27. What is a Dictionary?

Brief summary: A Dictionary is a data structure representing a collection of key-value pairs. Each key in the Dictionary must be unique.

A Dictionary is a data structure representing a collection of key-value pairs. Each key in the Dictionary must be unique.

Here is a Dictionary representing the mapping from the country name to its currency:

```
var currencies = new Dictionary<string, string>();
currencies["USA"] = "USD";
currencies["Japan"] = "JPY";
currencies["Brazil"] = "BRL";
```

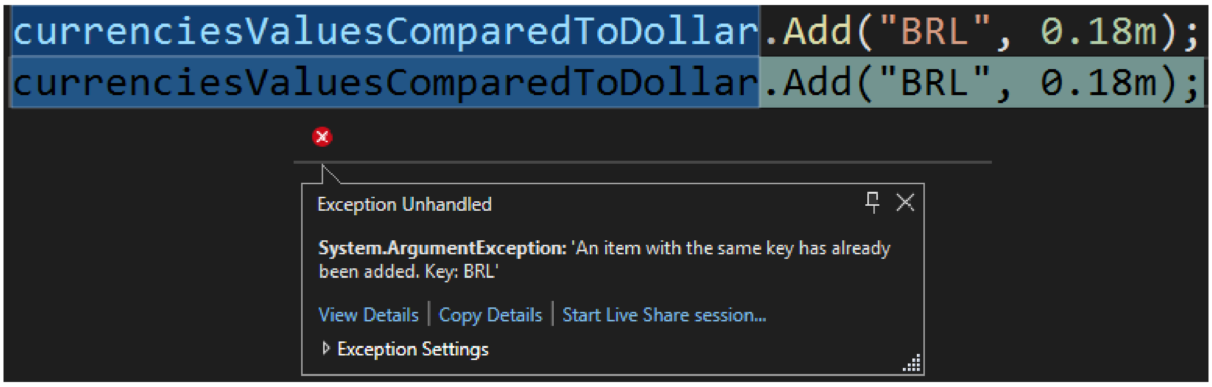
The key and the value in a dictionary don't need to be of the same type. Below we have a Dictionary mapping from string to decimal. You can also see the Add method, which is an alternative for setting a value under each key with the indexer:

```
var currenciesValuesComparedToDollar = new Dictionary<string, decimal>();
currenciesValuesComparedToDollar.Add("USD", 1m);
currenciesValuesComparedToDollar.Add("JPY", 0.0086m);
currenciesValuesComparedToDollar.Add("BRL", 0.18m);
```

We can use the collection initializer instead of adding the key-value pairs to the Dictionary one by one:

```
var savedGames = new Dictionary<string, string>
{
    ["save1"] = @"C:/saves/save1.dat",
    ["autosave"] = @"C:/saves/auto/save.dat",
    ["beforeBossFight"] = @"C:/saves/beforeBossFight.dat",
};
```

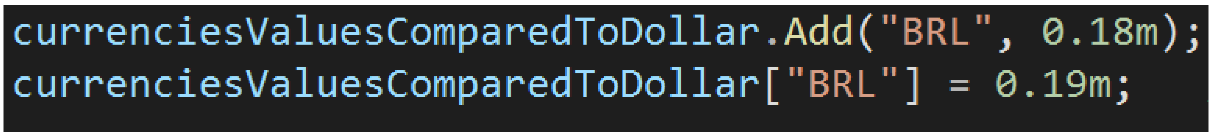
Remember that Dictionary's keys must be unique. That means, an attempt to add a new value under the same key will throw an exception:



```
currenciesValuesComparedToDollar.Add("BRL", 0.18m);
currenciesValuesComparedToDollar.Add("BRL", 0.18m);
```

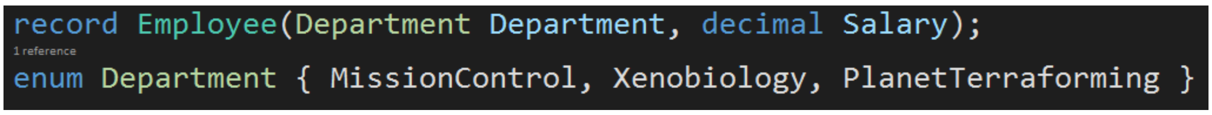
```
Exception Unhandled

System.ArgumentException: 'An item with the same key has already
been added. Key: BRL'

View Details | Copy Details | Start Live Share session...
▷ Exception Settings
```

