# Welcome To Solidity Fundamentals

A comprehensive guide to Solidity Fundamentals - This lesson explores the world of Solidity as the most dominant smart contract programming language. Learn how to code and work with Solidity to become a smart contract developer, security researcher, or enter the …

# Best Practices

A comprehensive guide to best practices for using Cyfrin Updraft. The lesson covers how to best utilize the Cyfrin Updraft resources including the Github repo, Discord, and the platform's built-in forums to ensure a better learning experience. You will also learn tips on how to

### Introduction

Welcome to the Cyfrin Updraft. This platform offers a comprehensive learning experience by combining video lessons, written content, and interactive discussions.

### Resources

In the top right corner of the lesson view, you'll find a link to the [**GitHub resources**](https://github.com/Cyfrin/foundry-full-course-cu) 📂 page. This guide contains all the code, information, and materials necessary for the course. On GitHub, the [**discussions tab**](https://github.com/Cyfrin/foundry-full-course-cu/discussions) 💬 allows you to interact with fellow learners and the community, where you can ask questions, share insights, and solve problems collaboratively.

Next to the video lesson, there is a **written lesson** tab 📝. This tab provides the course content in text format, which is useful for reading along, copy-pasting code snippets or reviewing updates.

If you're watching on YouTube, links to these resources are available in the video description. However, watching this course on Cyfrin Updraft offers additional benefits like tracking your progress and accessing written lessons, which will enhance your learning experience.

### Best Practices for Learning

1. 🌐 ***Engage with the Community*** by using GitHub discussions and Discord for real-time communication. Remember that asking questions and interacting with others can greatly enhance your understanding and retention of the material.
2. 🏝️ ***Take Regular Breaks***: Avoid cramming all the information in one go because your brain needs time to process and absorb the knowledge.
3. ⏩ ***Adjust the Video Playback Speed***: Use the speed controls in the video player to match the pace of the lesson with your learning speed.
4. 📝 ***Use Subtitles if Needed***: If English isn't your first language, take advantage of the subtitle options available in the video settings.
5. 🔄 ***Follow a Modular Approach***: The course is designed to be flexible. You can jump between topics based on your interests and needs. Revisit lessons to reinforce your knowledge and ensure thorough understanding.
6. 🎯 ***Utilize Quizzes and Coding Challenges***: Cyfrin Updraft offers both quizzes and coding challenges. You'll find quizzes at the end of each section and coding challenges at the end of each GitHub repository section. These will let you earn NFTs as badges of honor for your achievements.
7. ❓ ***Learn to Ask Well-Formatted Questions***: Asking questions is a skill too. Remember this when you are asking clear, concise questions in forums and discussions to get the best help from others.

### 🧑‍💻 Test Yourself

1. 📕 Provide at least 5 recommended best practices to follow throughout this course.

### Conclusion

By following these guidelines, you'll be well-equipped to make the most out of this course. Engaging with the community, pacing your learning, and utilizing Cyfrin Updraft resources will significantly enhance your learning experience.

# Introduction

This lesson provides an introduction to the course, guiding students through accessing and navigating the GitHub repository, understanding the usage of the repository for cloning lesson codes, and engaging in discussions. It also covers the importance of asking questions and

### Introduction

To get started, navigate to the official Cyfrin Updraft [GitHub repository](https://github.com/Cyfrin/foundry-full-course-f23)

👀❗**IMPORTANT**  
Each course will have an associated link, where you’ll find all the **code** that you will be working on within in each lesson and a **README** section, that contains instructions on how to work with the code.

The interface might look slightly different when you first access it. What you're looking for is the repository associated specifically with this lesson. This repository will contain all the code required for this stage of the course, together with a README section. The README will provide you with a wealth of notes on how to work with the code.

### Repository usage

The repository serves two main purposes:

* 🚪 **Easy access:** each lesson can be consulted and cloned effortlessly
* 🗣 👥 **Discussion and Network:** you can engage with fellow students, ask questions, and participate in collaborative learning.

🔥 **CAUTION**  
To raise issues or start discussions based on a specific repository, please use the [**Discussions tab**](https://github.com/Cyfrin/foundry-full-course-f23/discussions) of the Cyfrin Updraft - Career Path instead of creating issues directly on the repository itself.

#### Asking Questions

It’s very likely that during your journey you’ll have questions. It’s recommended to use the [**Q&A**](https://github.com/Cyfrin/foundry-full-course-f23/discussions/new?category=q-a) section provided inside the discussion tab. You will be guided into how to best formulate your doubts and queries, such that have the highest chance of being answered by the community, AI or a forum.

### Setting Up

Before we dive into coding, it is essential that you have access to the code repository and educational resources provided.

Having an account on the following platforms is highly recommended:

* [GitHub](https://github.com/)
* [Stack Exchange Ethereum](https://ethereum.stackexchange.com/)
* [ChatGPT](https://openai.com/blog/chatgpt) (but remember it might not always provide accurate information).
* [Google Gemini](https://gemini.google.com/) (Google's Free GPT Alternative that understands youtube videos for summarization, data extraction and content-seek).

### Conclusion

Now comes the exciting part: building and deploying your first smart contract. We're going to be utilizing a tool called [Remix](https://remix.ethereum.org/), an IDE (Integrated Development Environment) for deploying and interacting with smart contracts. You can access it through this [link](https://remix.ethereum.org/).

💡 **TIP**  
The best way to get the most out of this guide is to **code along**. You're encouraged to change the speed of the tutorial video to match your coding pace. Remember that **space repetition** is critical while building a new skill.

After concluding the next lesson, you'll have already built and deployed your first smart contract to a blockchain. Let's jump right into it!

### 👨‍💻 Test yourself

At the end of each lesson, you will find a Test Yourself section. This part will help you reinforce the concepts you just learned and coded about. There will be theoretical questions - marked with 📕, as well as coding questions -marked with 👨‍💻.

💡 **TIP**  
Be sure that you truly understand the answers before going on to the next lesson.

# Basic variable types

This lesson introduces basic variable types in Solidity, such as Boolean, Uint, Integer, Address, and Bytes. It explains how to define variables in a Solidity contract and their default values, providing a foundational understanding of data types in smart contract programming.

## Solidity Types

Solidity supports various elementary types that can be combined to create more complex ones. You can read more about them in the [Solidity documentation](https://docs.soliditylang.org/en/v0.8.20/types.html#types).

🕵️‍♂️ For now, let's focus on the most commonly used

* Boolean (bool): true or false
* Unsigned Integer (uint): unsigned whole number (positive)
* Integer (int): signed whole number (positive and negative)
* Address (address): 20 bytes value. An example of an address can be found within your MetaMask account.
* Bytes (bytes): low-level raw byte data

### Variables definition

Variables are just placeholders for **values**. A value can be one **data type** described in the list above. For instance, we could create a Boolean variable named hasFavoriteNumber, which would represent whether someone has a favourite number or not (constant true or false).

bool hasFavoriteNumber = true; // The variable `hasFavoriteNumber` represents the value `true`

It's possible to specify the number of **bits** used for uint and int. For example, uint256 specifies that the variable has 256 bits. uint is a shorthand for uint256.

🗒️ **NOTE**  
It's always advisable to be **explicit** when specifying the length of the data type.

The semicolon at the end of each line signifies that a statement is completed.

// SPDX-License-Identifier: MIT

pragma solidity 0.8.19;

contract SimpleStorage {

// Basic types

bool hasFavoriteNumber = true;

uint256 favoriteNumber = 88;

string favoriteNumberInText = "eighty-eight";

int256 favoriteInt = -88;

address myAddress = 0xaB1B7206AA6840C795aB7A6AE8b15417b7E63a8d;

bytes32 favoriteBytes32 = "cat";

}

### Bytes and strings

Bytes are a collection of characters written in hexadecimal representation.

bytes1 minBytes = "I am a fixed size byte array of 1 byte";

bytes32 maxBytes = "I am a fixed size byte array of 32 bytes";

bytes dynamicBytes = "I am a dynamic array, so you can manipulate my size";

Bytes can be allocated in size (up to bytes32). However, bytes and bytes32 represent distinct data types.

**Strings** are internally represented as dynamic byte arrays (bytes type) and designed specifically for working with text. For this reason, a string can easily be converted into bytes.

[Bits and Bytes overview](https://www.youtube.com/watch?v=Dnd28lQHquU)

## The contract logic

📋 Let's explore a scenario where there is a task involving the storage of a favourite number. For this purpose, we can start storing the variable favoriteNumber of type uint:

uint256 favoriteNumber;

👀❗**IMPORTANT**  
Every variable in Solidity comes with a default value. Uninitialized uint256 for example, defaults to 0 (zero) and an uninitialized boolean defaults to false.

## Conclusion

You've just filled in your first smart contract with variables and you explored the fundamental data types in Solidity.

## 🧑‍💻 Test yourself

1. 📕 What's the difference between a variable and a value?
2. 📕 Describe the default value of the following types: bool, uint, int256, string, address, bytes, bytes32
3. 📕 How does uint differ from bytes?
4. 🧑‍💻 Write a smart contract that contains at least five storage variables, each with a distinct data type.

# Functions

This lesson focuses on creating functions in Solidity, specifically a 'Store' function for updating a variable. It explains the syntax and structure of functions, including visibility specifiers, and guides students through deploying and interacting with the smart contract using the Remix

### Introduction

In the previous lesson, we added a storage variable favoriteNumber within our first smart contract and explored different solidity types. In this lesson, you'll discover how to update and retrieve a storage variable, while also learning about functions, visibility, deployment, transactions, gas usage and variable scope.

### Building the **store** function

📋 To store the favoriteNumber variable, we need to implement a new **function**. In Solidity, functions - or methods, are portions of code designed to execute specific tasks within the overall codebase. We'll call this new function store, and it will be responsible for updating the favoriteNumber variable.

contract SimpleStorage {

uint256 favoriteNumber; // a function will update this variable

// the function will be written here

}

Functions are identified by the keyword function, followed by a custom **name** (e.g. "store") and any additional **parameters** enclosed in rounded parentheses (). These parameters represent the values sent to our function. In this case, we inform the store function that we want to update favoriteNumber with some other value \_favoriteNumber:

contract SimpleStorage {

uint256 favoriteNumber; // storage variable: it's stored in a section of the blockchain called "Storage"

function store(uint256 \_favoriteNumber) public {

favoriteNumber = \_favoriteNumber;

}

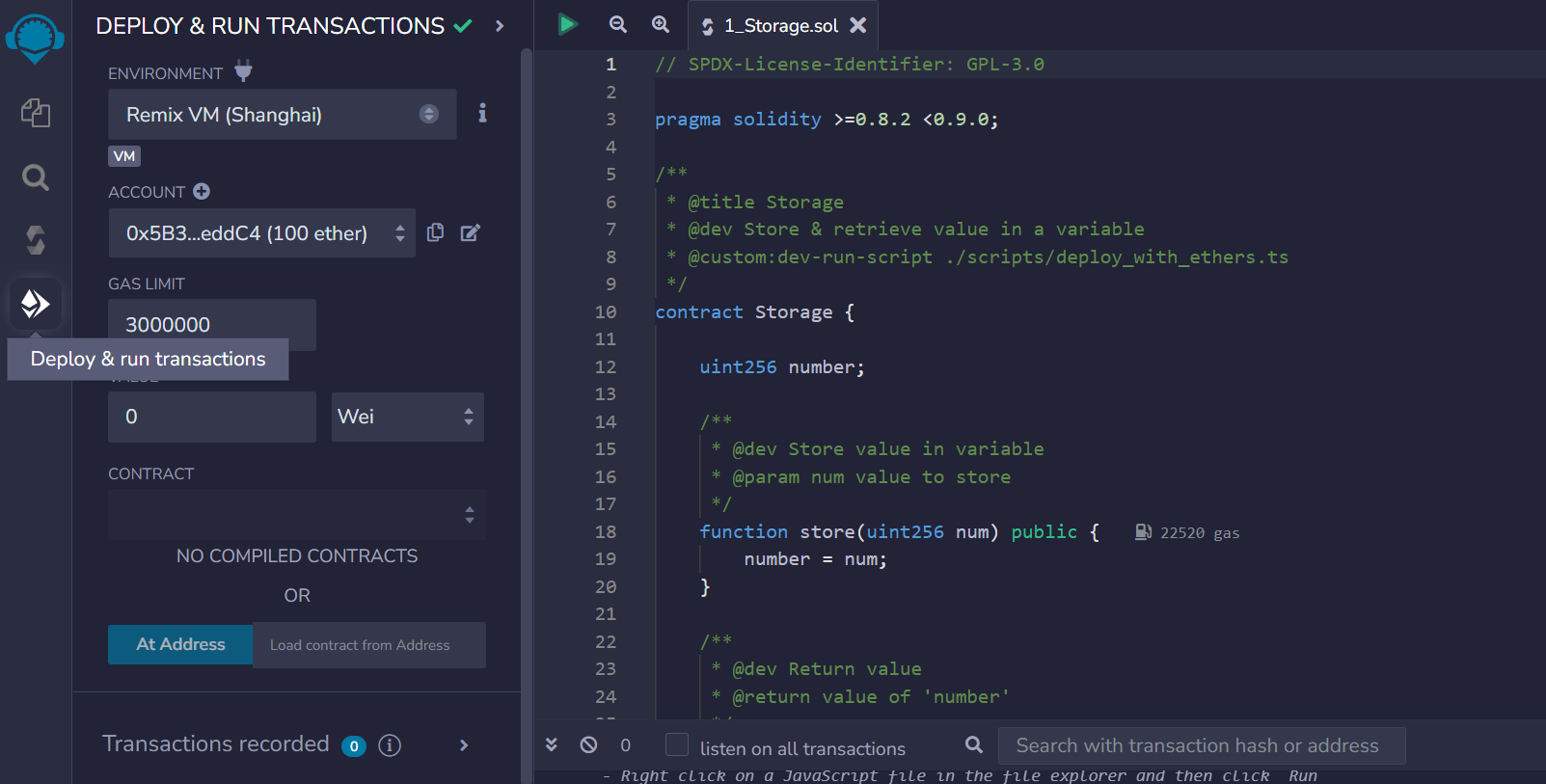
}

The content of the function is placed within the curly brackets {}. The prefix \_ before \_favoriteNumber is used to emphasise that the ***local*** variable \_favoriteNumber is a **different** variable from the ***state*** variable favoriteNumber. This helps prevent potential confusion when dealing with different variables with similar names in complex codebases.

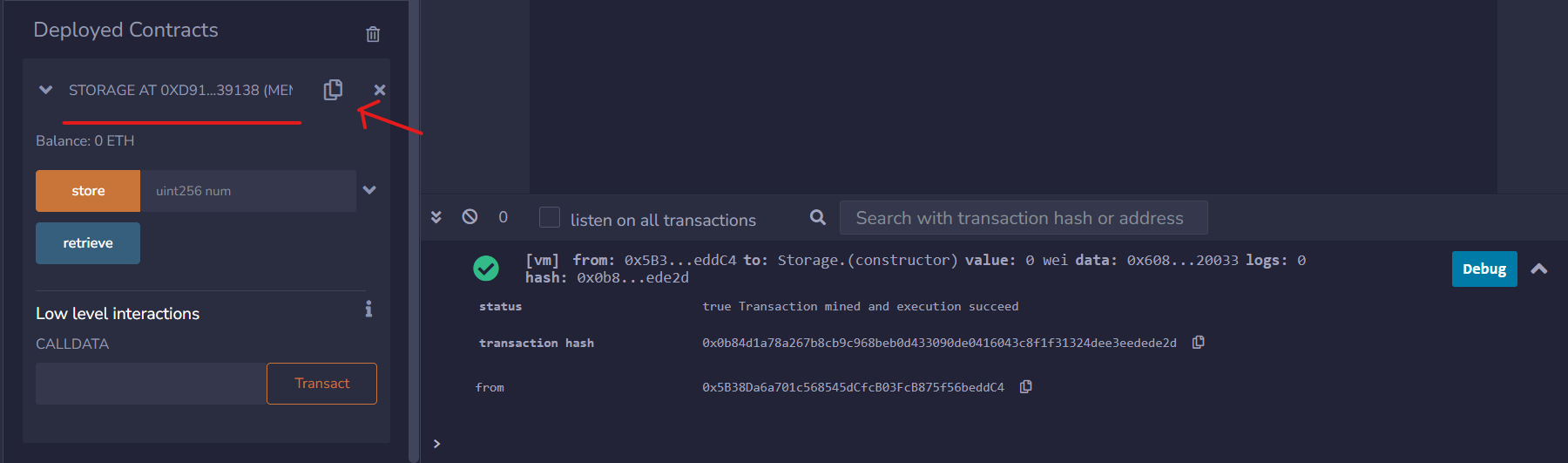
### Deploying the smart contract

You can test out this function in the Remix VM environment. At this stage, you can compile your code by navigating to the compile tab and hitting Compile. After compiling, navigate to the tab **Deploy and Run Transactions** to test your function.

The **Deploy and Run Transactions** tab holds a variety of parameters that are used during the deployment process. You'll be assigned an account with some ETH to deploy your smart contract.



In this environment, your contract is assigned a unique address. You can re-access your deployed contract by expanding the **Deployed Contracts** interface and simultaneously opening the terminal, which shows log data of all contract deployments and transactions.



If we open the Remix terminal we can see that deploying the contract has just sent a simulated transaction on the Remix environment. You can check out its details such as status, hash, from, to and gas.

👀❗**IMPORTANT**  
The process of sending a transaction is the **same** for deploying a contract and for sending Ethers. The only difference is that the machine-readable code of the deployed contract is placed inside the data field of the deployment transaction.

### Transactions creation

Let's send a transaction to the store function to change the value of the variable favoriteNumber: you can insert a number and press the store button in Remix. A transaction is initiated and after some time, its status will change from pending to complete.

💸 From the accounts section, it becomes visible that ETH is being consumed every time a transaction is submitted. When the state of the blockchain is modified (e.g. deploying a contract, sending ETH, ...), is done by sending a transaction that consumes **gas**. Executing the store function is more expensive than just transferring ETH between accounts, with the rising gas expenses primarily associated (though not exclusively) with the code length.

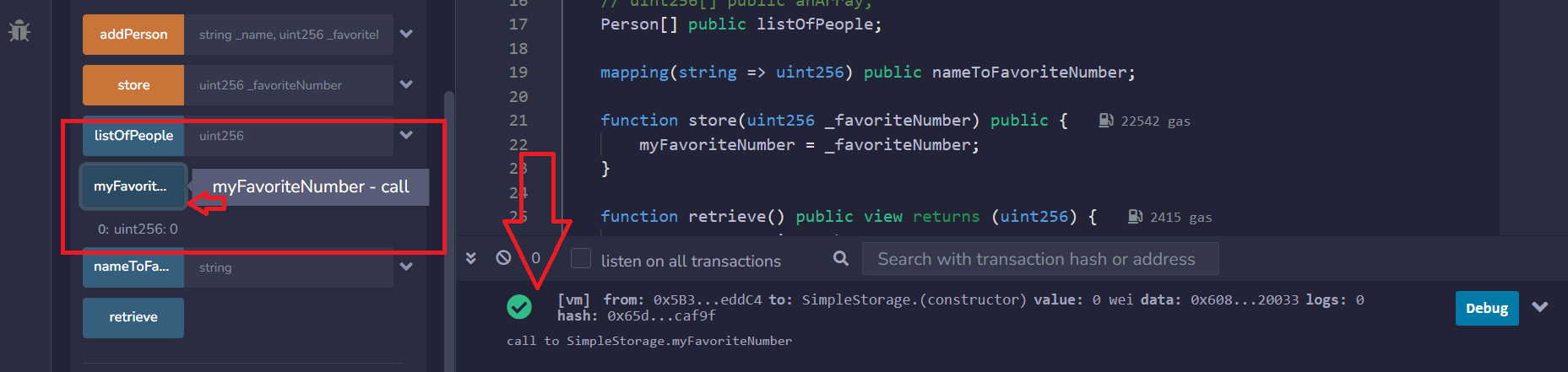
#### Verifying the stored value

This contract is missing a way to check if the number has been updated: now we can store a value but we cannot be sure if the transaction **actually** changed the variable value.

