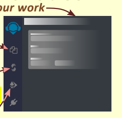
Intro & State Variables

A video discussing all of this material is available here: https://youtu.be/4XQsHBJScEk

variables declared outside a function are state variables, they are stored on the blockchain when they are **public**

variables declared inside a function are only available during a call of that function, and such variables are not stored on the blockchain

Smart contracts can be written and deployed to a test environment here: https://remix.ethereum.org The site cookies save your work

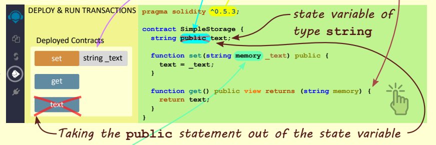


STEP 1: Write a smart contract in this file explorer

STEP 2: Compile the smart contract with compiler version compatable with this

STEP 3: Deploy the smart contract here ··· to see how it can be accessed on the blockchain with these options here

NOTE: If you hold down both Left-Alt and left-Shift at the same time, you can make the terminal font larger or smaller with the mouse wheel



```
pragma solidity ^0.5.3;

contract SimpleStorage {
  string public text;

  function set(string memory _text) public {
    text = _text;
  }

  function get() public view returns (string memory) {
    return text;
  }
}
```

state variable of type string

Taking the **public** statement out of the state variable will cause you to loose the **text** button on the blockchain

Variables in Solidity are stored in one of these three locations https://docs.soliditylang.org/en/v0.5.3/types.html

`storage`	this is where data can be permanently stored; these are state variables
`memory`	variables in memory are only available when the function is executing
`calldata`	special data location that contains the function arguments, only available for external function call parameters

Types of Functions in Solidity

- ones that create transactions
- ones that do not create transactions

functions that create transactions write data on the blockchain by changing the value of a state variable, which in turn either changes the state of the smart contract, or sends Ether (ETH) to another account, which in turn changes the balance of accounts recorded on the blockchain·
Functions that do not create any transaction on the blockchain are free to call (require no gas) and they do not change the state of the blockchain

This function has a single input of type **string**; for certain data types, you have to specifically declare the data location (here it must be **memory** because we (the users) are assigning the **text** value and we are outside the smart contract https://docs.soliditylang.org/en/v0.5.3/types.html#data-location

We need to declare the data location for our string type variable **text**; our **text** variable is stored in a contract storage, but need the actual value stored in the **text** variable, not the reference to the variable, so we will declare it as **memory**

```
pragma solidity ^0.5.3;

contract SimpleStorage {
  string public text;

  function set(string memory _text) public {
    text = _text;
  }

  function get() public view returns (string memory) {
    return text;
  }
}
```

this function will update the state variable

The convention in Solidity is to prefix a function input variable with an underscore "_" so as to avoid using the same names as state variables

We need to tell solidity that this function does not modify any state variables; we do this by using the keyword **view** or **pure**

If a function returns a value, it need a **returns** keyword in the function definition

view ensures that your function does not change the state of the blockchain, **pure** declares that your function neither changes the state of the blockchain nor does it read any state variables

view returns pure

When you want to return multiple values from a single function call you will use this option

Two ways to get state variables

1.) write your own function

2.) let Solidity compiler write it

Otherwise use this option

```
pragma solidity ^0.5.3;

contract SimpleStorage {
  string public text;

  function set(string memory _text) public {
    text = _text;
  }

  function get() public view returns (string memory) {
    return text;
  }
}
```

Use this one or this one in the smart contract, but don't use both of them

This example shows state variable (of type **string**) called **text** as well as a get function that returns the value of the same **text** variable; You would actually choose one of these two options but you do not need both!

