**Implementing a Deep Object Comparator**

In my approach to building a deep comparator for objects, I wanted to handle comparisons of both primitive data types and complex objects in an efficient manner. The goal was to ensure that two objects could be compared deeply, meaning that all the data members would be compared, including collections like lists and arrays, regardless of the order of elements inside the collection.

Handling Basic Known Types

For primitive types like int, double, or bool or collections like arrays, lists, and dictionaries, a direct equality comparison can be applied. However, for collections one more thing is done:

**- Lists and arrays**: Are deeply compared by their elements. If the order matters, they are compared element by element. But here as order doesn’t matter we are comparing the count and then finding each element in first in second Collection and either mark the element as visited/already seen or remove the element from list and end the loop/ return false –

If element is not found or

if there are elements present in second list but all elements in first list are already seen or no more elements to traverse

[Using Set and optimizing as there can be duplicate objects may be found with different references].

**- Dictionaries:** Are compared based on their keys and values, ensuring that each key in one dictionary has an equivalent value in the other.

Comparing Complex Objects

Getting the Data Members

When comparing objects, the key is to extract and inspect the internal data members—both fields and properties. Using reflection, I can dynamically access an object's structure at runtime and retrieve its fields and properties.

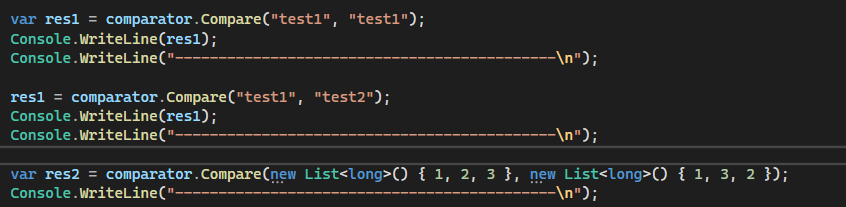
- Fields: Fields are the direct variables declared inside the class. Using reflection, I get the fields with the GetFields method from the Type class.

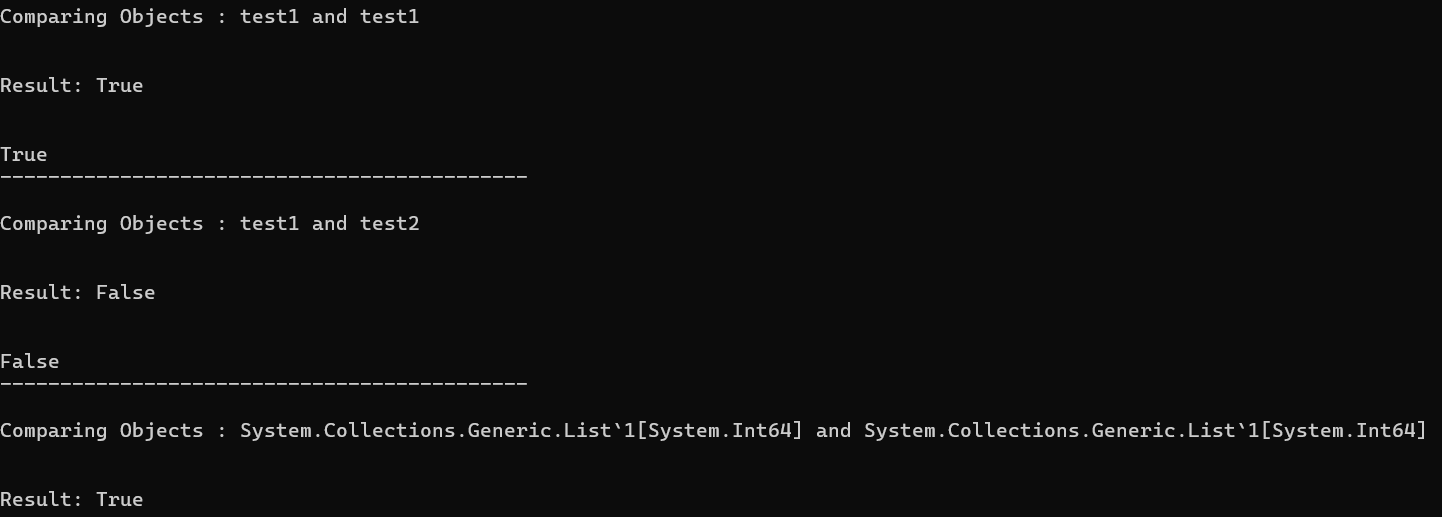
- Properties: Properties, being the encapsulated members of a class, need to be accessed with GetProperties. The challenge here is ensuring that only readable properties (i.e., those with getters) are evaluated.

