling tool)
* Your progress in ongoing long-form games (for example, an RPG game on a console)
* Your work in a word processing application

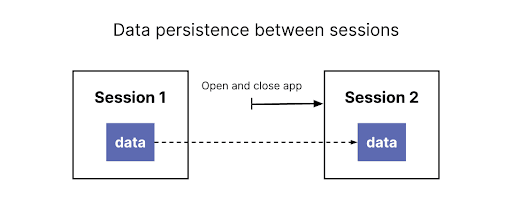
**Data persistence between scenes**

In Unity, data created within a scene is easily available within that scene. But when a user moves to another scene, what happens? Typically, this data is lost. Data persistence between scenes is the process of transferring data from scene to scene to give the user a consistent experience as they progress through your application. In the first two examples above are examples of data persistence between scenes — they are typically one-session experiences that use multiple scenes.



**Data persistence between sessions**

The other two examples (ongoing games and a word processing application) are typically multi-session experiences. Users want to save the progress they make during one session, and then restore it to pick up where you left off. These are example of **data persistence between sessions**.



You will frequently encounter both types of data persistence in more complex applications — data that needs to follow the user as they move between scenes, and that also needs to be saved and restored over multiple sessions.

Your brief for the resource management simulation project includes making it possible to select colors in the initial menu (Menu scene) and apply them to the forklifts in the warehouse (Main scene).

At the moment, the buttons exist, because we’ve provided a custom script for this — you reviewed these when you set up other buttons in the previous tutorial. Now let’s implement data persistence between scenes by making the selected color data available in the Main scene.

To achieve this, you will use:

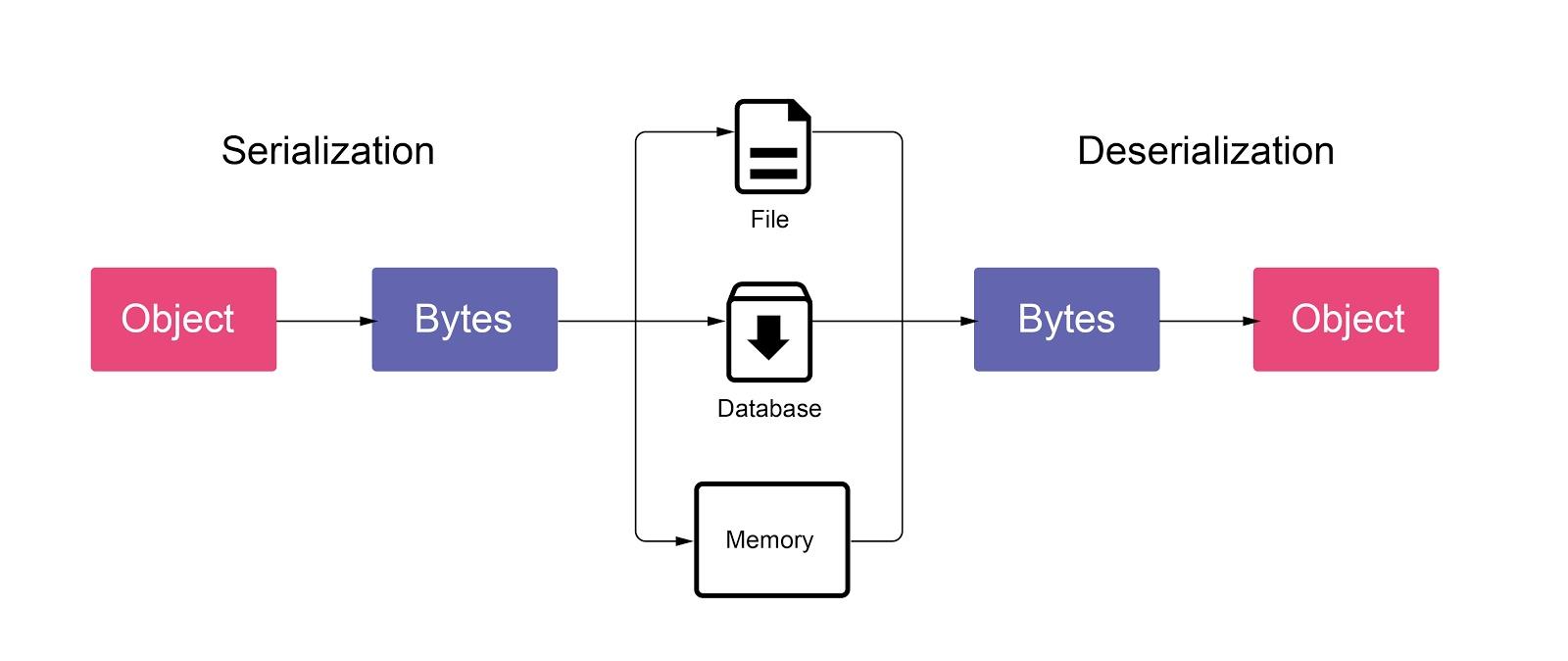
* **DontDestroyOnLoad**: A method in Unity that marks a GameObject to be saved in memory even when loading or unloading a new scene.
* **Static classes** and **class members**: Static class members can be accessed from anywhere without having to reference a specific object. You may already have used some of these, such as **Time.deltaTime** or **Vector3.forward**. Those aren’t a specific time object or a specific vector3 — the format for static class members is ClassName.memberName.