The default visibility of the favoriteNumber variable is **internal**, preventing external contracts and users from viewing it.

🗒️ **NOTE**  
Appending the public keyword next to a variable will automatically change its visibility and it will generate a **getter function** (a function that gets the variable's value when called).

uint256 public favoriteNumber;

After completing compilation and deployment, a button labelled favoriteNumber will become visible. When pressed, it should return the most recent stored value of the variable favoriteNumber.

#### Visibility

In Solidity, functions and variables can have one of these four visibility specifiers:

* 🌎 **public**: accessible from both inside the contract and from external contracts
* 🏠 **private**: accessible only within the current contract. It does not hide a value but only restricts its access.
* 🌲 **external**: used only for functions. Visible only from outside the contract.
* 🏠🏠 **internal**: accessible by the current contract and any contracts derived from it.

If a visibility specifier is not given, it defaults to internal.

#### Pure and View keywords

The terms view and pure are used when a function reads values from the blockchain without altering its state. Such functions will not initiate transactions but rather make calls, represented as blue buttons in the Remix interface. A pure function will prohibit any reading from the state or storage.

function retrieve() public view returns(uint256) {

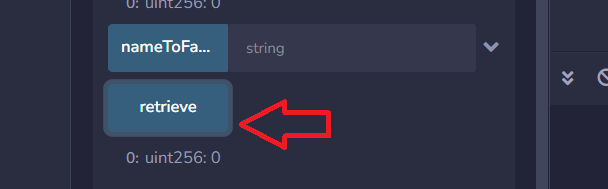
return favoriteNumber;

}

function retrieve() public pure returns(uint256) {

return 7;

}



The keyword returns specifies the type(s) of value a function will return.

🚧 **WARNING**  
While calling view or pure functions doesn’t typically require gas, they do require it when called by another function that modifies the state or storage through a transaction (e.g. calling the function retrieve inside the function storage). This cost is called **execution cost** and it will add up to the transaction cost.

### The scope of a variable

The scope of a variable refers to the **context** within which it is defined and accessible. This context is usually determined by the block of code, typically enclosed in curly braces {}, where the variable is declared. To access the same variable across different functions, it should be declared inside the scope of the main contract.

function store(uint256 \_favoriteNumber) public {

favoriteNumber = \_favoriteNumber;

uint256 testVar = 5;

}

function something() public {

testVar = 6; // will raise a compilation error

favoriteNumber = 7; // this can be accessed because it's in the main contract scope

}

### Conclusion

In this lesson, you have learned how to build a function in Solidity, define its visibility, and understand how it operates on values within a smart contract. You have also explored different transactions and their gas consumption.

### 🧑‍💻 Test yourself

1. 📕 Describe the four visibility keywords and their impact on the code.
2. 📕 What's the difference between view and pure?
3. 📕 In which circumstances a pure function will incur gas costs?
4. 📕 Explain what a scope is and provide an example of an incorrect scope.
5. 📕 What's the difference between a transaction that deploys a contract and a transaction that transfers ETH?
6. 🧑‍💻 Write a contract that features 3 functions:
   * a view function that can be accessed only by the current contract
   * a pure function that's not accessible within the current contract
   * a view function that can be accessed from children's contracts

# Arrays and structs

This lesson explores the use of arrays and structs in Solidity for creating a list of favorite numbers and tying them to individuals. It demonstrates how to create and manipulate arrays and structs, enhancing the functionality of a smart contract to handle multiple data

### Introduction

Up to this point, the SimpleStorage contract allows for storing, updating, and viewing a single favorite number. In this lesson, we'll enhance the code to store multiple numbers, enabling more than one person to store their values. We'll learn how to create a list of favorite numbers using **arrays**, and we'll explore the **struct** keyword for creating new types in Solidity.

### Arrays and struct

First we need to replace the uint256 favoriteNumber with a list of uint256 numbers:

uint256[] list\_of\_favorite\_numbers;

The brackets indicate that we have a list of uint256, an array of numbers. If we want to initialize this array we can do so by specifying its content:

uint256[] list\_of\_favorite\_numbers = [0, 78, 90];

🗒️ **NOTE**  
Arrays are zero-indexed: the first element is at position zero (has index 0), the second element is at position one (has index 1), and so on.

The issue with this data structure is that we cannot link the owner with its favorite value. One solution is to establish a **new type** using the struct keyword, named Person, which is made of two attributes: a favorite number and a name.

struct Person {

uint256 my\_favorite\_number;

string name;

}

🚧 **WARNING**  
Rename the variables favorite\_number to avoid name clashes

From this struct, we can instantiate a variable my\_friend that has the type Person, with a favorite number of seven and the name 'Pat'. We can retrieve these details using the getter function that was generated by the public keyword.

Person public my\_friend = Person(7, 'Pat');

/\* equals to

Person public my\_friend = Person({

favorite\_number:7,

name:'Pat'});

\*/

### Array of struct

Creating individual variables that represent several people might become a tedious task, due to the repetitive steps of the process. Instead of manually instantiating a variable for each person, we can combine the two concepts we just learned about: arrays and structs.

Person[] public list\_of\_people; // this is a dynamic array

Person[3] public another\_list\_of\_three\_people; // this is a static array

When using a **dynamic** array, we can add as many Person objects as we like, as the size of the array it's not static but can grow and shrink. We can access each Person object in our array by its index.

To add people to this list, we can create a function that populates the array:

function add\_person(string memory \_name, uint256 \_favorite\_number) public {

list\_of\_people.push(Person(\_favorite\_number, \_name));

}

add\_person is a function that takes two variables as input - the name and favourite number of the person. It creates first a new Person object and then it pushes it to our list\_of\_people array.

### Conclusion

With these features, our Solidity contract can now store multiple favorite numbers, each associated with a specific person. The add\_person function creates a new Person struct and adds it to the list\_of\_people state variable. We can then view each person's name and favorite number by accessing the Person object through the array index.

### 🧑‍💻 Test yourself

1. 📕 Define the difference between a dynamic array and a static array. Make an example of each.
2. 📕 What is an array and what is a struct?
3. 🧑‍💻 Create a smart contract that can store and view a list of animals. Add manually three (3) animals and give the possibility to the user to manually add an indefinite number of animals into the smart contract.

# Errors and warnings

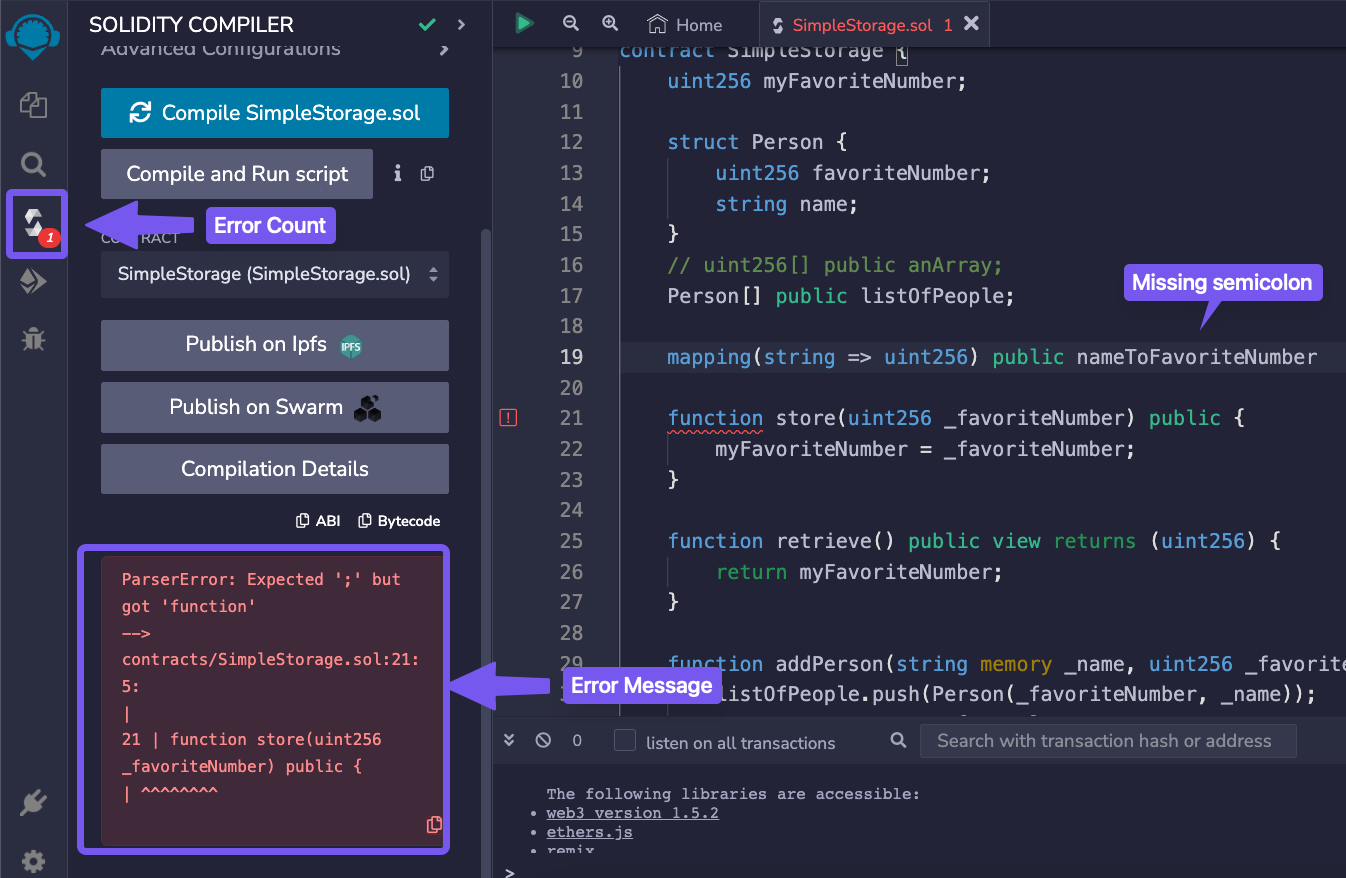
A guide to understanding and resolving errors and warnings in Solidity programming. The lesson covers interpreting the color coding of error messages, leveraging online resources like Phind, and effectively using communities like GitHub discussions and Stack Exchange for

### Introduction

In the previous lesson, we learned how to combine arrays and structs to store information and how to manipulate this information with the function addPerson. This time we'll explore **errors** and **warnings** and how to leverage forums, search engines and AI resources.

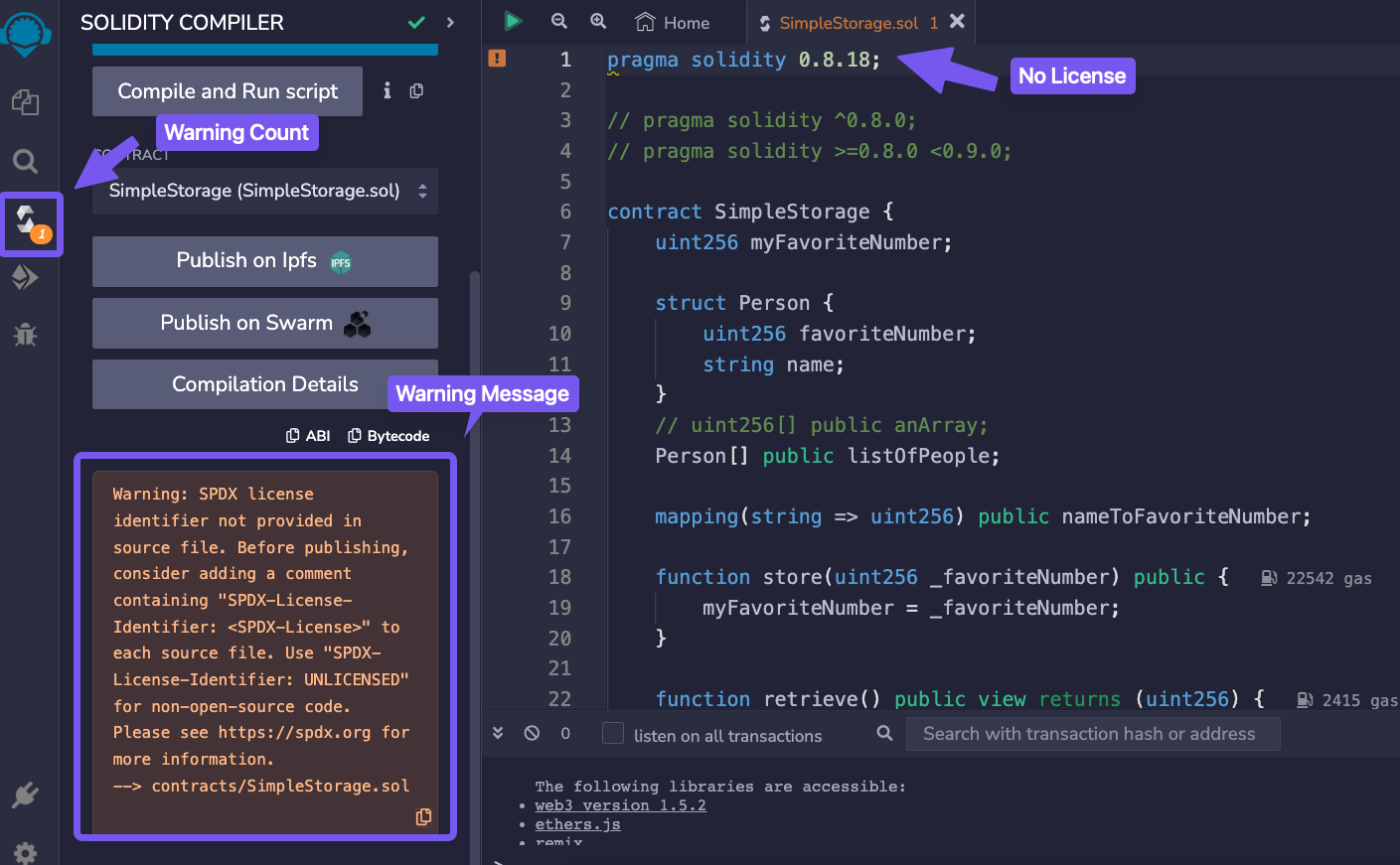
### Errors and Warnings

If we remove a semicolon from the code and then try to compile it, you'll encounter some 🚫 **error messages**. They will prevent the compiler from converting the code into a machine-readable form.



Restoring the semicolon to its correct position will prevent any errors, enabling us to proceed with deploying the code to the Remix VM. On the other hand, if we delete the SPDX license identifier from the top of our code and recompile, we will receive a yellow box showing a ⚠️ **warning**.

> Warning: SPDX license identifier not provided in source file



Unlike errors, **warnings** allow the code to be compiled and deployed but it's wise to take them seriously and aim to remove them entirely. They point out poor or risky practices in your code and sometimes indicate potential bugs.

* If it's <span style="color:red">red</span>, there is a compilation error in the code and it needs to be solved before deployment.
* If it's <span style="color:#808000">yellow</span>, you might want to double-check and adjust your code.

### Leverage your resources

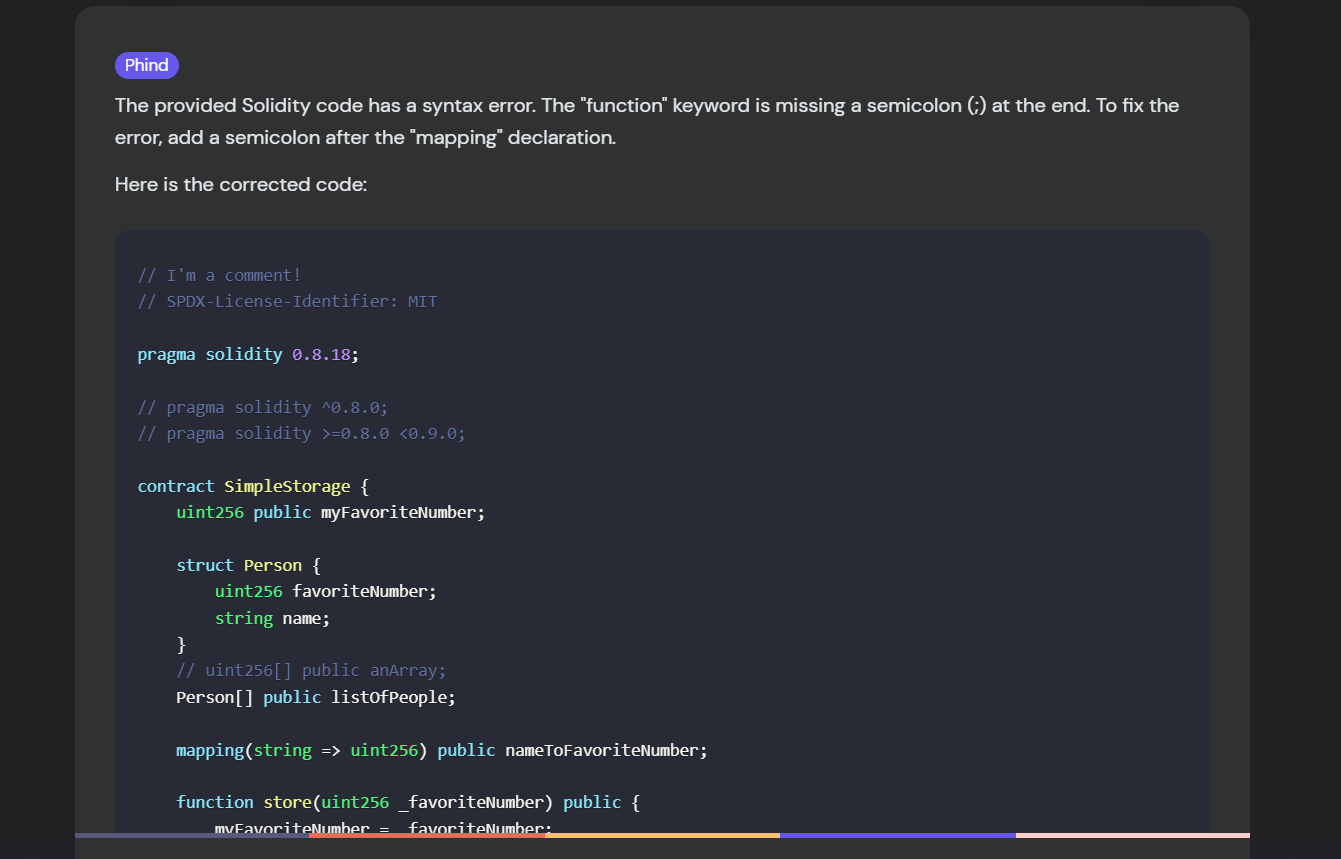
In situations when you do not understand the error that's prompted, using some online resources can make the situation clearer:

* AI Frens (ChatGPT, Phind, Bard, AI Chrome extensions,..)
* Github Discussions
* Stack Exchange Ethereum
* Peeranha

#### Phind

Let's now attempt to resolve the semicolon error we intentionally created before by using [Phind](https://www.phind.com/). Phind is an AI-powered search engine for developers. It operates by first conducting a Google search based on your query, and then parsing the results to give you a contextual response.