When using an indexer, the old value under the given key will simply be replaced with the new:

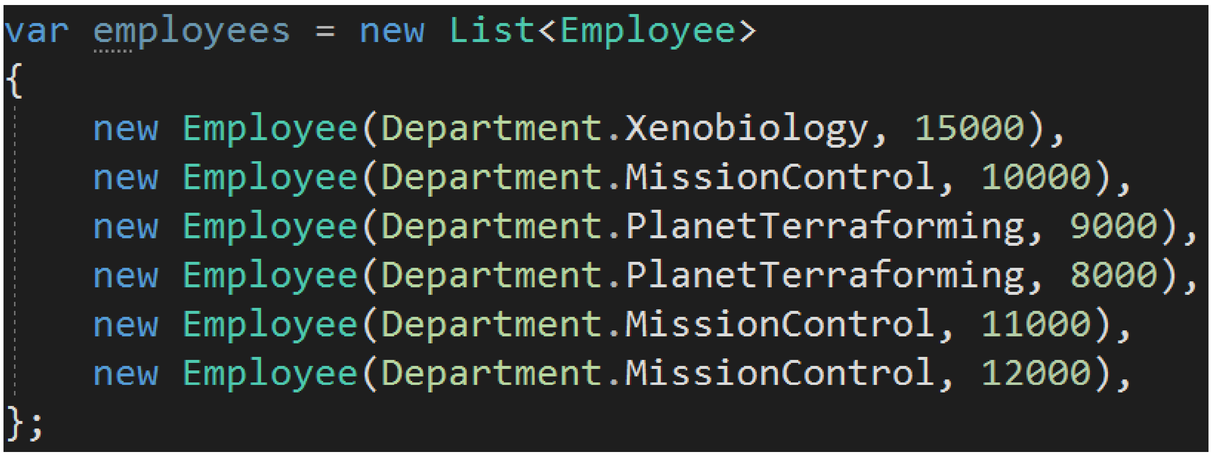


```
currenciesValuesComparedToDollar.Add("BRL", 0.18m);
currenciesValuesComparedToDollar["BRL"] = 0.19m;
```

The use cases for Dictionaries are endless. Whenever we need any kind of mapping, they are most likely the best choice. Let me give you a very simple example. We have a collection of Employees. Each Employee has a Department he or she works in and the Salary property. We want to create a method that calculates what is the average salary in each Department.

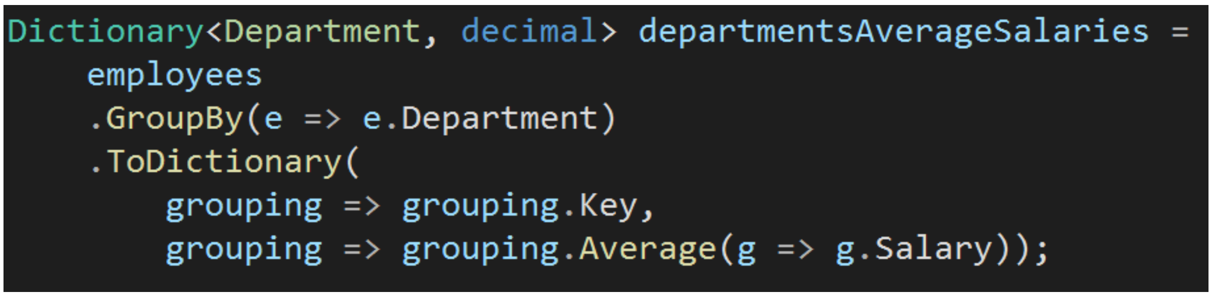


```
record Employee(Department Department, decimal Salary);
1 reference
enum Department { MissionControl, Xenobiology, PlanetTerraforming }
```



