ABOUT

You're looking, right now, at an interview project documentation. I don't have experience writing those, so I apologise if something strikes you as off.
Firstly, project specifications, if you already know those you can skip to the next section:

- There should be a player character that is able to walk and interact with the game world.
- Assuming the game is of the simulation genre, similar to that of 'The Sims' and 'Stardew Valley', please create a functional clothes shop with all of the features you imagine would be required. It should be a top-down view like Stardew Valley and not a sidescroller, platformer or birds-eye view.
- Features required: Talking to the shopkeeper, buying/selling items, item icons, item prices as well as equipping the bought outfits.

EXPLANATION

This project is logically separated into 3 main categories: inventory, trade and ui.

INVENTORY

Inventory system here uses ScriptableObject system. It allows to easily add new and modify existing objects without any coding expertice. Objects are divided by types, inheriting from an abstract ItemObject. This example project has only one object type – CostumeObject, but to add new ones you just need to create new ItemObject-inherited classes.

Inventories are created as ScriptableObject's as well. The main reason for this is being designer-friendly – creating and, potentially, tweaking new Inventories is incredibly simple. MonoBehaivior Inventories would clutter the inspector menu of every object they attached to, and plain C# classes are hard to tweak without opening an IDE.

Inventories do not store objects directly – instead, they use a wrapper class named InventorySlot and put it in a List. Primary reason for using those for handling stacks of objects, adding different amounts etc.. I considered using a Dictionary with ItemObjects as keys and their amount as values, and, with very large inventorie sizes this approach would be more performant. However, I found InventorySlot approach more readable and flexible – allowing for easily accesible and different methods for adding/removing items later down the line.

More visually, it goes like this:

Inventory (InventorySlot(ItemObject), amount)

Adding and removing objects from inventories requires a refence to the the InventoryObject and is achieved through simple Add(ItemObject) and Remove(ItemObject) methods. They are safe and check for various errors(for example, trying to remove a non-existent object from inventory). If I had more time I would like to expand the system with TryAdd() and TryRemove() methods, not allowing to add an object if it already exists, for example.

This system is simple, flexible and allows for multiple inventories to exist at the same time. You still need an additional logic to add items to them, of course, and that's how we transition to

TRADE

Trade, logically, is simple – it's an exchange of goods for money.
Let's start with money.
For the purposes of this project, I assumed only player requires money to trade. Money and methods to work with them exist in a Wallet singleton. I justify it being a singleton by ease of access and the mentioned before "only player has money".

The trade itself is handled through ShopTradeHandler class. It stores in itself 2 references – to the player inventory, and the shop inventory, and has methods with transfering items from one to another(or, more accurately, removing an object from one and adding to another. It's not a problem for this project, since all the objects are generic and don't change, but if they can change this system will require modification). It also handles required money transactions. For the purposes of this project it is assumed prices for buying and selling the same item are the same, but it can be easily changed.

It is assumed that the reference for the shop inventory will change based on the game context, but as there is only one for now, the reference is set through the Unity Editor.

There isn't much else here, as most of the systems are called from the

UI

I don't have much experience with UI programming, so everything I did I had to learn on the run. Despite this, the whole thing is relatively simple.

There is an InventoryDisplay class, which primary job consists of drawing the

Inventory grid. It's the same for the player and the shop, but can be given different values in each instant in the Inspector window,

InventoryDisplay works pretty similar to the InventoryObject, in a sense that it creates InventorySlot objects and stores the required data in it. First, it creates a grid of InventorySlotUIs, with them being empty by default, then goes through connected InventoryObject and passes InventorySlot's data to InventorySlotUI, which then assigns icon and number representing the item amount in stack by itself. It than listens for changes in InventoryObject and updates it's Slots if needed.

Connected InventoryObject can be changed though inspector before runtime, or through code after the game starts, by calling SetOwner() method. This is useful then you have multiple stores, for example, or adding different kinds of inventories – chests, containers and so on.

InventorySlotUI is the thing which holds a lot of logic – it changes it's appearenes based on passed InventorySlot, and it also uses UnityEvents for checking then pressed or hovered over with cursor. A lot of inventory and trade logic relies on those events – trading, equipping, etc.
It's usually simple – TradeHandler, for example, listens when you click on an icon, and then performs relevant operation – buys if you clicked on shop inventory or sells if you clicked on players.
It's okay for now, but won't scale well later and I wished I had more time to think of a better implementation.

OTHER

There are, of course, other systems. I'll cover them quickly:

- Equipment – for the purposes of this project it's assumed there is only one equipment slot, but it's relatively easy to implement new ones by adding new ItemObject types. Equiping something is done through right-clicking on an item in player inventory, and the costume is simply a sprite drawn on top of a player.
- Movement is done by simply changing rigidbody velocity. I pulled the script from a jam game I made few months ago, with unnececary stuff removed.
- There is simple player state controller, but it need work.
- Dialogue system works through DialogueHandler singleton, which simply contains methods for displaying dialogue and buttons text.
- Interactions are started by pressing E key. Interactables are detected if they have TriggerCollider2D and a component implementing IInteractable interface, put in the list on TriggerEnter and removed on TriggetExit. Then the E key is pressed all interactables in the list are interacted with. Logic can easily be changed to interact with the closest/latest interactable if needed.