We can input the compiler error under the drop-down menu, execute the search, and get a comprehensive explanation of why the error happened and how to fix it.



#### Other resources

It is advised to make active use of AI tools, as they can substantially boost your understanding and skills. Later in this course, we will explore how to ask effective questions, utilize AI prompts, structure your inquiries, and improve your search and learning techniques.

You can also take part of online communities like **GitHub discussions** and **Stack Exchange**, where you'll find valuable insights, answers to your questions, and support from fellow developers.

💡 **TIP**  
One of the most important aspects of being an excellent software engineer or prompt engineer is not just having the information but knowing where to find it.

### Conclusion

You’ve just learned how to effectively identifying and managing errors and warnings, enhancing your ability to maintain robust and reliable code. In the following lesson, we will delve deeper into Solidity’s data locations and some advanced Remix functionalities.

### 🧑‍💻 Test yourself

1. 📕 What's the difference between a warning and an error? Make an example of each.
2. 🧑‍💻 Make a written list (or a bookmark in your browser) with at least 3 useful online resources will help you solve future bugs.

# Memory storage and calldata

An in-depth look at data locations in Solidity, focusing on the differences and applications of 'memory', 'storage', and 'calldata'. The lesson explains these concepts with examples, clarifying their roles in temporary and permanent data storage within smart contracts.

### Introduction

In this section, we will explore how Solidity manages data storage, focusing on the differences between storage, memory, and calldata, and why these concepts are crucial for writing optimized and secure smart contracts.

### Data Locations

Solidity can store data in **six** different locations. In this lesson, we will focus on the first three:

1. Calldata
2. Memory
3. Storage
4. Stack
5. Code
6. Logs

### Calldata and Memory

In Solidity, calldata and memory are temporary storage locations for variables during function execution. calldata is read-only, used for function inputs that can't be modified. In contrast, memory allows for read-write access, letting variables be changed within the function. To modify calldata variables, they must first be loaded into memory.

🚧 **WARNING**  
Most variable types default to memory automatically. However, for **strings**, you must specify either memory or calldata due to the way arrays are handled in memory.

string memory variableName = "someValue";

#### Calldata

Calldata variables are read-only and cheaper than memory. They are mostly used for input parameters.

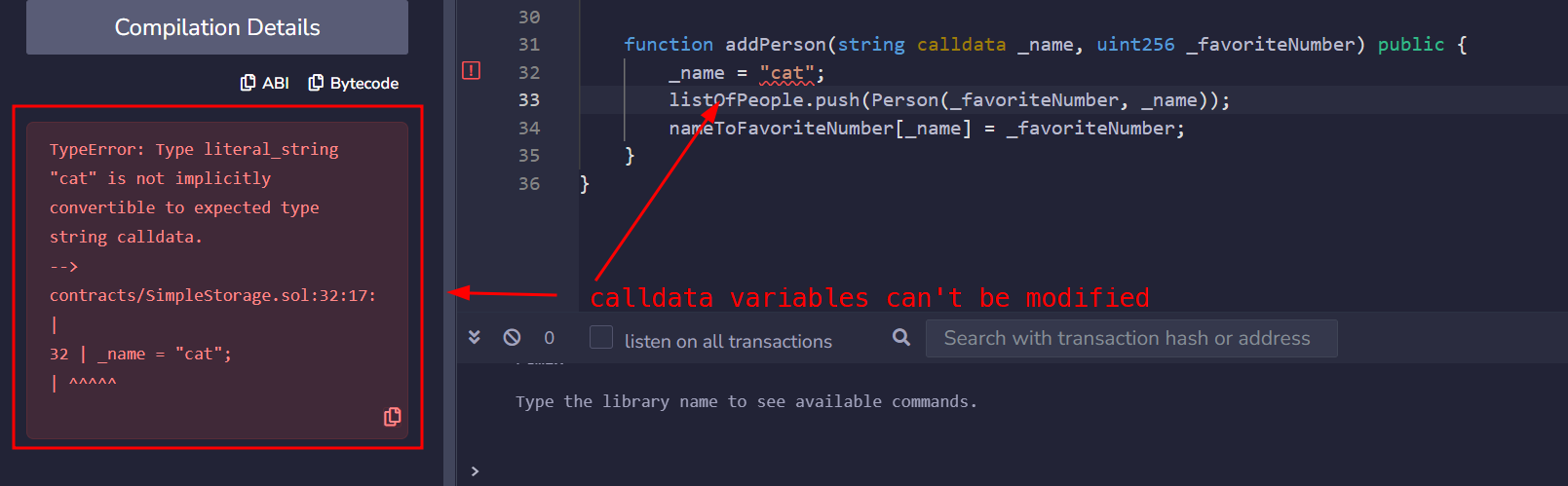
In the following example, if we try to replace the keyword memory with calldata, we receive an error because calldata variables can't be manipulated.

function addPerson(string calldata \_name, uitn256 \_favoriteNumber) public {

\_name = "cat";

listOfPeople.push(Person(\_favoriteNumber, \_name));

}



### Storage

Variables stored in storage are persistent on the blockchain, retaining their values between function calls and transactions.

In our contract, the variable myFavoriteNumber is a storage variable. Variables which are declared outside any function are implicitly converted to storage variables.

contract MyContract {

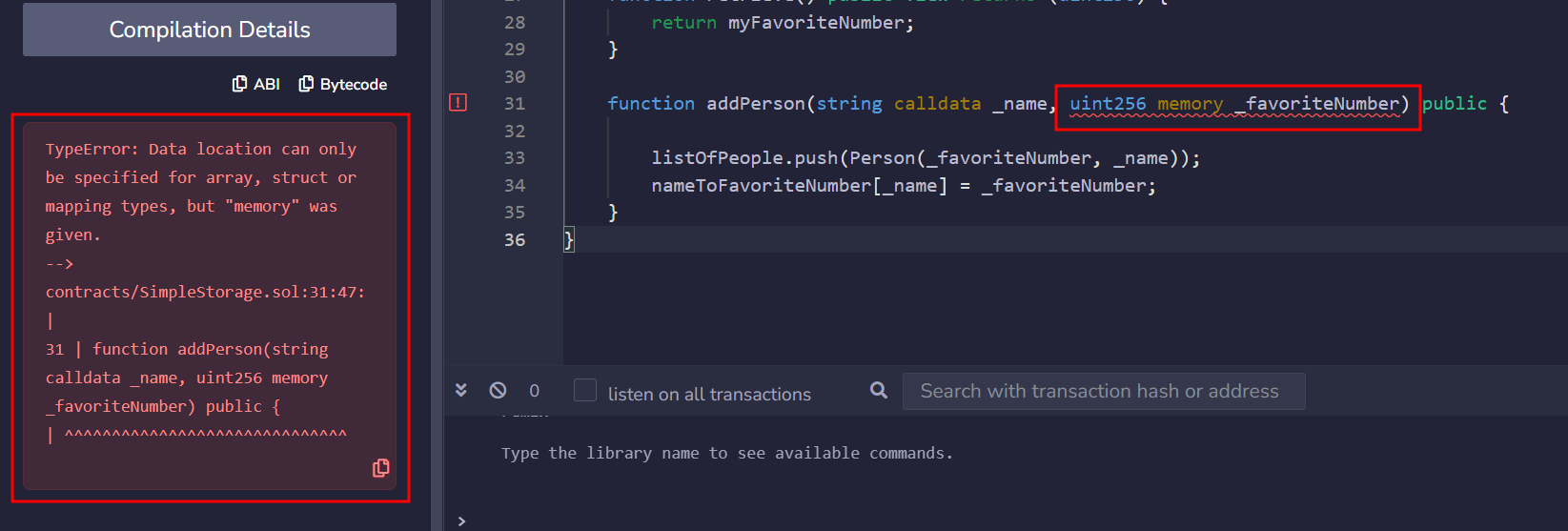
uint256 favoriteNumber; // this is a storage variable

};

### Strings and primitive types

If you try to specify the memory keyword for an uint256 variable, you'll encounter this error:

> Data location can only be specified for array, struct, or mapping type



In Solidity, a string is recognized as an **array of bytes**. On the other hand, primitive types, like uint256 have built-in mechanisms that dictate how and where they are stored, accessed and manipulated.

🚧 **WARNING**  
You can't use the storage keyword for variables inside a function. Only memory and calldata are allowed here, as the variable only exists temporarily.

function addPerson(string memory \_name, uitn256 \_favoriteNumber) public { // cannot use storage as input parameters

uint256 test = 0; // variable here can be stored in memory or stack

listOfPeople.push(Person(\_favoriteNumber, \_name));

}

### Conclusion

Well done! You've learned the differences between the keywords storage, memory, and calldata in Solidity, enhancing your skills to develop robust Ethereum-based applications.

### 🧑‍💻 Test yourself

1. 📕 How does the Solidity compiler handle primitive types and strings in terms of memory management?
2. 📕 Why can't the storage keyword be used for variables inside a function?
3. 🧑‍💻 Write a smart contract that uses storage, memory and calldata keywords for its variables.

# Mappings

This lesson introduces the concept of mappings in Solidity, explaining how they can be used to efficiently link information, such as connecting names to numbers. It demonstrates how to define and use mappings to improve data access in a smart contract.

Iterating through a long list of data is usually expensive and time-consuming, especially when we do not need to access elements by their index.

### Mapping

To directly access the desired value without the need to iterate through the whole array, we can use **mappings**. They are sets of 🔑 (unique) **keys** linked to a 🍱 **value** and they are similar to hash tables or dictionaries in other programming languages. In our case, looking up a name (key) will return its correspondent favorite number (value).

A mapping is defined using the mapping keyword, followed by the key type, the value type, the visibility, and the mapping name. In our example, we can construct an object that maps every name to its favorite number.

mapping (string => uint256) public nameToFavoriteNumber;

Previously, we created an addPerson function that was adding a struct Person to an array list\_of\_people. Let's modify this function and add the struct Person to a mapping instead of an array:

nameToFavoriteNumber[\_name] = \_favoriteNumber;

👀❗**IMPORTANT**  
Mappings have a constant time complexity for lookups, meaning that retrieving a value by its key is done in constant time.

🗒️ **NOTE**  
The default value for all key types is zero. In our case, nameToFavoriteNumber["ET"] equals 0.

### Conclusion

Mapping can be a versatile tool to increase efficiency when attempting to find elements within a larger set of data.

### 🧑‍💻 Test yourself

1. 📕 In which cases is better to use an array instead of a mapping?
2. 🧑‍💻 Create a Solidity contract with a mapping named addressToBalance. Implement functions to add and retrieve data from this mapping.

# Deploying your first contract

A practical guide to deploying a Solidity smart contract on a testnet. The lesson walks through the pre-deployment audit, compilation check, changing the environment, connecting accounts, confirming transactions, and interacting with the deployed contract.

### Introduction

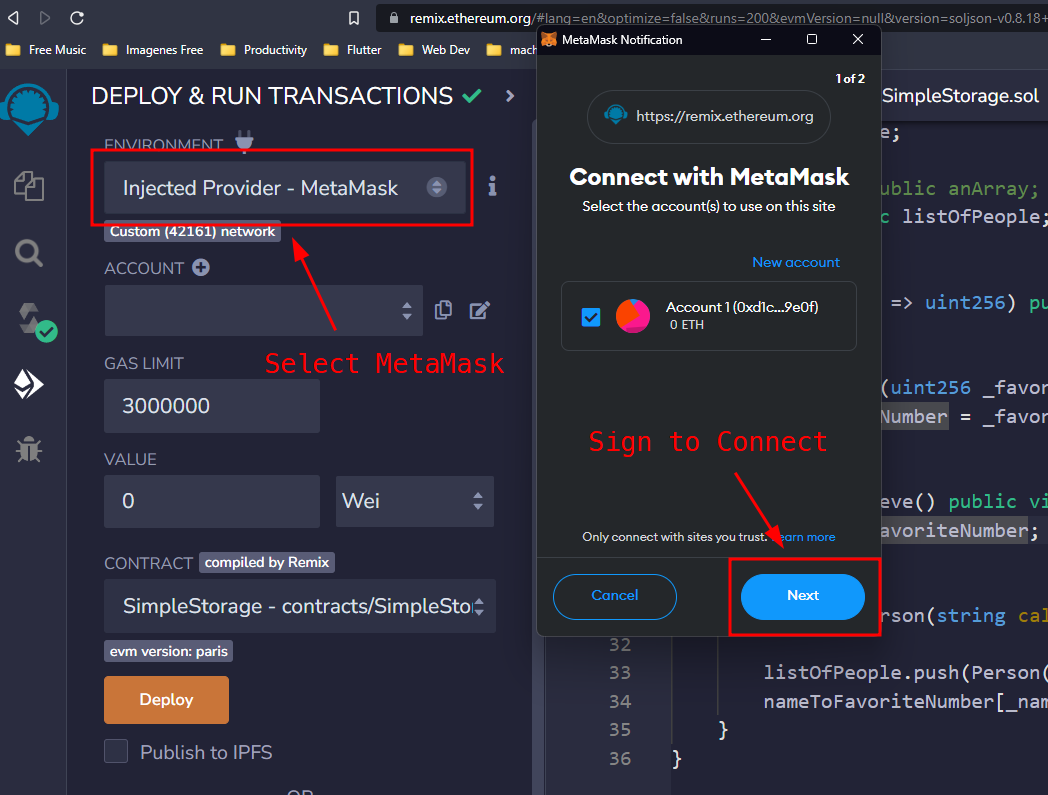
Over the past eight lessons, we crafted the SimpleStorage contract. It defines a custom type Person, includes an internal variable that can be read and updated, and contains a public array and mapping that can also be modified. In this lesson, we will deploy the contract to a **real testnet**, which fully simulates a live blockchain environment without using real Ether.

🔥 **CAUTION**  
You could be tempted to immediately deploy this contract to the mainnet. As a general rule, I caution against this. Make sure to write tests, carry out audits and ensure the robustness of your contract before deploying it to production. However, for the sake of this demonstration, we're going to deploy this as a dummy contract on a testnet.

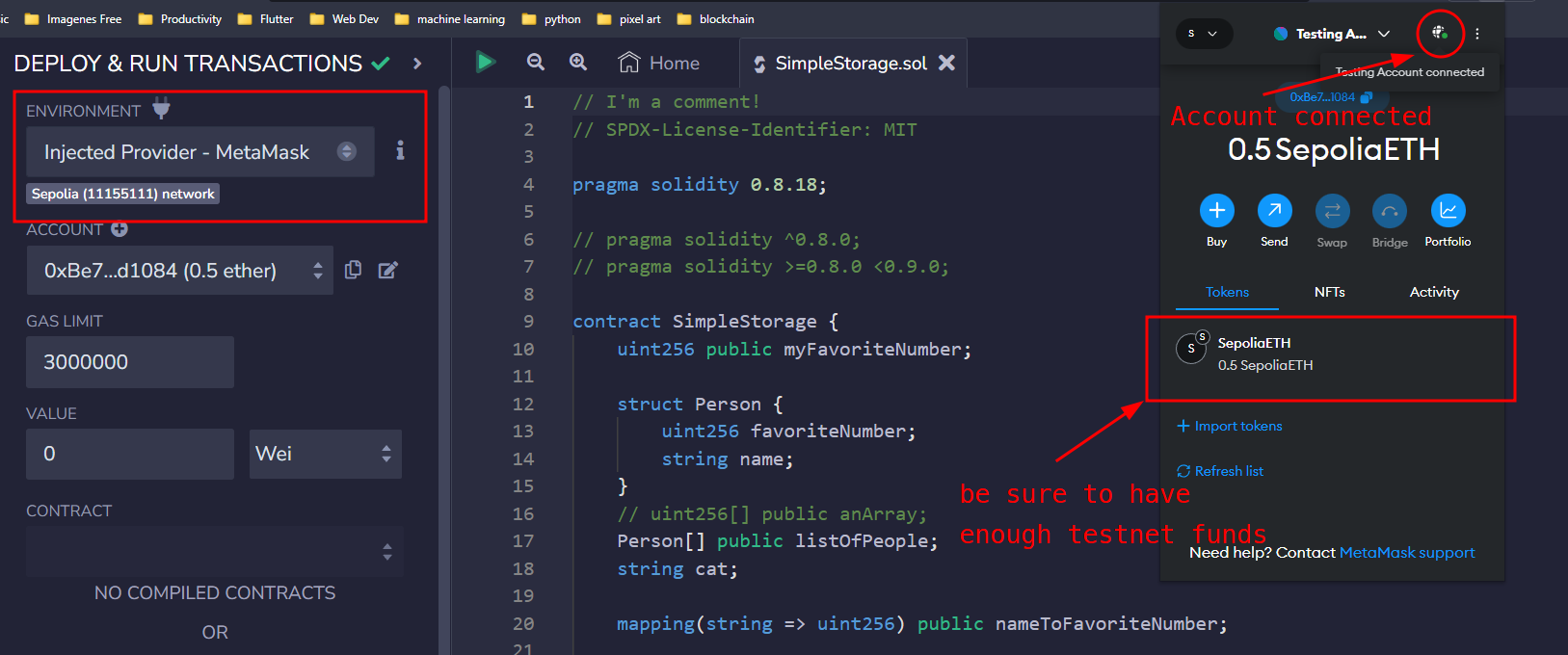
Before deploying, be always sure to make a **compilation check**. This ensures that the contract has no errors or warnings, and is fit for deployment.

### Deployment on a testnet

We can start the deployment process by going into the deployment tab and switching from the local virtual environment (Remix VM) to the Injected Provider - MetaMask. This action will allow Remix to send requests and interact with your MetaMask account.