Ether and Wei

A video discussing all of this material is available here: https://youtu.be/ybPQsjssyNw

The currency used within Ethereum is Ether; it can be used to:

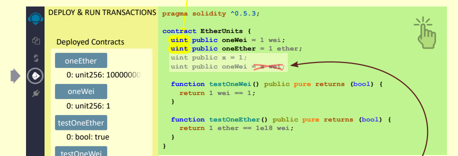
- pay block reward
- pay transaction fee
- transfer between accounts

1 Ether = 10^18 Wei

The smallest unit of Ether is Wei; 1 Ether is equal to 10^{18} Wei

= x wei

uint is an unsigned integer which means it cannot have a sign in front of it like "-" or "+" ··· this means that the integer cannot be negative



```
pragma solidity ^0.5.3;

contract EtherUnits {
  uint public oneWei = 1 wei;
  uint public oneEther = 1 ether;

  function testOneWei() public pure returns (bool) {
    return 1 wei == 1;
  }

  function testOneEther() public pure returns (bool) {
    return 1 ether == 1e18 wei;
  }
}
```

units can only be added after literal numbers; for example, **1 ether** is valid whereas **x wei** is not

giga-wei

Gwei stands for Giga-Wei and is equal to one billion wei; the most common situation where you would see the word Gwei is when you submit a transaction to the blockchain because for the transaction, Gwei is used to set the Gas price

Etherscan

If you head over to etherscan·io and look at any of the Latest Transactions, the Gas Price is listed in Gwei

Gas and Gas Price

A video discussing all of this material is available here: https://youtu.be/oTS9uxU6cAM

Transaction (Tx) fee = gas used * gas price (Gwei)

```
function update(newData) {
  uint x = 1 .......(might cost 8 gas)
  x + y .........(might cost 2 gas)
  return x ......(might cost 12 gas)
}
```

Each thing you do costs gas; the amount of gas required for each computation depends on what amount miners accept

Transaction (Tx)
Gas limit: 3,000
Gas Price: 2 gwei
Ether:
3,000 gas x 2 (gwei / gas)
= 6,000 gwei in weight

In this transaction, we are setting the gas limit to 3,000 and the gas price to 2 gwei, and thus, when this transaction is processed it will cost 6,000 gwei (3,000 x 2)

In abstract terms, let's assume 2 gwei is lower than the current average and we're specifying 2 gwei because we're not in a hurry to have our transaction added to the blockchain

Assuming you had 7,000 gwei in your Ether wallet

Your Account
Previous balance: 7,000 gwei
Transaction 'cost': 6,000 gwei
Current balance: 4,000 gwei

Assuming the transaction computations were such that they cost 1000 gas to process (per this) - then the cost would be 1000 gas * 2 gwei which would equal 2000 gwei, and that would mean that your account would be refunded 4,000 gwei since you effectively overpaid for the transaction and so the transaction 'cost' isn't the actual amount of gas that this whole thing actually costs you in the end·

happens when you overallocate gas

happens when you underallocate gas

An infinite loop would cost infinity gas to process; let's assume we alocate 6,000 gwei in weight (like we did here) and that each iteration happens to cost 1,800 gwei per supply and demand; here's what would happen:

```
function forever() {
  while (true) {
    ...
  }
}
```

after first iteration (- 1,800 gwei, 4,200 gas left)

after second iteration (- 1,800 gas, 2,400 gas left)

after third iteration (- 1,800 gas, 600 gas left)

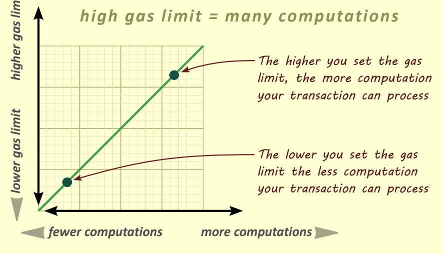
after forth iteration (- 1,800 gas, 0 gas left)

At every step of function execution gas is deducted until either the function finished execution or all of the gas is used up at which point the execution is avoided

not enough gas

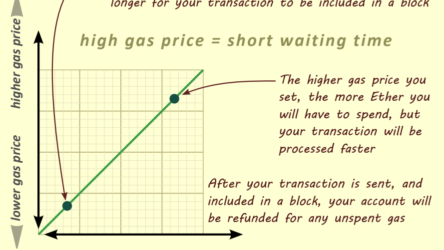
On the fourth iteration it uses up all remaining 600 gas midway through the iteration and then the function is forcefully stopped; **any changes that were made to a state variable will be undone, but you still have to pay for the gas spent!**

computations



tradeoffs

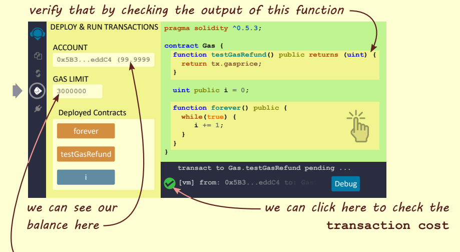
If you set the gas price low you'll have to pay less for your transaction, but you will have to wait longer for your transaction to be included in a block