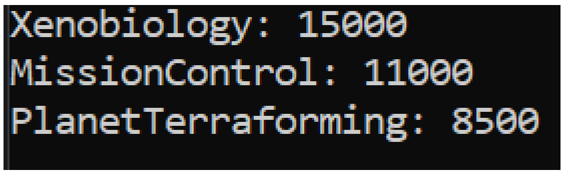
```
var employees = new List<Employee>
{
    new Employee(Department.Xenobiology, 15000),
    new Employee(Department.MissionControl, 10000),
    new Employee(Department.PlanetTerraforming, 9000),
    new Employee(Department.PlanetTerraforming, 8000),
    new Employee(Department.MissionControl, 11000),
    new Employee(Department.MissionControl, 12000),
};
```

The result of this method should be a mapping from Department to the average salary. A mapping is best represented with a Dictionary. I will use LINQ's GroupBy method to group the Employees by department, and then calculate the average salary for each group. I will transform the result into a Dictionary using the ToDictionary method.



```
Dictionary<Department, decimal> departmentsAverageSalaries =
    employees
    .GroupBy(e => e.Department)
    .ToDictionary(
        grouping => grouping.Key,
        grouping => grouping.Average(g => g.Salary));
```

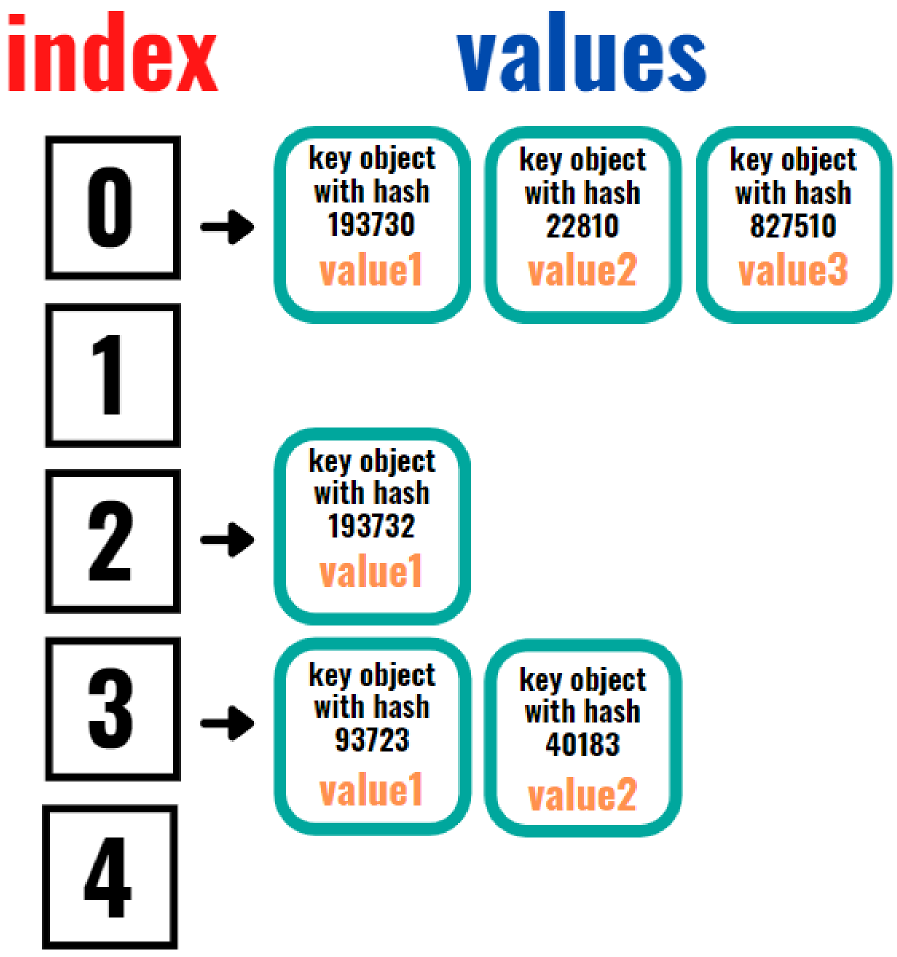
Let’s see if the result is as expected:



```
Xenobiology: 15000
MissionControl: 11000
PlanetTerraforming: 8500
```

The result looks good, and Dictionary is a perfect data structure to represent it.