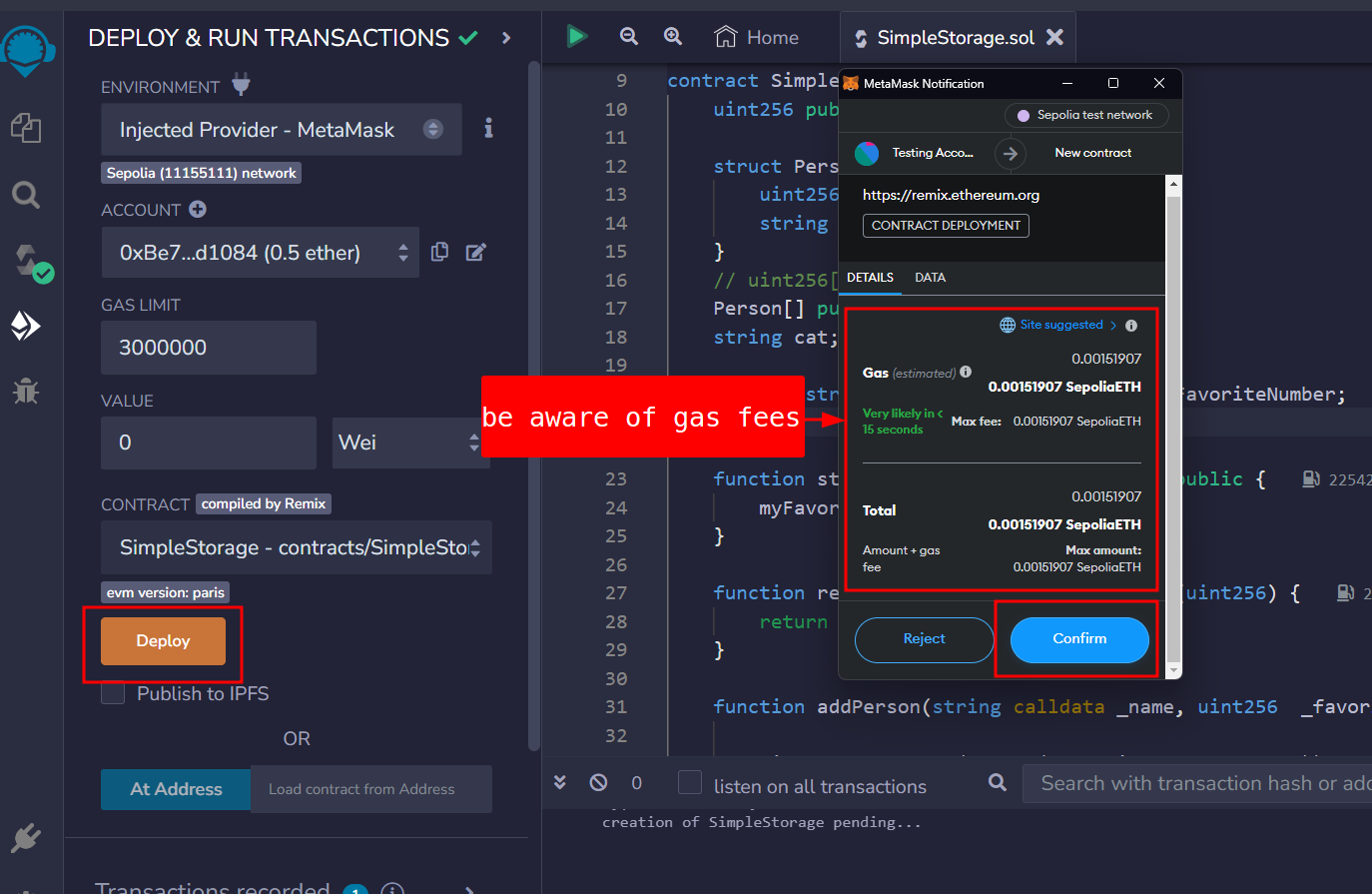


You will be then prompted to select an account from your MetaMask wallet. Once you've connected that account to Remix, you should see a confirmation that the account is properly linked and that you are using the Sepolia testnet.

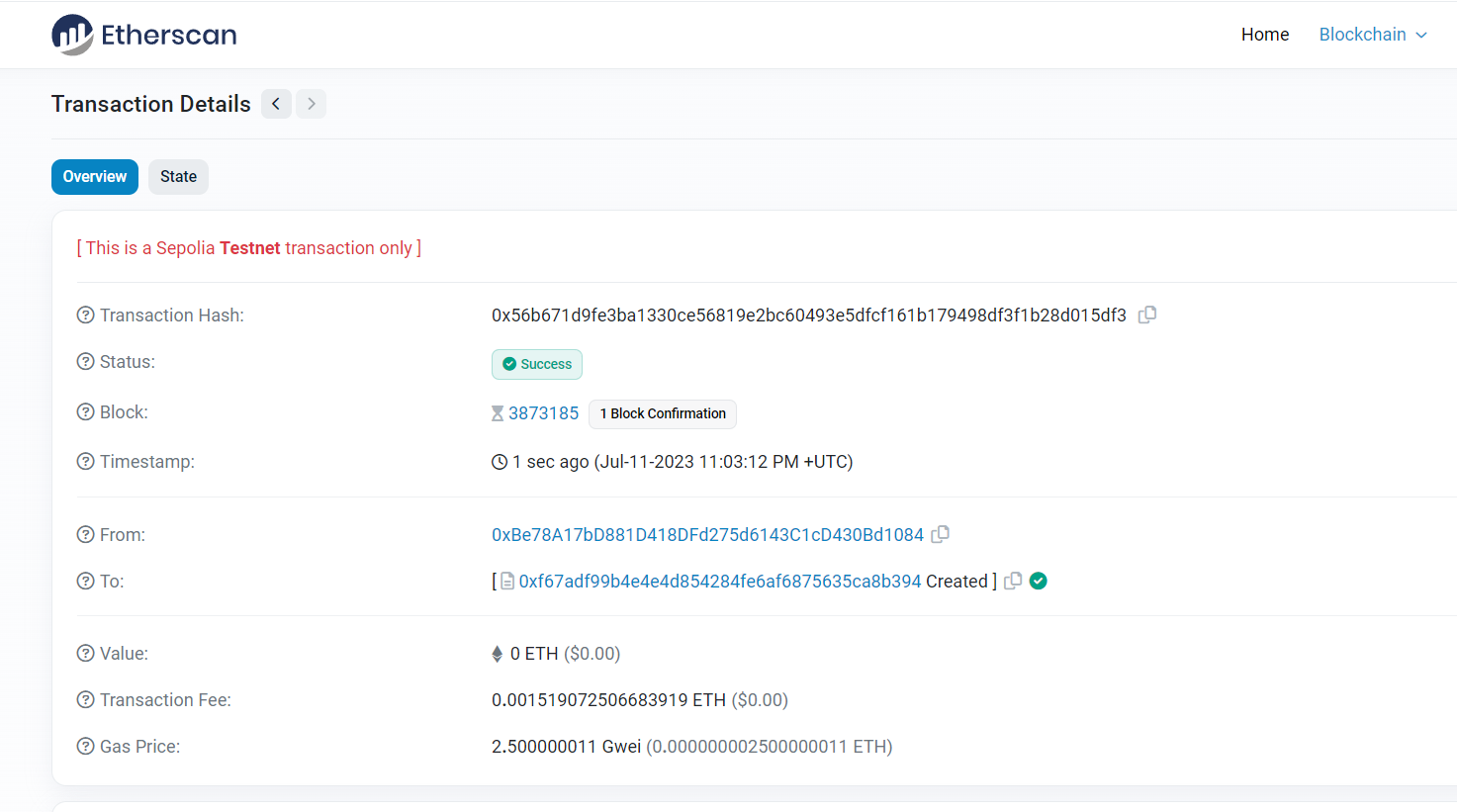


Ensure you have enough Sepolia ETH in your account, which you can obtain from a [faucet](https://www.alchemy.com/faucets/ethereum-sepolia). Once your balance is sufficient, you can proceed by clicking the "Deploy" button.

After that, MetaMask will ask to sign and send the transaction on the testnet.

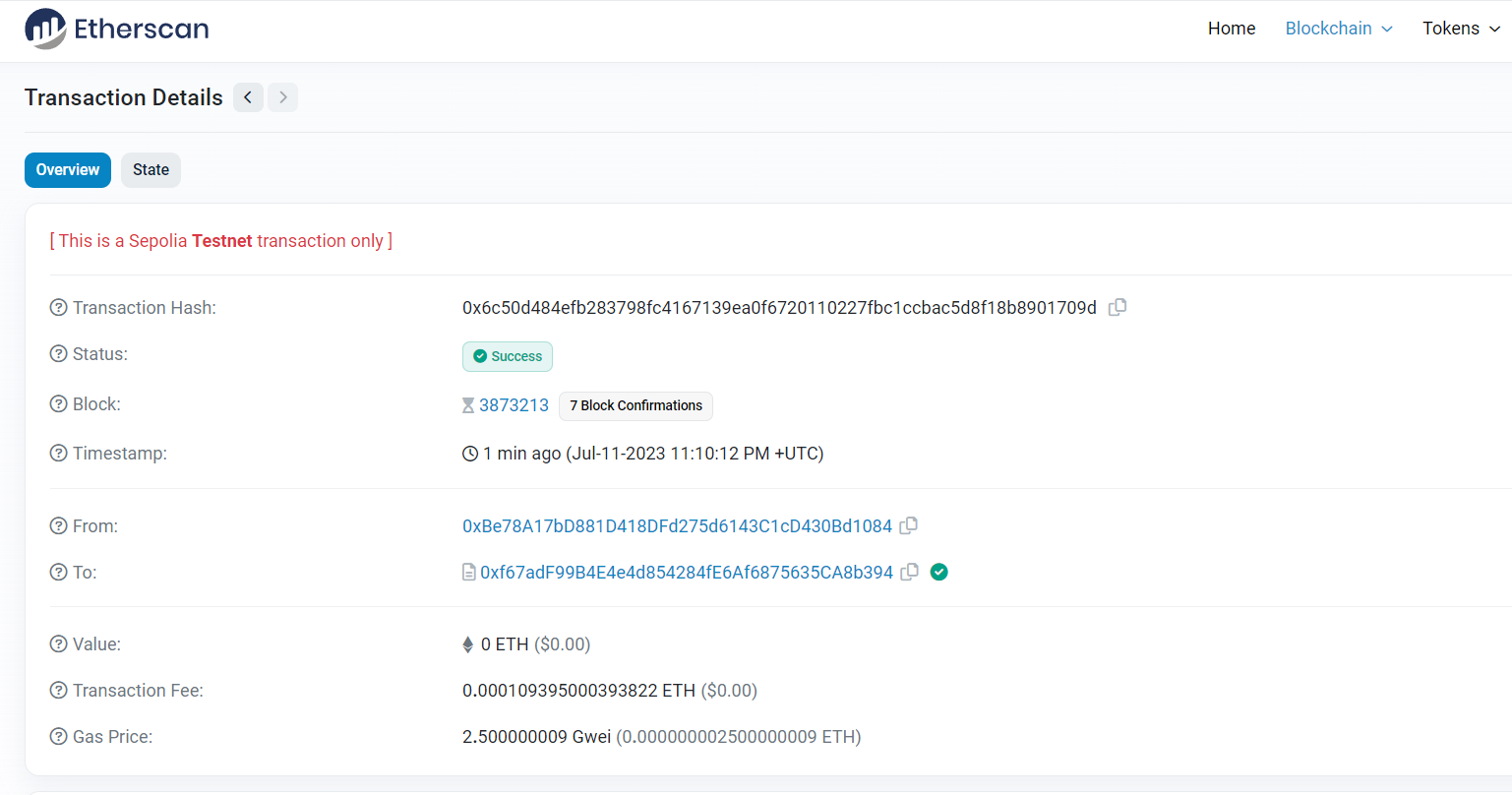


Once the transaction is executed, the contract address will be listed under deployed contracts, along with the transaction details. This is how the deployment transaction is displayed on Etherscan.



### Contract interaction

Since the contract has been deployed, we can now interact with it and **update the blockchain**. For example, if you want to store a number, you can do so by clicking the button 'store': MetaMask will ask for another transaction confirmation, that will update the favorite number. We can check the details on etherscan at the deployed address:



👀❗**IMPORTANT**  
View and pure functions will not send transactions

💡 **TIP**  
Celebrate small victories and milestones. These psychological boosts will keep you engaged in the learning process.

It's possible to deploy a contract to different testnets or a real mainnet, just by switching the MetaMask network. Be sure to have enough net-compatible ETHs to deploy your contract.

### Conclusion

Deploying a Solidity contract to a testnet is a crucial step in the development process, allowing you to test its functionality in a live blockchain environment without the risk of using real Ether. Always remember to perform necessary audits and tests to confirm the contract's safety and correctness before deployment.

### 💻 Test yourself

1. 📕 What steps should you take before deploying a contract to a testnet?
2. 🧑‍💻 Deploy one simple Solidity contract to the Sepolia testnet. Which important information can you see on [etherscan](https://sepolia.etherscan.io/)?

# Section 2 - Storage Factory

# Storage factory introduction

Introduction to deploying and interacting with contracts, focusing on Remix Storage Factory. The lesson involves working with 'SimpleStorage.sol', 'AddFiveStorage.sol', and 'StorageFactory.sol', demonstrating how other contracts can deploy and interact with new …

### Introduction

You can find the code for this section in the [Remix Storage Factory Github repository](https://github.com/cyfrin/remix-storage-factory-f23). In these nine lessons, we'll work with three new contracts:

1. SimpleStorage.sol - the contract we build in the previous section, with some modifications
2. AddFiveStorage.sol - a child contract of SimpleStorage that leverages inheritance
3. StorageFactory.sol - a contract that will deploy a SimpleStorage contract and interact with it

### Section overview

contract SimpleStorage {

SimpleStorage[] public listOfSimpleStorageContracts;

function createSimpleStorageContract() public {};

function sfStore(uint256 \_simpleStorageIndex, uint256 \_simpleStorageNumber) public {};

function sfGet(uint256 \_simpleStorageIndex) public view returns (uint256) {}

}

After deploying StorageFactory and executing its function createSimpleStorageContract, we can observe a new transaction appear in the Remix terminal. It's a **deployment transaction** of the SimpleStorage contract, executed by the StorageFactory contract.

It's possible to interact with this newly deployed SimpleStorage via the store function. We'll do this by using the **sfStore** function from the StorageFactory contract. This function accepts two parameters: the index of a deployed SimpleStorage contract, which will be '0' since we just deployed one contract, and the value of a favoriteNumber.

The **sfGet** function, when given the input '0', will indeed return the number provided by the previous function. The **address** of the SimpleStorage contract can then be retrieved by clicking on the get function listOfSimpleStorageContracts.

### Conclusion

The StorageFactory contract manages numerous instances of an external contract SimpleStorage. It provides functionality to deploy new contract instances dynamically and allows for the storage and retrieval of values from each instance. These instances are maintained and organized within an array, enabling efficient tracking and interaction.

### 🧑‍💻 Test yourself

1. 📕 What is the primary role of the StorageFactory contract?
2. 📕 Why is it important to specify the index when calling the sfStore function?

# Setting the project

This lesson explores the concept of composability in smart contracts, particularly in DeFi, and introduces the 'StorageFactory' contract that interacts with and deploys the 'SimpleStorage' contract. It covers setting up the StorageFactory contract in Remix and emphasizes the …

### Introduction

In this StorageFactory setup, we'll explore what composability means, showing its ability to deploy and interact with external SimpleStorage contracts.

### StorageFactory setup

You can begin by visiting the [Github repository of the previous section](https://github.com/cyfrin/remix-simple-storage-cu) and copying the contract SimpleStorage inside Remix. This contract allows to store a favorite number, a list of people with their favorite number, a mapping and different functionalities to interact with them. This lesson aims to create a **new contract** that can deploy and interact with SimpleStorage.

👀❗**IMPORTANT**  
One of the fundamental aspects of blockchain development is the seamless and permissionless interaction between contracts, known as **composability**. This is particularly crucial in decentralized finance (DeFi), where complex financial products interact effortlessly through common smart contract interfaces.

Let's set up the backbone of the code, that contains the function createSimplestorageContract. This function will deploy a SimpleStorage contract and save the result into a storage variable:

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.18;

contract StorageFactory {

function createSimplestorageContract() public {

// How does StorageFactory know what SimpleStorage looks like?

}

}

We need to establish a connection between the two contracts, since StorageFactory needs to have a complete knowledge of SimpleStorage. One first approach could be copying the SimpleStorage contract above StorageFactory.

🗒️ **NOTE**  
It's allowed to have multiple contracts in the same file. As best practice, however, it's recommended to use only one file for each contract

💡 **TIP**  
You can avoid confusion by keeping open **only** the file(s) you're currently working on.

### Conclusion

In this setup, we'll delve into the concept of composability and develop the StorageFactory contract, which will be capable of deploying and interacting with a SimpleStorage contract.

### 🧑‍💻 Test yourself

1. 📕 What does composability mean?
2. 📕 How many contracts is possible to deploy inside one .sol file?

# Deploying a contract from a contract

The chapter focuses on deploying a Simple Storage contract in Solidity and saving it to a storage or state variable. It covers the syntax for creating a Simple Storage contract within another contract and demonstrates the deployment and interaction process in Remix.

### Introduction

This lesson covers the process of **programmatically deploying** a SimpleStorage contract and saving it to a storage or state variable. By the end of this lesson, you will have a comprehensive understanding of how one contract can seamlessly deploy and manage another one.

### Creating a new variable

Following the format type-visibility-name, we can declare a new state variable of type SimpleStorage.

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.18;

contract StorageFactory {

SimpleStorage public simpleStorage;

function createSimplestorageContract() public {

simpleStorage = new SimpleStorage();

}

}

👀❗**IMPORTANT**  
SimpleStorage on the left and simpleStorage on the right are treated as entirely distinct entities due to their differing capitalization. Simple Storage refers to the contract type while simpleStorage refers to the variable name.

When the new keyword is used, the compiler recognizes the intention to deploy a new contract instance. After compiling, we can proceed to deploy it.

In Remix, you'll then notice two buttons: an orange createSimpleStorageContract and a blue one, SimpleStorage, generated by the public keyword. If we call both, first createSimpleStorageContract and then SimpleStorage, the address that appears below confirms that our SimpleStorage contract has been deployed.

### Conclusion

We have just deployed a contract that can programmatically create another contract, showing the principle of composability. In this way, contracts can know and interact with each other seamlessly.

### 🧑‍💻 Test yourself

1. 📕 What does the new keyword tell to the compiler?
2. 🧑‍💻 Create a contract AnimalFactory that includes a function createAnimals. This function must be capable of deploying the other 2 contracts Cows and Birds, which are simple contracts with just a constructor method.

# Solidity imports

This lesson covers the use of the 'import' statement in Solidity for organizing contract files, managing Solidity versions, and the advanced method of 'named imports'. It demonstrates how importing improves workflow and allows for selective inclusion of contract elements.

### Introduction

In the previous lesson, we integrated the SimpleStorage code directly into the StorageFactory contract. This allowed StorageFactory to have full access to the SimpleStorage contract’s functionality. In this lesson, we will explore a more efficient way to arrange and organize the code by using the **import** statement.

### Importing code

The import keyword enables a contract to utilize code from other files without needing to include the entire codebase directly within the contract. Here are two of the main advantages that the import keyword provides:

1. **No cluttering**: it prevents your current file from being cluttered with numerous lines of code, keeping it clean and organized.
2. **Simplified maintenance**: by keeping the code in separate files, it becomes easier to maintain and update individual components without affecting the entire codebase. For example, if we change some lines inside SimpleStorage, we would have also to constantly copy-paste the modified content into StorageFactory

You can now remove the previously added SimpleStorage code and replace it with the import shorthand:

import "./SimpleStorage.sol";

🚧 **WARNING**  
All the solidity contracts should be compiled together using the same compiler version. It's important to ensure **consistency** between compiler versions across files since each one will have its own pragma statement.

### Named Imports

Let's assume for a moment that SimpleStorage would contain multiple contracts, e.g. SimpleStorage, SimpleStorage1, SimpleStorage2, which are quite extensive in size. If we import the whole file as we did before, the statement will replace the import directive with all the code contained in SimpleStorage.sol. This will result in an unnecessary expensive deployment of the StorageFactory contract.

This can be prevented with **named imports**, which allow you to selectively import only the specific contracts you intend to use:

import { SimpleStorage } from "./SimpleStorage.sol";

You can also use named imports to import multiple contracts:

import { SimpleStorage, SimpleStorage1 } from "./SimpleStorage.sol";

👀❗**IMPORTANT**  
Try to always default to named imports instead of importing the entire file.

### Conclusion

The import keyword allows a contract to use code from other files without including the entire codebase. However, it can introduce compilation issues if different compiler versions are used in these files.

### 🧑‍💻 Test yourself

1. 📕 What's a named import and what are the advantages of using it?
2. 📕 In which way the pragma keyword can cause issues while using the import statement? Make 2 examples.

# Interacting with contracts ABI

This lesson teaches how to keep track of contract addresses when deploying new contracts using Solidity's 'new' keyword. It introduces the concept of ABI (Application Binary Interface) for contract interaction and demonstrates how to interact with contracts using ABI and addres

### Introduction

In this lesson, the StorageFactory contract will be upgraded to keep track of all deployed SimpleStorage contracts. This will allow us also to interact with each deployed contract individually.