There are two upper-bounds to limit the gas you can spend:

- Gas Limit - set by you
- Block Gas Limit - set by the network

When you send a transaction to the real Ethereum network you set the gas price, but gas price in remix is fixed at 1 wei, and we can verify that by checking the output of this function



```
pragma solidity ^0.5.3;

contract Gas {
  function testGasRefund() public returns (uint) {
    return tx.gasprice;
  }

  uint public i = 0;

  function forever() public {
    while(true) {
      i += 1;
    }
  }
}
```

we can see our balance here

we can click here to check the transaction cost

we can change the gas limit here

Invalid Functions

A video discussing all of this material is available here: https://youtu.be/71cmPaD_AnQ

Invalid Inputs & Outputs for Public Functions

- { } map
- [ ] [ ] multi-dimensional arrays (unfixed size)

In Solidity, there are certain data types that cannot be input variables in public functions (e·g· "{ }") and there are also data types that are not recommended (e·g· "[ ]") as well· This is also true for function outputs: "{ }" is not allowed and "[ ]" is not reccomended·

```
pragma solidity ^0.5.3;
pragma experimental ABIEncoderV2;

contract Function {
  function multiDimFixedSizeArrayInput(uint[9][9] memory _arr) public {
  }

  function multiDimDynamicSizeArrayInput(uint[][] memory _arr) public {
  }

  function arrayInput(uint[] memory _arr) public {
  }
}
```

inputs

If you get rid of this statement and replace it with this statement, then this function will compile; otherwise it will not compile; the other two functions will compile just fine without using this statement· It is not reccomended that you use any array as an input variable because different array sizes require different amounts of gas to process, and so sometimes the function will process just fine and other times it will run out of gas when using an array as an input variable·

```
pragma solidity ^0.5.3;

contract Function {
  uint MAX_ARR_LENGTH = 10;

  function arrayInput(uint[] memory _arr) public {
    if (_arr.length > MAX_ARR_LENGTH) {
      // throw error
    }
  }
}
```

One way to make an array more reliable (as an input variable) and avoid the problem with gas (discussed above) is to put an upper-bound to the array size that will in turn limit the amount of gas consumed

outputs

If we try using maps and/or multi-dimensional arrays of an unfixed size as function outputs, we will also get compile errors

```
pragma solidity ^0.5.3;

contract Function {
  mapping(uint => uint) map;
  uint[] arr;
  uint[3][3] arr2DFixed;
  uint[][] arr2D;

  function mapOutput() public returns (mapping(uint => uint) memory) {
    return map;
  }

  function multiDimFixedSizeArrayOutput() public view returns (uint[3][3] memory) {
    return arr2DFixed;
  }

  function multiDimDynamicSizeArrayOutput() public returns (uint[][] memory) {
    return arr2D;
  }

  function arrayOutput() public view returns (uint[] memory) {
    return arr;
  }
}
```

this function will not compile

this function will not compile

Using a one dimensional array with an unfixed size is not reccomended because your contract might get called by a second contract and so your contract's function output is another contract's function input; one way to solve these issues is to write functions that have a bounded consumption of gas

```
pragma solidity ^0.5.3;

contract Function {
  function returnMultipleVals() public pure returns (uint, bool, uint) {
    return (1, true, 2);
  }

  function named() public pure returns (uint x, bool b, uint y) {
    return (1, true, 2);
  }

  function assigned() public pure returns (uint x, bool b, uint y) {
    x = 1;
    b = true;
    y = 2;
  }
}
```

multiple un-named values

multiple named values

explicitly assigned to return variables and omit the last return statement

In Solidity, you can return multiple values, and those values can also be named; the options on the left are all valid ones

function that returns multiple functions

```
pragma solidity ^0.5.3;

contract Function {
  function firstFunc() public returns (uint) {
    return 1;
  }

  function secondFunc() public returns (uint) {
    return 2;
  }

  function thirdFunc() public returns (uint, uint) {
    return (firstFunc(), secondFunc());
  }
}
```