Now, let’s take a look under the hood of Dictionaries. The underlying data structure of a Dictionary is a hash table. A hash table is basically an **array** of linked lists. We can imagine it like this:


index
values
0
key object with hash 193730 value1
key object with hash 22810 value2
key object with hash 827510 value3
1
2
key object with hash 193732 value1
3
key object with hash 93723 value1
key object with hash 40183 value2
4

Each element in the list has a **value** and the **key** object for which the hash code is calculated. The placing of the key-value pair is not random in the array. The index is calculated like this:

hash code % array size

This, naturally, gives a number from 0 to array size minus one, which is a valid index.

When an item is inserted into the hash table, its hash code is calculated (we learned about hash codes in the “What is the purpose of the GetHashCode method?”). Then, the index in the array is calculated using the above formula. Finally, the key-value pair is added to the list stored under the given index. This means, under a certain index objects with different hash codes can be stored.

But what happens if we add a new key-value pair to the Dictionary, and the key has the same hash code as some other key that is already stored in it? Well, the Dictionary must ask this: is this actually the same key, or is it a different key that accidentally has the same hash code? There is one method that can answer this question: the **Equals method**.

If two keys have the same hash codes, and the **Equals** method returns **true** for them, it means it’s actually the same key. Then the dictionary will either throw an exception (if the key-value pair was added with the Add method, which expects that this key is not yet present in the Dictionary) or, if it was set with the indexer, it will simply update the value under the key.

But if the new and the old key have the same hash codes, but the **Equals** method returns **false** for them, it means there are actually two different keys, that have the same hashcode by accident. In this case, the Dictionary stores the new key-value pair under the same index, and it adds it at the end of the linked list that is stored under this index. When we try to retrieve the value under this key again, the Dictionary will quickly calculate the hash code, and based on this hash, the index in the array of linked lists. Then, it will iterate the list, looking for an object with the same key - so the key for which the Equals method returns true if compared with the key for which we try to retrieve the value.

Because the **Equals** method is needed in case of hashcode conflicts to properly identify the key, we should **always override it when overriding the GetHashCode** method, so their implementations are consistent. For example, if GetHashCode returns the social security number for a Person object, it means we consider this number the Person's identifier. The Equals method should also only compare the social security numbers.

Calculating the hashcode is (or at least should be) very fast. Accessing the array element at a given index is extremely fast. This means, as long as the list under each index is small, accessing the Dictionary value under a given key should be super fast, and this is the main power of Dictionaries.

So, how big are the lists stored under each index? Well, it depends on two factors:

- what is the size of the array that represents the hash table (this is something we don't control, the Dictionary itself adjusts the size similarly as List did)
- how often the hash codes for different objects are the same. If, for example, we implemented the GetHashCode method for some type as "return 1", it would mean that all hashcodes will be duplicated. There will be only one element in the array representing the hash table, and this element will be a very long list of key-value pairs. In other words, in this particular case, the performance of the Dictionary will be similar to the performance of a List. This is the reason for which the hash functions should be uniformly distributed. The fewer hash code conflicts, the better the Dictionary's performance.

Let's summarize. A Dictionary is a data structure representing a collection of key-value pairs. Each key in the Dictionary must be unique. When a key is added to the Dictionary, it calculates its hash code using the GetHashCode method. It uses this hashcode to properly place the value for the given key in the hash table that is the underlying data structure of a Dictionary.

Bonus questions:

- "**What is a hash table?**"
  A hash table is a data structure that stores values in an array of collections. The index in the array is calculated using the hash code. It allows quick retrieval of objects with given hashcode. A hash table is the underlying data structure of Dictionary.

- "**Will the Dictionary work correctly if we have hash code conflict for two of its keys?**"
  Yes. The Dictionary still can tell which key is which using the Equals method, so it will not mistake them only because they have the same hash codes.

- "**Why should we override the Equals method when we override the GetHashCode method?**"
  Because the Equals method is needed for the Dictionary to distinguish two keys in case of the hash code conflict, and because of that its implementation

should be in line with the implementation of the GetHashCode method. For example, if GetHashCode returns the social security number for a Person object, it means we consider this number the Person’s identifier. The Equals method should also only compare the social security numbers.