### Storing the deployed contracts

In the current StorageFactory version, every time createSimpleStorageContract is called, a new SimpleStorage contract is deployed and **overridden** inside the variable SimpleStorage. Past deployments are not being tracked down.

To solve this issue we can create a variable listOfSimpleStorageContracts, which is an array of SimpleStorage contracts. In this way, whenever a contract is created, it gets added to a dynamic array.

SimpleStorage[] public listOfSimpleStorageContracts;

We can then modify the function createSimpleStorageContract, pushing the newly deployed contract to this variable.

function createSimpleStorageContract() public {

SimpleStorage simpleStorageContractVariable = new SimpleStorage();

listOfSimpleStorageContracts.push(simpleStorageContractVariable);

}

In Remix, you can access listOfSimpleStorageContracts via the index of type uint256, which refers to the **position** of the deployed contract inside the dynamic array.

### Simple Storage interaction

StorageFactory can interact with the deployed contracts by calling their store function. To do this we need to create a **function** sfStore:

function sfStore(uint256 \_simpleStorageIndex, uint256 \_simpleStorageNumber) public {

//SimpleStorage store function will be called here

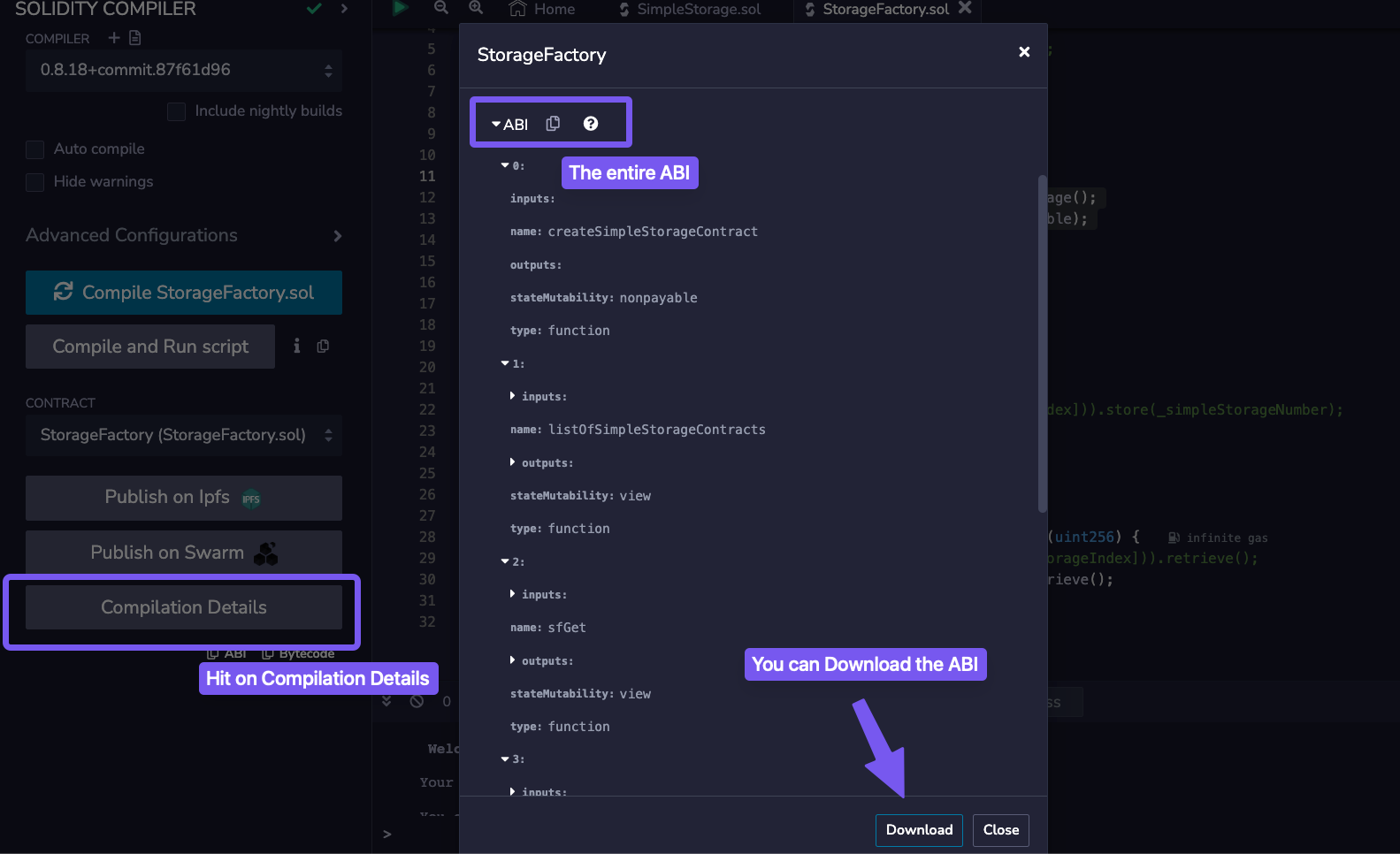
}

👀❗**IMPORTANT**  
Every time you have to interact with another contract, you need:

1. the contract **address**
2. the contract **ABI (Application Binary Interface)**: a standardized way for interacting with the binary version of a smart contract deployed on the blockchain. It specifies the functions, their parameters, and the structure of the data that can be used to interact with the contract. It's generated by the compiler.

🗒️ **NOTE**  
If you do not have the full ABI available, a function selector will suffice (see later in the course).

If you go to Solidity's compile tab, you will find a button that lets you copy the ABI to the clipboard.



🗒️ **NOTE**  
In Solidity, it's possible to **type cast** an address to a type contract

We can now proceed to store a new number on a SimpleStorage contract:

function sfStore(uint256 \_simpleStorageIndex, uint256 \_simpleStorageNumber) public {

listOfSimpleStorageContracts[\_simpleStorageIndex].store( \_simpleStorageNumber);

}

We can then retrieve the stored value with a get function:

function sfGet(uint256 \_simpleStorageIndex) public view returns (uint256) {

// return SimpleStorage(address(simpleStorageArray[\_simpleStorageIndex])).retrieve();

return listOfSimpleStorageContracts[\_simpleStorageIndex].retrieve();

}

### Conclusion

The StorageFactory contract was able to create a list of SimpleStorage contracts, store a variable in each of them and read it back.

### 🧑‍💻 Test yourself

1. 📕 What do you need to interact with an external contract?
2. 🧑‍💻 Deploy 3 instances of the SimpleStorage contract through the StorageFactory. Then store some numbers via sfStore and retrieve all of them via sfGet.

# Inheritance in Solidity

An introduction to inheritance and overriding in Solidity, showcasing how to extend the functionality of a contract without duplicating it. The lesson involves creating a new contract 'addFiveStorage.sol' that inherits from 'SimpleStorage.sol' and overrides its functions.

### Introduction

In this lesson, we are going to introduce the concept of **inheritance** and **overriding**, two powerful tools that allow developers to create more modular, maintainable, and reusable smart contracts. By leveraging these techniques, you can build upon existing contracts and customize their functions.

### Inheritance

We are going to enhance the SimpleStorage contract by adding a new functionality: the ability to add five (5) to the stored favoriteNumber. To achieve this, we could duplicate the existing SimpleStorage contract and make changes to the new version. However, this approach leads to redundant code. A better practice could be to utilize **inheritance**, which is the mechanism that allows the AddFiveStorage contract to derive all the functionalities of SimpleStorage.

Let's begin by creating a new file AddFiveStorage.sol and name-importing SimpleStorage.sol:

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.18;

import {SimpleStorage} from "./SimpleStorage.sol";

contract AddFiveStorage is SimpleStorage {}

The is keyword signifies inheritance and links the parent contract SimpleStorage to its child contract, AddFiveStorage.

### Override and virtual

The AddFiveStorage contract now inherits all methods from SimpleStorage. It's possible to add new functions to it, for example:

function sayHello() public pure returns(string memory) {

return "Hello";

}

We can also modify existing functions from SimpleStorage by using the **override** keyword. Let's say that we want to modify the store function, adding '5' to the favorite number being stored. If we copy the exact signature of the store function, an error will occur:

function store(uint256 \_newFavNumber) public {}

TypeError: Overriding function is missing "override" specifier.

🗒️ **NOTE**  
To override a method from the parent contract, we must replicate the exact function **signature**, including its name, parameters and adding the visibility and the override keyword to it:

function store(uint256 \_newFavNumber) public override {}

Yet, another error will pop up:

TypeError: Trying to override a non-virtual function.

To address this, we need to mark the store function in SimpleStorage.sol as **virtual**, enabling it to be overridden by child contracts:

function store(uint256 favNumber) public virtual {

// function body

}

Finally, we can add the new functionality to the store function in AddFiveStorage, allowing it to add '5' to the stored favoriteNumber:

function store(uint256 \_newFavNumber) public override {

favoriteNumber = \_newFavNumber + 5;

}

### Conclusion

In this lesson, we utilized inheritance to modify the SimpleStorage contract, without rewriting all its code. After deploying the contract AddFiveStorage and storing the number 2, it will return the favoriteNumber 7. This confirms that the store function in AddFiveStorage contract successfully overrides the existent store function in SimpleStorage.

### 🧑‍💻 Test yourself

1. 📕 Why do we need inheritance to extend a contract's functionality?
2. 📕 How are the keywords override and virtual used together?
3. 🧑‍💻 Create a contract Squared that overrides the store function and returns the favorite number squared.

# Sections summary and recap

A summary and recap of the lessons covering deploying contracts using the 'new' keyword, importing contracts, named imports, interacting with contracts using ABI, and contract inheritance in Solidity. The lesson celebrates progress made and encourages continued learning.

### Introduction

This section covered how to deploy contracts, how to import and interact with them, and using inheritance to customize their functionalities.

### Deploying and importing

We delved into the use of the **new** keyword to deploy multiple instances of a contract, allowing for the creation of numerous contract instances as needed.

Contracts can also be **imported**, which is equivalent to copying the code into the file but with the advantage of enhanced code reusability and modularity. It's good practice to use named imports, selecting only the contracts we intend to use from the file.

import { Contract as MyContract } from './myOtherContract.sol';

### Contracts interaction

Solidity lets you interact with other contracts. To do so we need the contract's address and its ABI (Application Binary Interface):

contract AddFiveStorage is SimpleStorage {}

### Inheritance and overriding

A contract can also derive functions from other contracts through **inheritance**. This can be obtained through the is keyword. To explicitly override a function from the parent contract, the override keyword is used in the child method. The parent's function must be marked as virtual to allow this interaction.

//child contract

import './ParentContract.sol';

contract ChildContract is ParentContract {

function store(uint256 \_num) public override {}

}

//parent contract

function store(uint256 \_num) public virtual {

// function body

}

### Conclusion

In this section, we explored deploying multiple contract instances using the new keyword and enhancing code reusability through contract imports. We also covered interacting with other contracts using their address and ABI. Additionally, we learned about inheritance and function overriding, allowing derived contracts to customize inherited functionalities. 💡 **TIP**  
When you finish a section, take a moment to acknowledge your progress, celebrate it and share your achievements with your community.

### 🧑‍💻 Test yourself

🏆 Attempt to answer all the theoretical questions from lesson 1 through 7, and then go back again to complete all the coding tasks.

# Section 3 – Fund Me

# Fund me introduction

Introduction to decentralized crowdfunding contract 'FundMe.sol', allowing users to send native blockchain cryptocurrency, with the owner being able to withdraw the funds. The lesson covers deploying on a testnet and handling transactions in Ethereum, Polygon, or Avalanche.

### Introduction

In this section, we'll create a decentralized crowdfunding contract. The complete codebase is available in the [Github repository](https://github.com/Cyfrin/remix-fund-me-f23).

### Overview

For this project, we will be using two contracts: FundMe, the main crowdfunding contract, and PriceConverter. They function much like Kickstarter, allowing users to **send** any native blockchain cryptocurrency. They also enable the owner of the contract to **withdraw** all the funds collected. We will then deploy these contracts on a **testnet**.

🗒️ **NOTE**  
Use testnet sparingly. Limiting testnet transactions helps prevent network congestion, ensuring a smoother testing experience for everyone.

### fund and withdraw

Once FundMe is deployed on Remix, you'll notice a set of functions, including a new red button labelled fund, indicating that the function is payable. A payable function allows you to send native blockchain currency (e.g., Ethereum, Polygon, Avalanche) to the contract.

We'll additionally indicate a **minimum USD amount** to send to the contract when the function fund is called. To transfer funds to the FundMe contract, you can navigate to the value section of the Remix deployment tab, enter a value (e.g. 0.1 ether) then hit fund. A MetaMask transaction confirmation will appear, and the contract balance will remain zero until the transaction is finalized. Once completed, the contract balance will be updated to reflect the transferred amount.

The contract owner can then withdraw the funds. In this case, since we own the contract, the balance will be removed from the contract's balance and transferred to our wallet.

### Conclusion

These 25 lessons will guide you step-by-step through the implementation of a crowdfunding contract, that supports cryptocurrency contributions and owner withdrawals.

# Project setup

This lesson guides through the initial steps in coding the 'FundMe' contract, which allows users to send funds and an owner to withdraw them. It involves setting up the Remix IDE workspace, outlining the contract functions, and focusing on the 'fund' function.

### Introduction

Let's begin by coding FundMe, a crowdfunding contract allowing users to send funds, which the owner can later withdraw. Before we start, let's clean up our Remix IDE workspace

### Setting up the project

Start from scratch by opening your [Remix IDE](https://remix.ethereum.org/) and deleting all existing contracts. Next, create a new contract named FundMe.

👀❗**IMPORTANT**  
Before you start coding, try to write down in plain English what you want your code to achieve. This helps clarify your goals and structure your approach.

We want FundMe to perform the following tasks:

1. **Allow users to send funds into the contract:** users should be able to deposit funds into the 'FundMe' contract
2. **Enable withdrawal of funds by the contract owner:** the account that owns FundMe should have the ability to withdraw all deposited funds
3. **Set a minimum funding value in USD:** there should be a minimum amount that can be deposited into the contract

Let's outline the core structure of the contract:

// SPDX-License-Identifier: MIT

pragma solidity ^0.8.18;

contract FundMe {}

### fund and withdraw functions

The FundMe contract will have two primary functions that serve as the main interaction points:

1. **fund:** allows users to deposit funds into the contract
2. **withdraw:** grants the contract owner the ability to withdraw the funds that have been previously deposited

First, let's code the fund function and leave the withdraw function commented out for the moment.

contract FundMe {

// send funds into our contract

function fund() public {}

// owner can withdraw funds

/\*function withdraw() public {}\*/

}

### Conclusion

In this lesson, we created a new FundMe contract and broadly defined the logic that will be performed.

### 🧑‍💻 Test yourself

1. 📕 Why should a developer always outline his coding goals before starting to code?

# Sending ETH through a function

This chapter explains how to create a function in a smart contract that requires a minimum amount of Ethereum (ETH) to be sent

### Introduction

In this part, we'll explore how to transfer Ethereum (ETH) to a smart contract by creating a fund function. This function will require a minimum amount of ETH to ensure proper transaction handling.

### value and payable

When a transaction it's sent to the blockchain, a **value** field is always included in the transaction data. This field indicates the **amount** of the native cryptocurrency being transferred in that particular transaction. For the function fund to be able to receive Ethereum, it must be declared **payable**. In the Remix UI, this keyword will turn the function red, signifying that it can accept cryptocurrency.

Wallet addresses and smart contracts are capable of **holding** and **managing** cryptocurrency funds. These entities can interact with the funds, perform transactions, and maintain balance records, just like a wallet.

function fund() public payable {

// allow users to send $

// have a minimum of $ sent

// How do we send ETH to this contract?

msg.value;

//function withdraw() public {}

}

In Solidity, the **value** of a transaction is accessible through the [msg.value](https://docs.soliditylang.org/en/develop/units-and-global-variables.html#special-variables-and-functions) **property**. This property is part of the global object msg. It represents the amount of **Wei** transferred in the current transaction, where Wei is the smallest unit of Ether (ETH).

### Reverting transactions

We can use the require keyword as a checker, to enforce our function to receive a minimum value of one (1) whole ether:

require(msg.value > 1e18); // 1e18 = 1 ETH = 1 \* 10 \*\* 18

This require condition ensures that the transaction meets the minimum ether requirements, allowing the function to execute only if this threshold is satisfied. If the specified requirement is not met, the transaction will **revert**.

The require statement in Solidity can include a custom error message, which is displayed if the condition isn't met, clearly explaining the cause of the transaction failure:

require(msg.value > 1 ether, "Didn't send enough ETH"); //if the condition is false, revert with the error message

An online tool like [Ethconverter](https://eth-converter.com/) can be useful for executing conversions between Ether, Wei, and Gwei.

👀❗**IMPORTANT**  
1 Ether = 1e9 Gwei = 1e18 Wei

🗒️ **NOTE**  
Gas costs are usually expressed in Gwei

If a user attempts to send less Ether than the required amount, the transaction will **fail** and a message will be displayed. For example, if a user attempts to send 1000 Wei, which is significantly less than one Ether, the function will revert and does not proceed.

### Conclusion

In this lesson, we explored how to use the value field of a transaction to transfer Ether to a contract. We also learned how to generate an **error message** when the user sends insufficient Ether to the FundMe contract.

### 🧑‍💻 Test yourself

1. 📕 Describe the role of the payable keyword. How does it affect the functionality of a function?
2. 📕 Explain how the require statement works in Solidity and what prevents.
3. 📕 What's the difference between Wei, Gwei and Ether?
4. 🧑‍💻 Create a tinyTip function that requires the user to send less than 1 Gwei.

# Solidity reverts

The lesson focuses on understanding 'reverts' and 'gas' in Ethereum transactions. It covers the concept of reverting transactions, checking gas usage, and how gas is used and refunded in failed transactions. The lesson also explores transaction fields and gas limits.

### Introduction

In this lesson, we will delve into how do transaction reverts work, what is gas where is used.

### Revert

Let's start by adding some logic to the fund function:

uint256 public myValue = 1;

function fund() public {

myValue = myValue + 2;

}

A revert action **undoes** all prior operations and returns the remaining gas to the transaction's sender. In this fund function, myValue increases by two (2) units with each successful execution. However, if a revert statement is encountered right after, all actions performed from the start of the function are undone. myValue will then reset to its initial state value, or one.

uint256 public myValue = 1;

function fund() public {

myValue = myValue + 2;

require(msg.value > 1e18, "didn't send enough ETH");

// a function revert will undo any actions that have been done.

// It will send the remaining gas back

}

### Gas Usage

🔥 **CAUTION**  
The gas used in the transaction will not be refunded if the transaction fails due to a revert statement. The gas has already been **consumed** because the code was executed by the computers, even though the transaction was ultimately reverted.

Users can specify how much gas they're willing to allocate for a transaction. In the case where the fund function will contain a lot of lines of code after the require and we did indeed set a limit, the gas which was previously allocated but not used will not be charged to the user

🗒️ **NOTE**  
If a transaction reverts, is defined as failed

### Transaction Fields

During a **value** transfer, a transaction will contain the following fields:

* **Nonce**: transaction counter for the account
* **Gas price (wei)**: maximum price that the sender is willing to pay per unit of gas
* **Gas Limit**: maximum amount of gas the sender is willing to use for the transaction. A common value could be around 21000.
* **To**: recipient's address
* **Value (Wei)**: amount of cryptocurrency to be transferred to the recipient
* **Data**: 🫙 empty
* **v,r,s**: components of the transaction signature. They prove that the transaction is authorised by the sender.

