**Creating your own Libraries**

**Introduction:**

In the previous lesson, we used the `getPrice()` function and `getConversionRate`. These methods can be reused multiple times for anyone working with Price Feeds. When a functionality can be commonly used, we can create a library to efficiently manage repeated parts of codes.

**Libraries:**

Great examples of Libraries can be found in the Solidity by example (<https://solidity-by-example.org/library/>) website.

Solidity libraries are similar to contracts but do not allow the declaration of any state variables and cannot receive ETH.

**Important:** All functions in a library must be declared as `internal` and are embedded in the contract during compilation. If any function is not marked as such, the library cannot be embedded directly, but it must be deployed independently and then linked to the main contract.

We can start by creating a new file called `PriceConverter.sol`, and replace the `contract` keyword with `library`.

// SPDX-License-Identifier: MIT

Pragma solidity ^0.8.18;

Library PriceConverter {}

Let’s copy `getPrice`, `getConversionRate`, and `getVersion` functions from the `FundMe.sol` contract into our new library, remembering to import the `AggregatorV3Interface` into `PriceConverter.sol`. Finally, we can mark all the functions as `internal`.

// SPDX-License-Identifier: MIT

Pragma solidity ^0.8.18;

Import {AggregatorV3Interface} from “@chainlink/contracts/src/v0.8/shared/interfaces/AggregatorV3Interface.sol”;

Library PriceConverter {

Function getPrice() internal view returns (uint256) {

AggregatorV3Interface priceFeed = AggregatorV3Interface(0x694AA1769357215DE4FAC081bf1f309aDC325306);

(, int256 answer, , , ) = priceFeed.latestRoundData();

Return uint256(answer \* 10000000000);

}

Function getConversionRate(uint256 ethAmount) internal view returns (uint256) {

Uint256 ethPrice = getPrice();

Uint256 ethAmountInUsd = (ethPrice \* ethAmount) / 1000000000000000000;

Return ethAmountInUsd;

}

}

**Accessing the Library:**

You can import the library in your contract and attach it to the desired type with the keyword `using`:

Import {PriceConverter} from “./PriceConverter.sol”;

Using PriceConverter for uint256;

The `PriceConverter` functions can then be called as if they are native to the `uint256` type. For example, calling the `getConversionRate()` function will now be changed into:

Require(msg.value.getConversionRate() >= minimumUsd, “didn’t send enough ETH”);

Here, `msg.value`, which is a `uint256` type, is extended to include the `getConversionRate()` function. The `msg.value` gets passed as the first argument to the function. If additional arguments are needed, they are passed in parentheses:

Uint256 result = msg.value.getConversionRate(123);

In this case, `123` is passed as the second `uint256` argument to the function.

**Conclusion:**

In this lesson, we explored the benefits of using \_libraries\_ to reuse code and add new functionalities. We created a `PriceConverter` library to handle `getPrice`, `getConversionRate`, and `getVersion` functions, demonstrating how to structure and utilize libraries effectively.

**Updates:**

Changes on Chainlink import Reminder

Last updated on November 19, 2024

As mentioned on previous lessons, the import for VRF changed on the 2024 edition, so be sure to consider the changes.

From:

Import {AggregatorV3Interface} from “@chainlink/contracts/src/v0.8/interfaces/AggregatorV3Interface.sol”;

To:

AggregatorV3Interface} from “@chainlink/contracts/src/v0.8/shared/interfaces/AggregatorV3Interface.sol”;

So be sure to add the `shared` on your code.

Take into consideration that this might change with time, so be sure to check Chainlink VRF documentation (<https://docs.chain.link/vrf>) to be sure you are using the correct import.