The Mapping Type

Mappings

Copy page

Use the mapping data type to store key-value pairs.

In Solidity, the hashtable/hashmap/dictionary-comparable type used to store key-value pairs is called a `mapping`. `mapping`s are a powerful tool with many uses, but they also have some unexpected limitations. They also **aren't** actually hash tables!

Objectives

By the end of this lesson you should be able to:

- Construct a Map (dictionary) data type
- Recall that assignment of the Map data type is not as flexible as for other data types/in other languages
- Restrict function calls with the `msg.sender` global variable
- Recall that there is no collision protection in the EVM and why this is (probably) ok

Mappings in Solidity vs. Hash Tables

On the surface, the `mapping` data type appears to be just another hash table implementation that stores pairs of any hashable type as a key, to any other type as a value. The difference is in implementation.

In a more traditional implementation, the data is stored in memory as an array, with a hash-to-index *(hashmod)* function used to determine which spot in the array to store a given value, based on the key. Sometimes, the *hashmod* function for two different keys results in the same index, causing a *collision*.

Collisions are resolved via additional solutions, such as linked list chaining; when the underlying array starts to get full, a bigger one is created, all the keys are re-hash-modded, and all the values moved over to the new array.

In the EVM, `mappings` do **not** have an array as the underlying data structure. Instead, the `keccak256` hash of the key plus the storage slot for the mapping itself is used to determine which storage slot out of all 2**256 will be used for the value.

There is no collision-handling, for the same reason that makes wallets work at all - 2**256 is an unimaginably large number. One of the biggest numbers you might encounter regularly is the number of possible configurations for a **shuffled deck of cards**, which is:

80658175170943878571660636856403766975289505440883277824000000000000

Meanwhile, the number of variations of a `keccak256` hash are:

115792089237316195423570985008687907853269984665640564039457584007913129639935

Collisions are very unlikely.

As a result, there are a few special characteristics and limitations to keep in mind with the `mapping` data type:

- Mappings can only have a data location of `storage`
- They can't be used as parameters or returns of public functions
- They are not iterable and you cannot retrieve a list of keys
- All possible keys will return the default value, unless another value has been stored

Creating a Mapping

Create a contract called `Mappings`. In it, add a `mapping` from an `address` to a `uint` called `favoriteNumbers`.

Reveal code

Writing to the Mapping

Add a function called `saveFavoriteNumber` that takes an `address` and `uint`, then saves the `uint` in the mapping, with the `address` as the key.

Reveal code

Deploy and test it out. Does it work? Probably...

You don't have a way to read the data in `favoriteNumber`, but this problem is easy to correct. Similar to arrays, if you mark a `mapping` as public, the Solidity compiler will automatically create a getter for values in that `mapping`.

Update the declaration of `favoriteNumbers` and deploy to test again.

Utilizing msg.sender

Another issue with this contract is that a `public` function can be called by anyone and everyone with a wallet and funds to pay gas fees. As a result, anyone could go in after you and change your favorite number from lucky number **13** to anything, even **7**!

That won't do at all!

Luckily, you can make use of a **global variable** called `msg.sender` to access the `address` of the wallet that sent the transaction. Use this to make it so that only the owner of an `address` can set their favorite number.

Reveal code

Deploy and test again. Success!

Retrieving All Favorite Numbers

One challenging limitation of the `mapping` data type is that it is **not** iterable - you cannot loop through and manipulate or return **all** values in the `mapping`.

At least not with any built-in features, but you can solve this on your own. A common practice in Solidity with this and similar problems is to use multiple variables or data types to store the right combination needed to address the issue.

Using a Helper Array

For this problem, you can use a helper array to store a list of all the keys present in `favoriteNumbers`. Simply add the array, and push new keys to it when saving a new favorite number.

Reveal code

To return all of the favorite numbers, you can then iterate through `addressesOfFavs`, look up that addresses' favorite number, add it to a return array, and then return the array when you're done.

Reveal code

On the surface, this solution works, but there is a problem: What happens if a user **updates** their favorite number? Their address will end up in the list twice, resulting in a doubled entry in the return.

A solution here would be to check first if the `address` already has a number as a value in `favoriteNumbers`, and only push it to the array if not.

Reveal code

You should end up with a contract similar to this:

Conclusion

In this lesson, you've learned how to use the `mapping` data type to store key-value pairs in Solidity. You've also explored one strategy for solving some of the limitations found in the `mapping` type when compared to similar types in other languages.

Using msg.sender
How Mappings are Stored