During a ***contract interaction transaction***, it will instead be populated with:

* **Nonce**: transaction counter for the account
* **Gas price (wei)**: maximum price that the sender is willing to pay per unit of gas
* **Gas Limit**: maximum amount of gas the sender is willing to use for the transaction. A common value could be around 21000.
* **To**: address the transaction is sent to (e.g. smart contract)
* **Value (Wei)**: amount of cryptocurrency to be transferred to the recipient
* **Data**: 📦 the content to send to the ***To*** address, e.g. a function and its parameters.
* **v,r,s**: components of the transaction signature. They prove that the transaction is authorised by the sender.

### Conclusion

**Reverts** and **gas usage** help maintain the integrity of the blockchain state. Reverts will undo transactions when failures occur, while gas enables transactions execution and runs the EVM. When a transaction fails, the gas consumed is not recoverable. To manage this, Ethereum allows users to set the maximum amount of gas they're willing to pay for each transaction.

### 🧑‍💻 Test yourself

1. 📕 Describe the two types of transactions listed in this lesson.
2. 📕 Why are reverts used?
3. 🧑‍💻 Bob sets his gas price to 20 Gwei and his gas limit to 50,000 units. The transaction consumes 30,000 units of gas before a revert occurs. How much ETH will be effectively charged?

# Intro to oracles - getting real world price data

This lesson introduces the concept of decentralized oracles and Chainlink for getting real-world price data into smart contracts. It explains how to update contracts for currency conversion, use Chainlink data feeds, and discusses Chainlink's role in blockchain oracles.

### Introduction

With the rapid advancement of blockchain technology and the growing adoption of decentralized finance platforms (DeFi), the necessity to support **multiple digital currencies** has significantly increased. Enabling users to utilize their preferred digital currencies extends market reach and improves the usability of an application.

This lesson will walk you through adding a **currency conversion feature** to the fundMe contract and setting **price thresholds** with Chainlink Oracle, a decentralized network for external data.

### USD values

Currently, our contract will require the transaction value to be greater than one Ethereum (ETH). If we want to give the users the flexibility to spend instead 5 USD, we would need to update our contract. We can begin by specifying the new value with a state variable uint256 public minimumUSD = 5 at the top of the contract.

The next step would be changing the condition inside the fund function, to check if the value sent is equal to or greater than our minimumUSD. However, we are facing a roadblock here: the minimumUSD value is in USD while the transaction message value is specified in ETH.

### Decentralized Oracles

The USD price of assets like Ethereum cannot be derived from blockchain technology alone but it is determined by the financial markets. To obtain a correct price information, a connection between off-chain and on-chain data is necessary. This is facilitated by a decentralized Oracle network.

This blockchain limitation exists because of its **deterministic nature**, ensuring that all nodes univocally reach a consensus. Attempting to introduce external data into the blockchain, will disrupt this consensus, resulting in what is referred to as a smart contract connectivity issue or the Oracle problem.

For smart contracts to effectively replace traditional agreements, they must have the capability to interact with **real-world data**.

Relying on a centralized Oracle for data transmission is inadequate as it reintroduces potential failure points. Centralizing data sources can undermine the trust assumptions essential for the blockchain functionality. Therefore, centralized nodes are not enough for external data or computation needs. Chainlink addresses these centralization challenges by offering a decentralized Oracle Network.

### How Chainlink Works

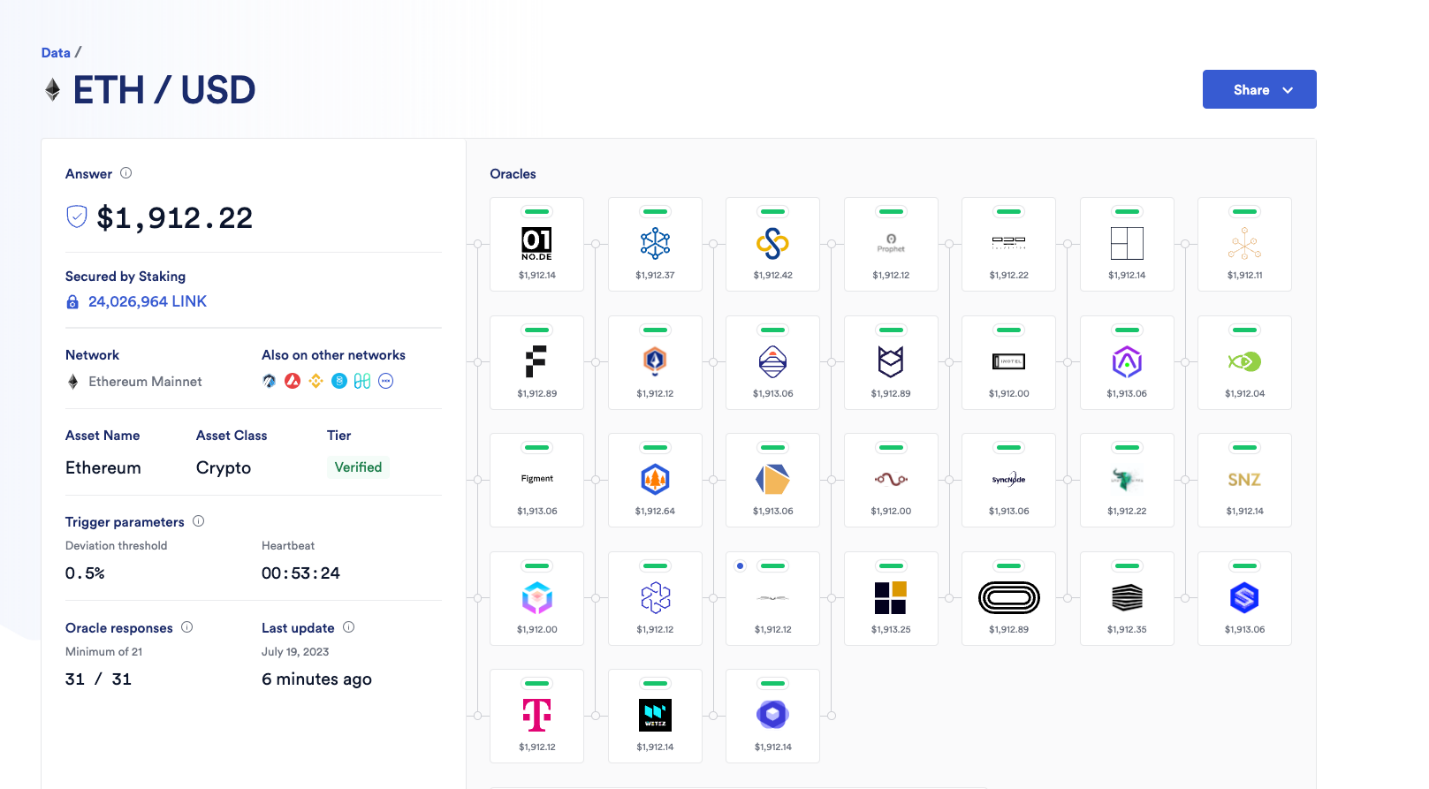
Chainlink is a modular and decentralized Oracle Network that enables the integration of data and external computation into a smart contract. When a smart contract combines on-chain and off-chain data, can be defined as **hybrid** and it can create highly feature-rich applications.

Chainlink offers ready-made features that can be added to a smart contract. And we'll address some of them:

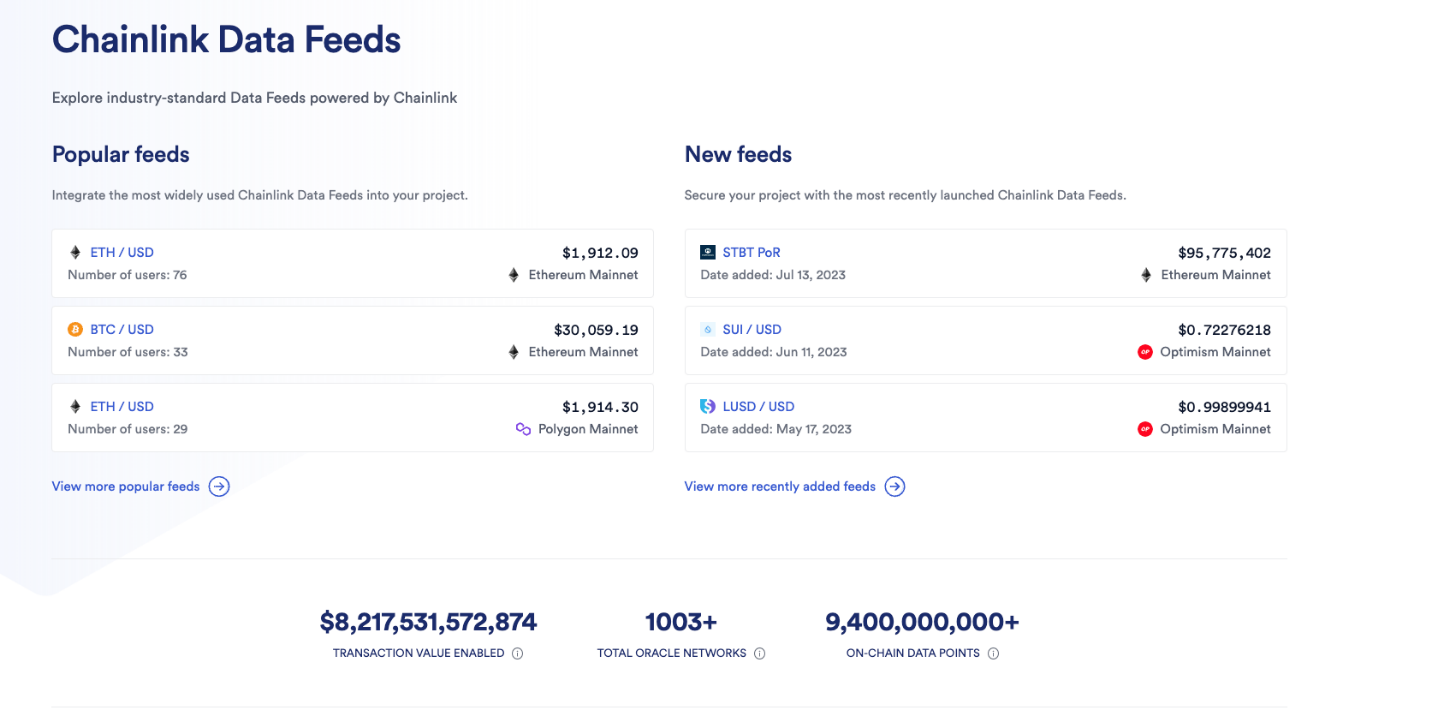
* **Data Feeds**
* **Verifiable Random Number**
* **Automation (previously known as "Keepers")**
* **Functions**

### Chainlink Data Feeds

Chainlink Data Feeds are responsible for powering over $50 billion in the DeFi world. This network of Chainlink nodes aggregates data from various **exchanges** and **data providers**, with each node independently verifying the asset price.

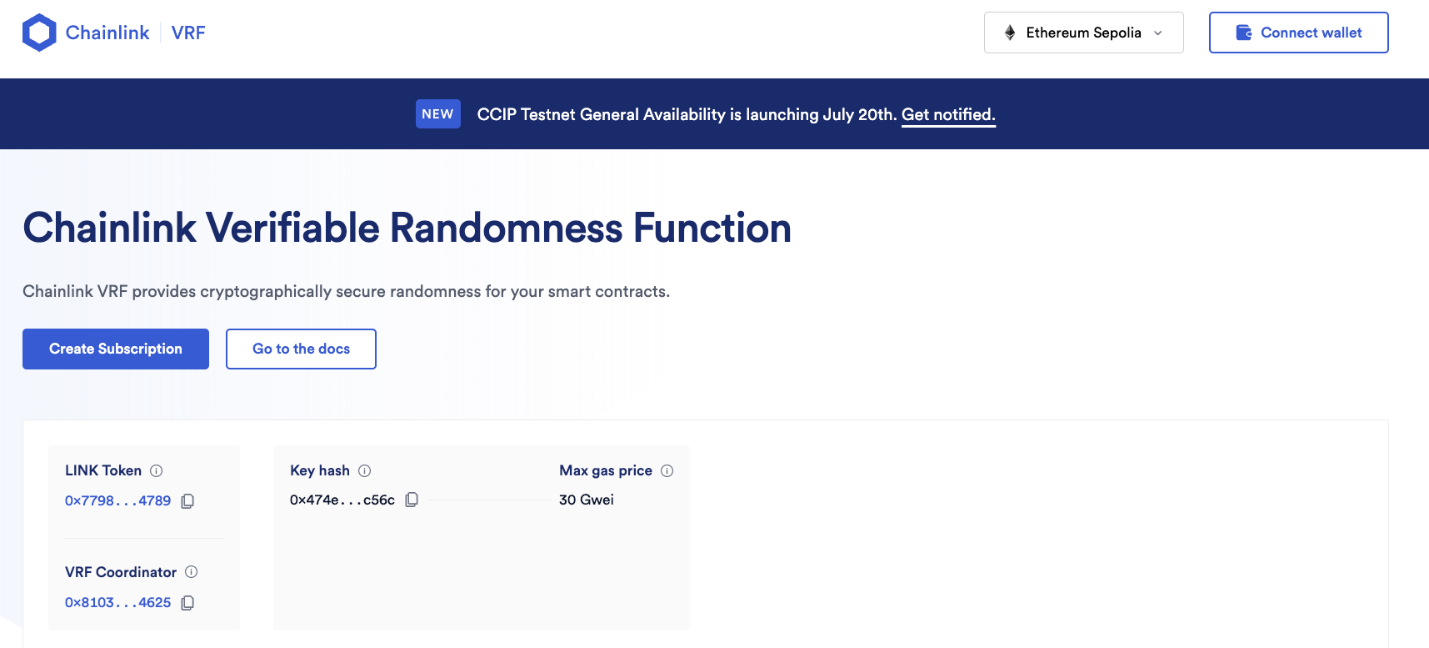


They aggregate this data and deliver it to a reference contract, the **price feed contract**, in a single transaction. Each contract will store the pricing details of a specific cryptocurrency



### Chainlink VRF

The Chainlink VRF (Verifiable Random Function) provides a solution for generating **provably random numbers**, ensuring true fairness in applications such as NFT randomization, lotteries, and gaming. These numbers are determined off-chain, and they are immune to manipulation.



### Chainlink Automation (previously known as "Keepers")

Another great feature is Chainlink's system of Keepers. These **nodes** listen for specific events and, upon being triggered, automatically execute the intended actions within the calling contract.

### Chainlink Function

Finally, Chainlink Functions allow **API calls** to be made within a decentralized environment. This feature is ideal for creating innovative applications and is recommended for advanced users with a thorough understanding of Chainlink ecosystem.

### Conclusion

Chainlink Data Feeds will help integrate currency conversion inside of our FundMe contract. Chainlink's decentralized Oracle network not only addresses the 'Oracle problem', but provides a suite of additional features for enhancing every dApp capabilities.

### 🧑‍💻 Test yourself

1. 📕 Describe 4 Chainlink products and what problem each one solves.

# Mid section recap

A recap of key concepts covered so far, including marking functions as payable for transactions, using 'require' statements, handling values with 'msg.value', and integrating external data using Chainlink for accurate real-world asset pricing in smart contracts.

### Introduction

From lessons 1 to 5 we've explored the usage of the keyword payable, the global property msg.value and what happens when a function reverts.

### payable, required, msg.value

To enable a function to receive a native blockchain token such as Ethereum, it must be marked as payable:

function deposit() public payable {

balances[msg.sender] += msg.value;

}

If we want a function to fail under certain conditions, we can use the require statement. For example, in a bank transfer scenario, we want the operation to fail if the sender does not have enough balance. Here's an example:

function transfer(address recipient, uint amount) public {

require(balances[msg.sender] >= amount);

balances[msg.sender] -= amount;

balances[recipient] += amount;

}

In this code, if the condition balances[msg.sender] >= amount is not met, the transaction will revert. This means the operation undoes any previous actions and will not consume the total maximum gas allocated by the user.

The Solidity global property msg.value contains the amount of cryptocurrency sent with a transaction.

### Integrating External Data with Chainlink

Chainlink is a revolutionary technology that enables the integration of external data and computation into smart contracts. It provides a decentralized way of **injecting data**, which is particularly useful for assets whose values fluctuate over time. For instance, if a smart contract deals with real-world assets such as stocks or commodities, obtaining real-time pricing information is crucial.

🗒️ **NOTE**  
Chainlink enables smart contracts to interact with real-world data and services without sacrificing the security and reliability guarantees inherent to blockchain technology.

Consider a smart contract that deals with a commodity like gold. Chainlink Price Feeds can provide real-time gold prices, allowing the smart contract to reflect the current market prices.

import "@chainlink/contracts/src/v0.6/interfaces/AggregatorV3Interface.sol";

contract GoldPriceContract {

AggregatorV3Interface internal priceFeed;

// The Chainlink price feed contract address

constructor() public {

priceFeed = AggregatorV3Interface(0x8468b2bDCE073A157E560AA4D9CcF6dB1DB98507);

}

// Get the latest gold price

function getLatestGoldPrice() public view returns (int) {

(,int price,,,) = priceFeed.latestRoundData();

return price;

}

}

In this example, Chainlink Feeds are used to query the latest price of gold, ensuring the smart contract has up-to-date market information.

### Conclusion

Understanding and utilizing payable, require, and msg.value is crucial for handling transactions in Solidity. Besides that, Chainlink enhances smart contract functionality by providing access to real-world data, allowing for more dynamic and reliable decentralized applications.

### 🧑‍💻 Test yourself

1. 📕 What are the three primary topics covered from lessons 1 to 5?

# Solidity interfaces

This lesson delves into using Solidity interfaces for converting Ethereum into USD and interacting with contracts. It explains how interfaces work, the importance of contract addresses and ABIs, and demonstrates interfacing with the Chainlink Aggregator V3 for price feeds.

### Introduction

In this part, we'll learn how to **convert** Ethereum (ETH) into Dollars (USD) and how to use **Interfaces**.

### Converting Ethereum into USD

We begin by trying to convert the msg.value, which is now specified in ETH, into USD. This process requires fetching the **current USD market price** of Ethereum and using it to convert the msg.value amount into USD.

// Function to get the price of Ethereum in USD

function getPrice() public {}

// Function to convert a value based on the price

function getConversionRate() public {}

### Chainlink Data Feed

Our primary source for Ethereum prices is a **Chainlink Data Feed**. [Chainlink Data Feed documentation](https://docs.chain.link/data-feeds/using-data-feeds) provides an example of how to interact with a Data Feed contract:

1. AggregatorV3Interface: a contract that takes a Data Feed address as input. This contract maintains the ETH/USD price updated.
2. latestRoundData: a function that returns an answer, representing the latest Ethereum price.

To utilize the **Price Feed Contract**, we need its address and its ABI. The address is available in the Chainlink documentation under the [Price Feed Contract Addresses](https://docs.chain.link/data-feeds/price-feeds/addresses). For our purposes, we'll use ETH/USD price feed.

### Interface

To obtain the ABI, you can import, compile, and deploy the PriceFeed contract itself. In the previous section, we imported the SimpleStorage contract into the StorageFactory contract, deployed it, and only then we were able to use its functions.

An alternative method involves the use of an **Interface**, which defines methods signature without their implementation logic. If compiled, the Price Feed Interface, it would return the ABI of the Price Feed contract itself, which was previously deployed on the blockchain. We don't need to know anything about the function implementations, only knowing the AggregatorV3Interface methods will suffice. The Price Feed interface, called Aggregator V3 Interface, can be found in [Chainlink's GitHub repository](https://github.com/smartcontractkit/chainlink/blob/develop/contracts/src/v0.8/shared/interfaces/AggregatorV3Interface.sol).