For each of these fields and properties, I compare the values recursively. When comparing objects of complex types (i.e., user-defined types), I handle the comparison deeply by reflecting over the internal fields and properties and invoking the comparator recursively. The key here is to handle nested structures gracefully, ensuring that each field or property is compared down to the most basic type. This recursive process ensures that even nested collections, dictionaries, or objects within objects are thoroughly compared.

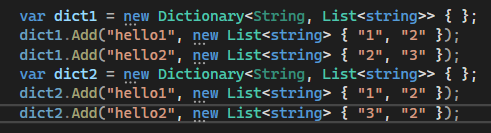
Example :

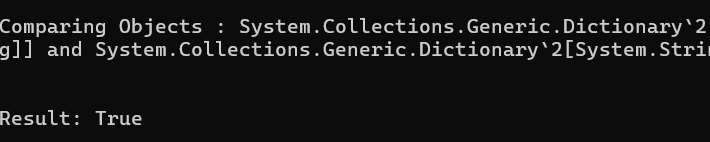
1.



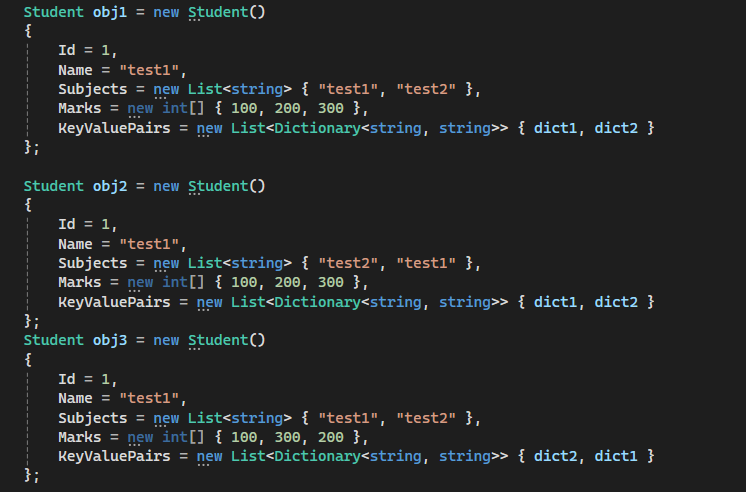


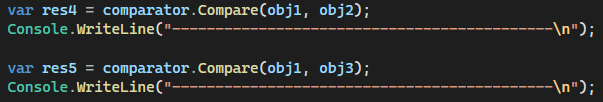
2.

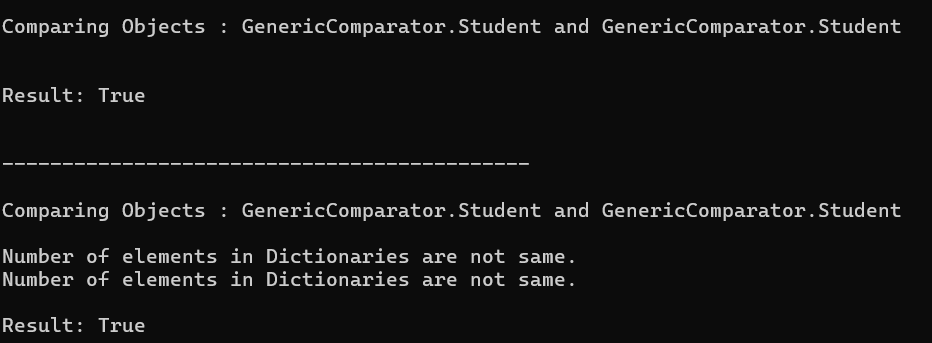




3.







4.