**How can data persist between sessions?**

Q&A (0)

For data to persist between sessions, it needs to be stored in some way. In your case, you’ll need to convert the color your user selects into a format that can be stored and then read when they load the application again.

The process of converting complex data into a format in which it can be stored is called **serialization**. When you’re ready to access the data again, the process of converting it back is called **deserialization**.



There are different formats that you can use to store data, depending on what the data is and what you want to do with it. In this case, you’re going to use the **JSON** format.

## What is JSON?

Q&A (0)

JSON is a text format used to store data and exchange it between platforms. It was first developed for the web, and its full name is **J**ava**S**cript **O**bject **N**otation. It’s based on JavaScript but is language independent — you can use it whether you’re writing C# code or using any other programming language. The JSON format stores data in the form of a **key:value** pair. The key is a string, and the value can be:

* a number
* a string
* a Boolean (true/false)
* an array of values
* Another JSON object

##### How does this format work?

The following JSON string encodes an object storing basic information about a person:

{

"name": "John",

"age": 27,

"address": {

"streetAddress": "21 2nd Street",

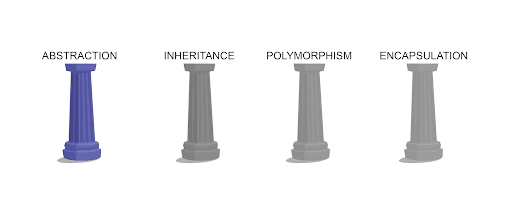
"city": "New York"

}

"pet": [“dog”, “cat”]

}

Let’s break down the example:

* Each **object** is surrounded by curly braces ( **{ }** ).
* Each **entry** is in the form of a key:value pair, separated by a comma.
* The entry “pet” is an array of strings — array values are listed between square brackets ( **[ ]** ).
* The value associated with the entry “address” is between curly braces too — this is because it is another JSON object.
* **What is Abstraction?**
* Q&A (0)
* 
* The first pillar of OOP is all about keeping your code clean and simple for the programmer using it, whether that is you or someone else. Abstraction is the process of removing complex code from the scripts where other programmers will see it, and only exposing the functionality other programmers really need. When you “abstract out” the details, you reduce duplicate code and provide easy access to the most useful functions. You’re actually already quite familiar with this pillar, because you’ve benefited from it frequently throughout this pathway. Whenever you call on a method to perform a task rather than writing out all of the code by hand, you’re benefiting from abstraction!
* Let’s take a look at a situation where you already used abstraction. During[Create with Code. In Unit 4 - Gameplay Mechanics](https://learn.unity.com/project/unit-4-gameplay-mechanics), you created some functionality to spawn an enemy wave.