🗒️ **NOTE**  
Interfaces allow different contracts to interact seamlessly by ensuring they share a common set of functionalities.

We can test the Interface usage by calling the version() function:

function getVersion() public view returns (uint256) {

return AggregatorV3Interface(0x5f4eC3Df9cbd43714FE2740f5E3616155c5b8419).version();

}

🗒️ **NOTE**  
It's best to work on testnets only after your deployment is complete, as it can be time and resource consuming.

### Conclusion

Using interfaces is a common and effective way to interact with external contracts. First, obtain the interface of the external contract, compile it to get the ABI, and then use the deployed contract's address. This allows you to call any function available at that address seamlessly.

### Test yourself

1. 📕 Explain the role of interfaces in Solidity and why are they advantageous.
2. 📕 What are the steps required to convert a variable containing a value in ETH to its equivalent in USD?
3. ‍💻 Implement another function on the FundMe contract that implements the decimals() methods of the Data Feed address.

# Importing libraries from NPM and Github

This chapter explores how to import libraries and interfaces directly from GitHub or NPM in Ethereum contract development. It covers the benefits of direct imports for managing interfaces, using the Chainlink AggregatorV3Interface as an example.

### Introduction

As we delve into smart contract development, **interacting** with external smart contracts will become increasingly frequent. This can involve the use of multiple interfaces, which can clog up the source code and make it difficult to read.

### Interfaces

Let's take a look at the SmartContract interface as an example:

interface SmartContract {

function someFunction() external view returns(uint, uint){};

}

To include this contract, we simply use the import "./SimpleStorage.sol" statement at the top of our file. Instead, if the project is not stored locally, we can use imports from Github.

### Direct Imports from GitHub

Smart Contracts hosted on GitHub can be imported directly into your project. For instance, consider the AggregatorV3Interface contract from Chainlink, which is located in Chainlink's GitHub repository and not in our project's directory.

Instead of manually copying all its code into your project and then importing it like this:

import { AggregatorV3Interface } from "./AggregatorV3Interface.sol";

we can import it more efficiently, as specified in the [Chainlink documentation](https://docs.chain.link/docs/using-chainlink-reference-contracts):

import { AggregatorV3Interface } from "@chainlink/contracts/src/v0.8/interfaces/AggregatorV3Interface.sol";

This import statement includes the **path** to the AggregatorV3Interface.sol file in the GitHub repository, allowing you to directly import the contract from GitHub or NPM (Node Package Manager).

### Importing from NPM

The @chainlink/contracts package, available on NPM, follows **Semantic Versioning (SemVer)**, which allows you to download and use specific versions in your contracts (e.g., npm install @chainlink/contracts@1.2.3) while being directly synchronized with Chainlink's GitHub repository. The rest of the import path specifies the exact file that Remix should use.

Remix interprets @chainlink/contracts as a reference to the [NPM package](https://www.npmjs.com/package/@chainlink/contracts), and downloads all the necessary code from it.

pragma solidity ^0.8.18;

import {AggregatorV3Interface} from "@chainlink/contracts/src/v0.8/interfaces/AggregatorV3Interface.sol";

contract FundMe {}

### Conclusion

Efficiently managing external smart contracts is crucial in smart contract development to maintain clean and readable source code. Utilizing **direct imports** from GitHub and NPM, as shown with Chainlink's AggregatorV3Interface, simplifies this process.

### 🧑‍💻 Test yourself

1. 📕 What is this statement @chainlink/contracts/src/v0.8/interfaces/AggregatorV3Interface.sol translated into when interpreted by the solidity compiler?

# Getting real world price data from Chainlink

The lesson focuses on extracting real-world pricing information using the Aggregator V3 interface from Chainlink. It covers creating contract instances, summoning 'latestRoundData', dealing with decimals in Solidity, and typecasting for price and value compatibility.

### Introduction

The AggregatorV3Interface provides a streamlined ABI for interacting with the Data Feed contract. In this lesson, we'll explore how to retrieve pricing information from it.

### Creating a New Contract Instance

The AggregatorV3Interface includes the latestRoundData function, which retrieves the latest cryptocurrency price from the specified contract. We'll start by declaring a new variable, priceFeed, of type AggregatorV3Interface. This interface requires the address of the Data Feed contract deployed on the Sepolia Network.

AggregatorV3Interface priceFeed = AggregatorV3Interface(0x1b44F3514812d835EB1BDB0acB33d3fA3351Ee43);

### The latestRoundData Function

We can call the latestRoundData() function on this interface to obtain several values, including the latest price.

function latestRoundData() external view returns (uint80 roundId, int256 answer, uint256 startedAt, uint256 updatedAt, uint80 answeredInRound);

For now, we'll focus on the answer value and ignore the other returned values by using commas as placeholders for the unneeded variables.

function getLatestPrice() public view returns (int) {

(,int price,,,) = priceFeed.latestRoundData();

return price;

}

Our getLatestPrice() function now retrieves the latest ETH price in USD from the latestRoundData() function of the Data Feed contract. The returned price is an int256 with a precision of 1e8, as indicated by the decimals function.

### Decimals

* msg.value is a uint256 value with 18 decimal places.
* answer is an int256 value with 8 decimal places (USD-based pairs use 8 decimal places, while ETH-based pairs use 18 decimal places).

This means the price returned from our latestRoundData function isn't directly compatible with msg.value. To match the decimal places, we multiply price by 1e10:

return price \* 1e10;

### Typecasting

Typecasting, or type conversion, involves converting a value from one data type to another. In Solidity, not all data types can be converted due to differences in their underlying representations and the potential for data loss. However, certain conversions, such as from int to uint, are allowed.

return uint(price) \* 1e10;

We can finalize our view function as follows:

function getLatestPrice() public view returns (uint256) {

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

return uint(answer) \* 1e10;

}

### Conclusion

This complete getLatestPrice function retrieves the latest price, adjusts the decimal places, and converts the value to an unsigned integer, making it compatible for its use inside other functions.

### 🧑‍💻 Test yourself

1. 📕 Why we need to multiply the latest ETH price by 1e10?
2. 📕 What's the result of the typecasting uint256(-2)?
3. 🧑‍💻 Create a contract with a getLatestBTCPriceInETH() function that retrieves the latest BTC price in ETH.

# Solidity math

This lesson provides insights into converting Ethereum value to USD using Solidity. It covers the implementation of 'getPrice' and 'getConversionRate' functions, understanding decimal places, value validation, and deployment on a testnet.

### Introduction

In this lesson, we will guide you through converting the value of ETH to USD. We'll use the previously defined getPrice function within the new getConversionRate function.

### The getPrice and getConversionRate Functions

The getPrice function returns the current value of Ethereum in USD as a uint256.  
The getConversionRate function converts a specified amount of ETH to its USD equivalent.

### Decimal Places

In Solidity, only integer values are used, as the language does not support floating-point numbers.

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

uint256 ethPrice = getPrice();

uint256 ethAmountInUsd = (ethPrice \* ethAmount) / 1e18;

return ethAmountInUsd;

}

🗒️ **NOTE**  
The line uint256 ethAmountInUsd = (ethPrice \* ethAmount) results in a value with a precision of 1e18 \* 1e18 = 1e36. To bring the precision of ethAmountInUsd back to 1e18, we need to divide the result by 1e18.

🔥 **CAUTION**  
Always multiply before dividing to maintain precision and avoid truncation errors. For instance, in floating-point arithmetic, (5/3) \* 2 equals approximately 3.33. In Solidity, (5/3) equals 1, which when multiplied by 2 yields 2. If you multiply first (5\*2) and then divide by 3, you achieve better precision.

### Example of getConversionRate

* ethAmount is set at 1 ETH, with 1e18 precision.
* ethPrice is set at 2000 USD, with 1e18 precision, resulting in 2000e18.
* ethPrice \* ethAmount results in 2000e36.
* To scale down ethAmountInUsd to 1e18 precision, divide ethPrice \* ethAmount by 1e18.

### Checking Minimum USD Value

We can verify if users send at least 5 USD to our contract:

require(getConversionRate(msg.value) >= MINIMUM\_USD, "You need to spend more ETH!");

Since getConversionRate returns a value with 18 decimal places, we need to multiply 5 by 1e18, resulting in 5 \* 1e18 (equivalent to 5 \* 10\*\*18).

### Deployment to the Testnet

In Remix, we can deploy the FundMe contract to a testnet. After deployment, the getPrice function can be called to obtain the current value of Ethereum. It's also possible to send money to this contract, and an error will be triggered if the ETH amount is less than 5 USD.

Gas estimation failed. Error execution reverted, didn't send enough ETH.

### Conclusion

In this lesson, we've demonstrated how to convert ETH to USD using the getConversionRate function, ensuring precise calculations by handling decimal places correctly.

### 🧑‍💻 Test yourself

1. 📕 Why is it important to multiply before dividing in Solidity calculations, and how does this practice help maintain precision?
2. 📕 What is the purpose of the getConversionRate function, and how does it utilize the getPrice function to convert ETH to USD?
3. 🧑‍💻 Create a function convertUsdToEth(uint256 usdAmount, uint256 ethPrice) public returns(uint256), that converts a given amount of USD to its equivalent value in ETH.

# Msg sender explained

The lesson introduces the use of Solidity's global variables, arrays, and mappings to track users sending money to a contract. It covers creating a mechanism to record addresses and amounts sent by users using 'msg.sender' and mappings.

### Introduction

In this lesson, we will learn how to track addresses that are funding the contract and the amounts they will send to it.

### Tracking Funders

To track the addresses are sending money to the contract, we can create an array of addresses named funders:

address[] public funders;

Whenever someone sends money to the contract, we will add their address to the array with the push function:

funders.push(msg.sender);

The msg.sender global variable refers to the address that **initiates the transaction**.

### Mapping Users to Funds Sent

We can also map each funder's address to the amount they have sent using **mappings**. Let's define a mapping in Solidity:

mapping(address => uint256) public addressToAmountFunded;

The addressToAmountFunded mapping associates each funder's address with the total amount they have contributed. When a new amount is sent, we can add it to the user's total contribution:

addressToAmountFunded[msg.sender] += msg.value;

### Conclusion

We have successfully implemented a system to track users who fund the fundMe contract. This mechanism records every address that is sending ETH to the contract, and maps the sender's address to the total amount they have contributed.

### 💻 Test yourself

1. 📕 Explain why we need to use the mapping addressToAmountFunded inside the fundMe contract
2. 🧑‍💻 Implement a function contributionCount to monitor how many times a user calls the fund function to send money to the contract.

# Quick section recap

A comprehensive refresher on key concepts in Advanced Solidity, covering contract addresses and ABIs, interfacing with contracts, using Chainlink Price Feeds, handling decimals and global units in Solidity, and the importance of these elements in smart contract development.

### Introduction

In this recap, we'll review how to interact with an external contract and utilize its functions, understand Chainlink Price Feeds, perform Solidity math, and explore global properties.

### Interacting with an External Contract

To interact with any external contract, you need the contract's address and ABI (Application Binary Interface). Think of the address as a house number that identifies the specific contract on the blockchain, while the ABI serves as a manual that explains how to interact with the contract.

To obtain the contract ABI, you can compile a Solidity **interface** that the target contract implements. Then, create a new instance of the interface pointing to the specific address of the deployed contract.

### Chainlink Price Feeds

[Chainlink Price Feeds](https://docs.chain.link/docs/using-chainlink-reference-contracts/) provide a reliable way to access real-world data, such as pricing data, and inject it into smart contracts. This is particularly useful for executing mathematical operations in Solidity and the Ethereum Virtual Machine (EVM), where floating-point numbers are not used.

### Solidity Global Properties

The [Solidity documentation](https://docs.soliditylang.org/en/latest/cheatsheet.html#block-and-transaction-properties) provides several global properties that are essential for interacting with the Ethereum blockchain. Here are two key properties:

* msg.sender: this property refers to the address of the account that **initiated the current function call**
* msg.value: this property represents the **amount of Wei** sent with a function call

function updateValue() public payable {

require(msg.value >= 1 ether, "Not enough Ether provided.");

}

By understanding these concepts, you can effectively interact with external contracts, leverage Chainlink Price Feeds for real-world data, and utilize Solidity's global properties for more robust smart contract development.

### 🧑‍💻 Test yourself

1. 🏆 Attempt to answer all the theoretical questions from lessons 1 through 12, and then go back again to complete all the coding tasks.

# Creating your own libraries

This lesson covers the creation and use of Solidity Libraries to streamline code and avoid redundancy. It demonstrates how to create a library, transfer functions to it, and utilize the library in contracts for efficient code management and functionality enhancement.

### 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.

### 🧑‍💻 Test yourself

1. 📕 What are the differences between Solidity libraries and contracts?
2. 📕 What are the consequences if a library function is not marked as internal?
3. 🧑‍💻 Create a simple library called MathLibrary that contains a function sum to add two uint256 numbers. Then create a function calculateSum inside the fundMe contract that uses the MathLibrary function.

# Using Safemath

An introduction to the SafeMath library in Solidity, explaining its significance before Solidity 0.8 and the reasons for its reduced usage post Solidity 0.8. The lesson covers integer overflow issues and the implementation of automatic checks in newer Solidity versions.

### Introduction

In this lesson, we will explore SafeMath, a widely used library before Solidity version 0.8, and understand why its usage has now decreased.

### Integer Overflow

SafeMath.sol was a staple in Solidity contracts before version 0.8. After this version, its usage has significantly dropped.

Let's begin by creating a new file called SafeMathTester.sol and adding a function add that increments the bigNumber state variable.

// SafeMathTester.sol

pragma solidity ^0.6.0;

contract SafeMathTester {

uint8 public bigNumber = 255;

function add() public {

bigNumber = bigNumber + 1;

}

}

Notice we are using compiler version 0.6.0. The bigNumber is a uint8 variable with a maximum value of 255. If we call the add function, it will return 0 instead of the expected 256.

Before Solidity version **0.8.0**, signed and unsigned integers were **unchecked**, meaning that if they exceeded the maximum value the variable type could hold, they would reset to the lower limit. This pattern is known as **integer overflow** and the SafeMath library was designed to prevent it.

### SafeMath

SafeMath.sol provided a mechanism to revert transactions when the maximum limit of a uint256 data type was reached. It was a typical security measure across contracts to avoid erroneous calculations and potential exploits.

function add(uint a, uint b) public pure returns (uint) {

uint c = a + b;

require(c >= a, "SafeMath: addition overflow");

return c;

}

### Solidity 0.8.0

With the introduction of Solidity version 0.8, automatic checks for overflows and underflows were implemented, making SafeMath redundant for these checks. If SafeMathTester.sol is deployed with Solidity 0.8.0, invoking the add function will cause a transaction to fail, when, in older versions, it would have reset to zero.

For scenarios where mathematical operations are known not to exceed a variable's limit, Solidity introduced the unchecked construct to make code more gas-efficient. Wrapping the addition operation with unchecked will ignore the overflow and underflow checks: if the bigNumber exceeds the limit, it will wrap its value to zero.

uint8 public bigNumber = 255;

function add() public {

unchecked {

bigNumber = bigNumber + 1;

}

}

🔥 **CAUTION**  
It's important to use unchecked blocks with caution as they reintroduce the possibility of overflows and underflows.

### Conclusion

The evolution of Solidity and SafeMath.sol highlights the continuous advancements in Ethereum smart contract development. Although recent updates have made SafeMath.sol less essential, it remains a significant part of Ethereum's history. Understanding its role provides valuable insight into the progress and maturation of Solidity.

### 🧑‍💻 Test yourself

1. 📕 Why was the SafeMath library widely used before version 0.8?
2. 📕 Explain the meaning of integer overflow and integer underflow. Make an example using uint16.
3. 📕 What happened after Solidity version 0.8?
4. 📕 What is the unchecked construct?
5. 🧑‍💻 Modify the SafeMathTester contract by using the SafeMath library to prevent integer overflow.

# Solidity for Loop

This lesson teaches the concept of for loops in Solidity, demonstrating how they can be used to access and manipulate arrays. It focuses on practical applications in a smart contract, particularly for iterating over arrays and resetting mappings.

### Introduction

In this tutorial, we'll set up the withdraw function to enable the retrieval of accumulated funds. We'll then reset all the mappings to zero using a method known as a **for loop**.

### For Loops

A for loop is a widely-used concept in many programming languages that enables the execution of a block of code a repeated amount of times.

For example, consider this list:

//[1, 2, 3, 4] elements

// 0 1 2 3 indexes

The elements of the list are the numbers 1 through 4, with indices ranging from 0 to 3: the number 1 is at index 0, the number 2 is at index 1, and so on. To access all the elements in this list, we can use their **indexes**.

A typical for loop structure in programming languages involves:

1. Initializing at some starting index
2. Iterating until an end index
3. Incrementing by certain steps

For instance, if we start the loop at index 0, end at index 10, and increment by 1 each time, we'll get:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

However, if we start at index 3, end at index 12, and increment by 2 each time, we get:

3, 5, 7, 9, 11

### Using for Loops in the FundMe Contract

To implement this concept in the FundMe contract:

uint256 funderIndex;

for (funderIndex = 0; funderIndex < funders.length; funderIndex++) {

address funder = funders[funderIndex];

addressToAmountFunded[funder] = 0;

}

The loop begins at index 0 and goes through all the elements in the funders array until it reaches the final element. With each iteration, it performs the following actions:

1. Accesses the funder address at the current index
2. Resets the corresponding funding amount in the addressToAmountFunded mapping to zero, clearing the funder's record.

🗒️ **NOTE**  
The **addressToAmountFunded** map connects addresses with the respective amounts they funded.

### Shortcuts

Additionally, we have used two shorthands in our code:

1. funderIndex++: shorthand for funderIndex = funderIndex + 1.
2. +=: adds a value to an existing one. x = x + y is equivalent to x += y.

To illustrate the code snippet, we start from funderIndex 0. When entering the loop, we get the address of the funder at the 1st position in the funders array and set its amount to zero. After that, we repeat the loop, incrementing the funderIndex by 1 and checking whether it is still less than the total number of funders. We then get the address of the funder at the next position and so on.

### Conclusion

In this lesson, we learned how to implement the withdraw function to manage accumulated funds and reset mappings using a for loop. This process ensures efficient handling of funders' records.

### 🧑‍💻 Test yourself

1. 📕 What are the shortcuts we addressed in this lesson?
2. 📕 How does a for loop work in Solidity?
3. 🧑‍💻 Implement a function named pushNumbers to populate a numbers array with values from 1 to 10.

# Resetting an Array

A guide on effectively resetting arrays in Solidity, particularly within the context of smart contracts. The lesson addresses the importance of resetting arrays for managing and updating contract states, and demonstrates the process using practical examples.

### Introduction

In this section, we'll focus on one of the final steps to complete the withdraw function: effectively resetting the funders array.

### Resetting an Array

The simplest way to reset the funders array is similar to the method used with the mapping: iterate through all its elements and reset each one to 0. Alternatively, we can create a brand new funders array.

funders = new address[]();

🗒️ **NOTE**  
You might recall using the new keyword when deploying a contract. In this context, however, it resets the funders array to a zero-sized, blank address array.

### Conclusion

In this lesson, we learned how to reset the funders array by either iterating through its elements or creating a new zero-sized array. This step is crucial for completing the withdraw function and ensuring the contract's data is properly managed.

### 🧑‍💻 Test yourself

1. 📕 Why is it important to reset the funders array when implementing the withdraw function?
2. 🧑‍💻 Create a method expensiveReset that resets an array using the iteration method.

# Sending ETH from a contract

An exploration of three methods for sending Ether from a contract in Solidity: transfer, send, and call. The lesson compares these methods, discussing their syntax, behavior, and appropriate use cases, with a focus on their gas usage and security implications.

### Introduction

This lesson explores three different methods of sending ETH from one account to another: transfer, send, and call. We will understand their differences, how each one works, and when to use one instead of another.

### Transfer

The transfer function is the simplest way to send Ether to a recipient address:

payable(msg.sender).transfer(amount); // the current contract sends the Ether amount to the msg.sender

It's necessary to convert the recipient address to a **payable** address to allow it to receive Ether. This can be done by wrapping msg.sender with the payable keyword.

However, transfer has a significant limitation. It can only use up to 2300 gas and it reverts any transaction that exceeds this gas limit, as illustrated by [Solidity by Example](https://solidity-by-example.org/sending-ether/).

### Send

The send function is similar to transfer, but it differs in its behaviour:

bool success = payable(msg.sender).send(address(this).balance);

require(success, "Send failed");

Like transfer, send also has a gas limit of 2300. If the gas limit is reached, it will not revert the transaction but return a boolean value (true or false) to indicate the success or failure of the transaction. It is the developer's responsibility to handle failure correctly, and it's good practice to trigger a **revert** condition if the send returns false.

### Call

The call function is flexible and powerful. It can be used to call any function **without requiring its ABI**. It does not have a gas limit, and like send, it returns a boolean value instead of reverting like transfer.

(bool success, ) = payable(msg.sender).call{value: address(this).balance}("");

require(success, "Call failed");

To send funds using the call function, we convert the address of the receiver to payable and add the value inside curly brackets before the parameters passed.

The call function returns two variables: a boolean for success or failure, and a byte object which stores returned data if any.

👀❗**IMPORTANT**  
call is the recommended way of sending and receiving Ethereum or other blockchain native tokens.

### Conclusion

In conclusion, transfer, send, and call are three unique methods for transferring Ether in Solidity. They vary in their syntax, behaviour, and gas limits, each offering distinct advantages and drawbacks.

### 🧑‍💻 Test yourself

1. 📕 What are the primary differences between transfer, send, and call when transferring Ether?
2. 📕 Why is it necessary to convert an address to a payable type before sending Ether to it?
3. 🧑‍💻 Implement a function callAmountTo using call to send Ether from the contract to an address provided as an argument. Ensure the function handles failures appropriately.

# Smart contract constructor

This lesson focuses on using the constructor function in Solidity for role assignment, particularly for setting a contract owner. It discusses the security implications and demonstrates how to restrict certain functionalities, like fund withdrawal, to the owner.

#### Introduction

In this lesson, we will address a security gap present in the current fundMe contract.

### Constructor

Currently, **anyone** can call the withdraw function and drain all the funds from the contract. To fix this, we need to **restrict** the withdrawal function to the contract owner.

One solution could be to create a function, callMeRightAway, to assign the role of contract owner to the contract's creator immediately after deployment. However, this requires two transactions.

A more efficient solution is to use a **constructor** function:

constructor() {}

🗒️ **NOTE**  
The constructor does not use the function and public keywords.

### Assigning the Owner in the Constructor

The constructor function is automatically called during contract deployment, within the same transaction that deploys the contract.

We can use the constructor to set the contract's owner immediately after deployment:

address public owner;

constructor() {

owner = msg.sender;

}

Here, we initialize the state variable owner with the contract deployer's address (msg.sender).

### Modifying the Withdraw Function

The next step is to update the withdraw function to ensure it can only be called by the owner:

function withdraw() public {

require(msg.sender == owner, "must be owner");

// rest of the function here

}

Before executing any withdrawal actions, we check that msg.sender is the owner. If the caller is not the owner, the operation **reverts** with the error message "must be the owner" This access restriction ensures that only the intended account can execute the function.

### Conclusion

By incorporating a constructor to assign ownership and updating the withdraw function to restrict access, we have significantly improved the security of the fundMe contract. These changes ensure that only the contract owner can withdraw funds, preventing unauthorized access.

### 🧑‍💻 Test yourself

1. 📕 What is the purpose of a constructor function and how does it differ from regular functions?
2. 📕 Why is it necessary to restrict access to the withdraw function?
3. 🧑‍💻 Write a function called withdrawOnlyFirstAccountRemix that allows only the first Remix account to withdraw all funds from the contract.

# Solidity function modifiers

A deep dive into the use of function modifiers in Solidity. The lesson covers how modifiers can streamline code, especially for administrative functions, and includes practical examples to illustrate the implementation and benefits of using modifiers in contracts.

### Introduction

In this lesson, we will explore **modifiers** and how they can simplify code writing and management in Solidity. Modifiers enable developers to create reusable code snippets that can be applied to multiple functions, enhancing code readability, maintainability, and security.

### Repeated Conditions

If we build a contract with multiple administrative functions, that should only be executed by the contract owner, we might repeatedly check the caller identity:

require(msg.sender == owner, "Sender is not owner");

However, repeating this line in every function clutters the contract, making it harder to read, maintain, and debug.

### Modifiers

Modifiers in Solidity allow embedding **custom lines of code** within any function to modify its behaviour.

Here's how to create a modifier:

modifier onlyOwner {

require(msg.sender == owner, "Sender is not owner");

\_;

}

🗒️ **NOTE**  
The modifier is named onlyOwner to reflect the condition it checks.

### The \_ (underscore)

The underscore \_ placed in the body is a placeholder for the modified function's code. When the function with the modifier is called, the code before \_ runs first, and if it succeeds, the function's code executes next.

For example, the onlyOwner modifier can be applied to the withdraw function like this:

function withdraw(uint amount) public onlyOwner {

// Function logic

}

When withdraw is called, the contract first executes the onlyOwner modifier. If the require statement passes, the rest of the withdraw function executes.

If the underscore \_ were placed before the require statement, the function's logic would execute first, followed by the require check, which is not the intended use case.

### Conclusion

Using modifiers like onlyOwner simplifies contract development by centralizing common conditions, reducing code repetition, and enhancing contract readability and maintainability.

### 🧑‍💻 Test yourself

1. 📕 Why is it beneficial to use modifiers for access control?
2. 🧑‍💻 Implement a modifier named onlyAfter(uint256 \_time) that ensures a function can only be executed after a specified time.

# Test the smart contract with a testnet

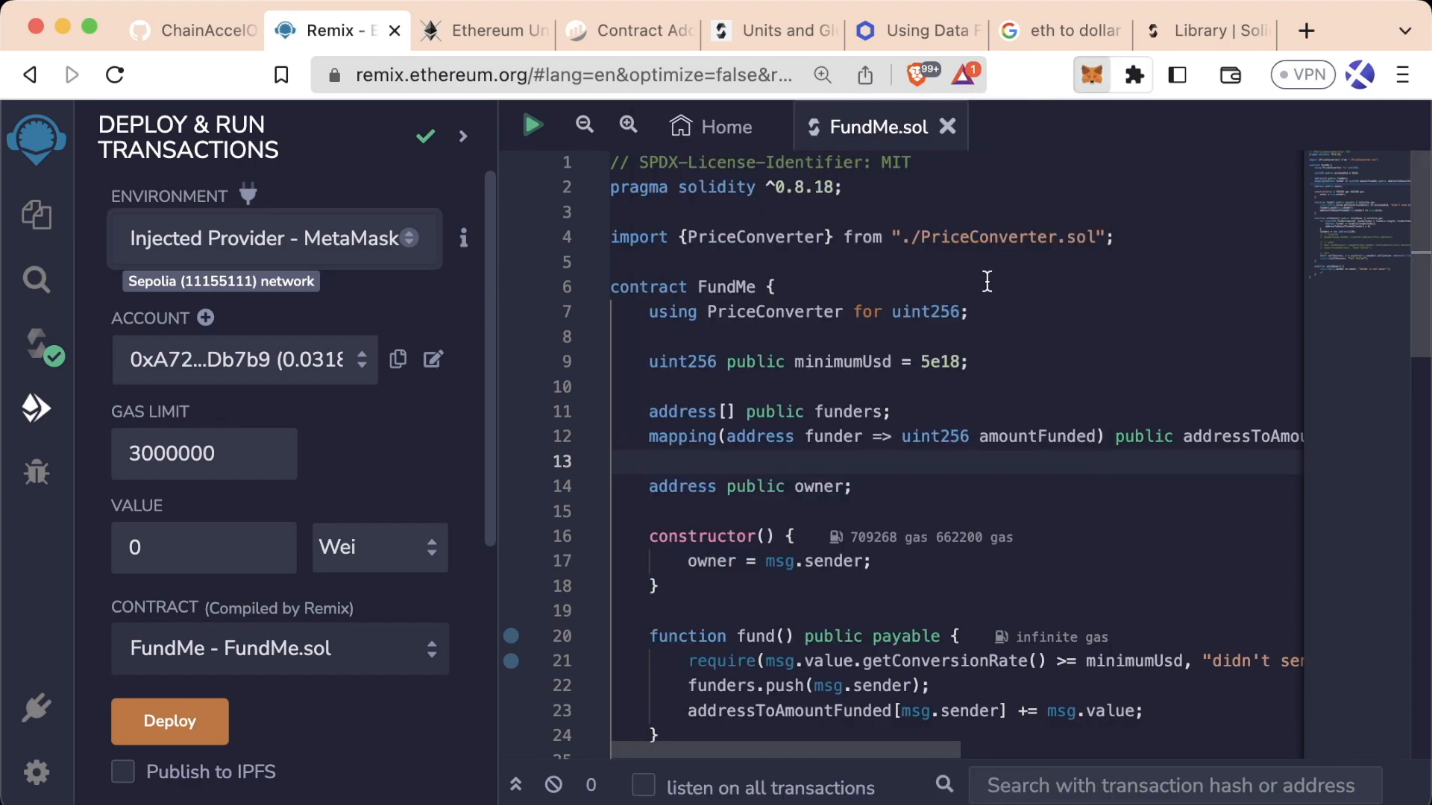
A guide to testing Solidity contracts and deploying to a testnet, focusing on compiling, deploying, and interacting with the 'FundMe.sol' contract. The lesson includes steps for using MetaMask, tracking transactions, and ensuring successful contract interaction.

### Introduction

In this lesson, we'll delve into end-to-end testing of a Solidity contract's deployment and execution.

### Deployment Transaction

First, we need to compile the contract to ensure the code is correct. On Remix, set the **injected provider** to MetaMask and confirm it is properly synced to the testnet. Ensure you have some Sepolia Ether (ETH) in your wallet if you plan to deploy the contract on Sepolia.



We'll deploy the FundMe contract by clicking deploy and then confirming the transaction in MetaMask, which may take some time.

### Contract Interaction

After successfully deploying the FundMe contract, you'll see several buttons to interact with it:

* **Red button**: Payable functions (e.g., fund)
* **Orange button**: Non-payable functions (e.g., withdraw)
* **Blue buttons**: view and pure functions

The fund function allows us to send ETH to the contract (minimum 5 USD). The owner of the contract is our MetaMask account, as the **constructor** sets the deployer as the owner.

🗒️ **NOTE**  
If the fund function is called without any value or with less than 5 USD, you will encounter a gas estimation error, indicating insufficient ETH, and gas will be wasted.

### Successful Transaction

If you set the amount to 0.1 ETH and confirm it in MetaMask, you can then track the successful transaction on Etherscan. In the Etherscan transaction log, you will see that the fundMe balance has increased by 0.1 ETH. The funders array will register your address, and the mapping addressToAmountFunded will record the amount of ETH sent.

### Withdraw Function and Errors

After funding the contract, we can initiate the withdraw function. This function can only be called by the owner; if a non-owner account attempts to withdraw, a gas estimation error will be thrown, and the function will revert.

Upon successful withdrawal, the fundMe balance, the addressToAmountFunded mapping, and the funders array will all reset to zero.

### Conclusion

In this lesson, we've explored the end-to-end process of deploying and interacting with a Solidity contract using Remix and MetaMask. We covered the deployment transaction, contract interaction, and how to handle successful transactions and potential errors.

### 🧑‍💻 Test yourself

1. 🧑‍💻 Interact with the FundMe contract on Remix and explore all possible outcomes that its functions can lead to.

# Immutability and constants

A tutorial on optimizing Solidity smart contracts for gas efficiency using custom errors. The lesson explains the concept of custom errors and demonstrates how to use them for efficient error handling and reverts in smart contracts.

### Introduction

In this lesson, we'll explore tools to optimize **gas usage** for variables that are set only once.

### Optimizing Variables

The variables owner and minimumUSD are set one time and they never change their value: owner is assigned during contract creation, and minimumUSD is initialized at the beginning of the contract.

### Evaluating the FundMe Contract

We can evaluate the gas used to create the contract by deploying it and observing the transaction in the terminal. In the original contract configuration, we spent almost 859,000 gas.

### Constant

To reduce gas usage, we can use the keywords constant and immutable. These keywords ensure the variable values remain unchanged. For more information, you can refer to the [Solidity documentation](https://solidity.readthedocs.io/).

We can apply these keywords to variables assigned once and never change. For values known at **compile time**, use the constant keyword. It prevents the variable from occupying a storage slot, making it cheaper and faster to read.

Using the constant keyword can save approximately 19,000 gas, which is close to the cost of sending ETH between two accounts.

🗒️ **NOTE**  
Naming conventions for constant are all caps with underscores in place of spaces (e.g., MINIMUM\_USD).

🚧 **WARNING**  
Converting the current ETH gas cost to USD, we see that when ETH is priced at 3000 USD, defining MINIMUM\_USD as a constant costs 9 USD, nearly 1 USD more than its public equivalent.

### Immutable

While constant variables are for values known at compile time, immutable can be used for variables set at deployment time that will not change. The naming convention for immutable variables is to add the prefix i\_ to the variable name (e.g., i\_owner).

Comparing gas usage after making owner an immutable variable, we observe similar gas savings to the constant keyword.

💡 **TIP**  
Don't focus too much on gas optimization at this early stage of learning.

### Conclusion

In this lesson, we have explored the use of constant and immutable keywords in Solidity to optimize gas usage for variables that are set only once. Understanding how and when to use these keywords can significantly reduce gas costs, making your smart contracts more efficient.

### 🧑‍💻 Test yourself

1. 📕 Why a developer can choose to use immutable instead of constant for specific variables?
2. 🧑‍💻 Invent one constant variable and one immutable variable that can be integrated into the current version of the fundMe contract.

# Creating custom errors

A tutorial on optimizing Solidity smart contracts for gas efficiency using custom errors. The lesson explains the concept of custom errors and demonstrates how to use them for efficient error handling and reverts in smart contracts.

### Introduction

In the previous lesson, we learned how to make our contracts more gas efficient. In this lesson, we will further enhance their efficiency.

### Require

One way to improve gas efficiency is by optimizing our require statements. Currently, the require statement forces us to store the string 'sender is not an owner'. Each character in this string is stored individually, making the logic to manage it complex and expensive.

### Custom Errors

Introduced in **Solidity 0.8.4**, custom errors can be used in revert statements. These errors should be declared at the top of the code and used in if statements. The cheaper error code is then called in place of the previous error message string, reducing gas costs.

We can start by creating a custom error:

error NotOwner();

Then, we can replace the require function with an if statement, using the revert function with the newly created error:

if (msg.sender != i\_owner) {

revert NotOwner();

}

By implementing custom errors, we reduce gas costs and simplify error handling in our smart contracts.

### Conclusion

In this lesson, we have learned how to further optimize gas efficiency in Solidity contracts by using custom errors instead of traditional require statements with strings.

### 🧑‍💻 Test yourself

1. 📕 What are the benefits of declaring custom errors instead of using the require keyword?
2. 🧑‍💻 Create a custom error that is triggered when msg.sender is address(0) and then convert it into an equivalent if statement with a revert function.

# Implementing the receive fallback

This lesson covers the implementation of '\_receive\_' and '\_fallback\_' functions in Solidity. It explains their significance in handling Ether sent directly to a contract and demonstrates their practical application in a 'FundMe' contract scenario.

### Introduction

In Solidity, if Ether is sent to a contract without a receive or fallback function, the transaction will be **rejected**, and the Ether will not be transferred. In this lesson, we'll explore how to handle this scenario effectively.

### receive and fallback functions

receive and fallback are special functions triggered when users send Ether directly to the contract or call non-existent functions. These functions do not return anything and must be declared external.

To illustrate, let's create a simple contract:

//SPDX-License-Identifier: MIT

pragma solidity ^0.8.7;

contract FallbackExample {

uint256 public result;

receive() external payable {

result = 1;

}

fallback() external payable {

result = 2;

}

}

In this contract, result is initialized to zero. When Ether is sent to the contract, the receive function is triggered, setting result to one. If a transaction includes **data** but the specified function does not exist, the fallback function will be triggered, setting result to two. For a comprehensive explanation, refer to [SolidityByExample](https://solidity-by-example.org/fallback/).

// Ether is sent to contract

// is msg.data empty?

// / \

// yes no

// / \

// receive()? fallback()

// / \

// yes no

// / \

//receive() fallback()

### Sending Ether to fundMe

When a user sends Ether **directly** to the fundMe contract without calling the fund function, the receive function can be used to redirect the transaction to the fund function:

receive() external payable {

fund();

}

fallback() external payable {

fund();

}

To test this functionality, send some Sepolia Ether to the fundMe contract using MetaMask. This does not directly call the fund function, but the receive function will trigger it. After confirming the transaction, you can check the funders array to see that it has been updated, reflecting the successful invocation of the fund function by the receive function.

This approach ensures that all transactions are processed as intended. Although directly calling the fund function costs less gas, this method ensures the user's contribution is properly acknowledged and credited.

### Conclusion

By implementing receive and fallback functions, contracts can handle direct Ether transfers and non-existent function calls effectively, ensuring that transactions are processed as intended and users' contributions are properly tracked.

### 🧑‍💻 Test yourself

1. 📕 How does the fallback function differ from the receive function?
2. 📕 What does it happen when Ether is sent with data but in the contract only a receive function exist?

# Congratulations

A recap of the advanced aspects of Solidity covered in previous lessons, highlighting the transition from using Remix to a code editor. The lesson congratulates learners on mastering Solidity basics and introduces upcoming advanced topics for further exploration.

### Introduction

In this second part of the FundMe section, we have covered the majority of Solidity basics, including special functions, custom errors, immutable variables, modifiers, constructors, arrays, for loops, libraries, and much more.

### Special Functions

We have encountered the special functions receive, fallback, and constructor. These functions do not require the function keyword before their name. The receive function is triggered when Ether is sent to a contract and the **data** field is empty. The fallback function is triggered when data is sent with a transaction, but no matching function is found.

### Saving Gas

To save gas, Solidity provides keywords like constant and immutable for variables that can only be set once:

uint constant minimumUSD = 50 \* 1e18;

In this example, minimumUSD is a constant and cannot be changed, saving gas. Unlike constant, which is set at compile time, immutable allows a variable to be assigned once during deployment. Attempts to change either constant or immutable variables will result in a compilation error.

### Sending Ether

Remix provides an easy method to send Ether to a contract. After deploying the contract, you can press the transact button, set the transaction's value, and omit the call data. If no call data is included, the receive function, if exists, will be triggered. Otherwise will be executed the fallback function.

### Conclusion

In the next section, we will move from Remix to a code editor to experiment with more advanced Solidity features. We will explore enums, events, try-catch, function selectors, abi.encode, hashing, Yul, and assembly.

### 🧑‍💻 Test yourself

1. 🏆 Attempt to answer all the theoretical questions from lessons 13 through 25, and then go back again to complete all the coding tasks.