We can call **firstFunc** and then call **secondFunc** seperately, or we can just call **thirdFunc** instead

destructuring assignments

Destructuring assignments can be used to assign variables to the output of a function which returns multiple values

```
pragma solidity ^0.5.3;

contract Function {
  function returnMultipleVals() public pure returns (uint, bool, uint) {
    return (1, true, 2);
  }

  function destructuringAssignments() public pure returns (uint, bool, uint, uint, uint) {
    (uint i, bool b, uint j) = returnMultipleVals();

    (uint x, , uint y) = (4, 5, 6);

    return (i, b, j, x, y);
  }
}
```

Here we are assigning the outputs of the function **returnMultipleVals()** and the variable types declared here are consistant with the types of values here that are being returned by the function being called

If a function returns three parameters, but you don't care about the second one (in this case, the 5) then you can use destructuring by adding in an empty-space with a comma to let Solidity know to skip that value

Inheritance (Constructor-1)

A video discussing all of this material is available here: https://youtu.be/dcNpIOhTNQg

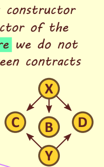
In Solidity, constructors are always executed in a certain order· You cannot override a state variable that is declared in a parent contract by simply redeclaring it in a child contract

```
pragma solidity ^0.5.3;

contract X {
  string public name;

  constructor(string memory _name) public {
    name = _name;
  }
}

contract Y {
  string public text;

  constructor(string memory _text) public {
    text = _text;
  }
}

contract B is X("Fixed input"), Y("Another fixed input") {
}

contract C is X, Y {
  constructor() X("Fixed input") Y("Another fixed input") public {
  }
}

contract D is X, Y {
  constructor(string memory _name, string memory _text) X(_name) Y(_text) public {
  }
}
```

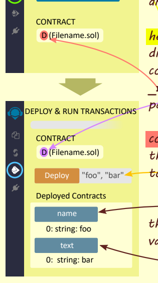
commas needed outside the constructor

no commas needed inside the constructor

this is one way to call the constructors of multiple parent contracts ··· this is another way to call the parent constructor inside the constructor of the child contract; here we do not put commas between contracts when listing them



this is how you pass variables to parent contracts, the syntax is similiar to this, except here we are passing in a fixed input and here we are passing in variables



here the constructor is accepting two different inputs to be passed to the parent constructors (X & Y) and here passes the **_name** variable to contract X and here it passes **_text** to contract Y

compile and then deploy contract D, and then upon deployment, set the **string_name** to **"foo"** and the **string_text** to **"bar"**

Calling the state variable **name** returns the string **"foo"** and calling the state variable **text** returns the string **"bar"**

View and Pure Functions

A video discussing all of this material is available here: https://youtu.be/xknoxALAL8c

- view functions do not modify the state of the blockchain
- pure functions do not modify the state of the blockchain, nor do they read the state either

According to the Solidity docs here, The following statements are considered modifying the state:

1· Writing to state variables
2· Emitting events - also knows as 'logging'
3· Creating other contracts
4· Using **selfdestruct**
5· Sending Ether via calls
6· Calling any function not marked **view** or **pure**
7· Using low-level calls
8· Using inline assembly that contains certain opcodes·

NOTE: this is a function to call when you want to delete your contract from the blockchain

pure makes a stronger statement than **view**

According to the Solidity docs here, the following are considered reading from the state:

1· Reading from state variables
2· Accessing **address(this).balance** or
3· **<address>.balance**
4· Accessing any of the members of **block**, **tx**, **msg**
5· (with the exception of **msg.sig** and **msg.data**)
Calling any function not marked **pure**
Using inline assembly that contains certain opcodes·

```
pragma solidity ^0.5.3;

contract ViewAndPure {
  uint public x = 1;

  function addToX(uint y) public view returns (uint) {
    return x + y;
  }

  function updateX() public view {
    x += 1;
  }

  function foo() public {}

  function viewFunc() public view {}

  function invalidViewFunc() public view {
    foo();
  }

  function invalidPureFunc() public pure {
    viewFunc();
  }

  function add(uint i, uint j) public pure returns (uint) {
    return i + j;
  }
}
```

will not compile

will not compile

this is a valid **view** function, but because it reads from the **state**, it could not be a **pure** function

A **view** function cannot call another function that is neither **view** or **pure**

This is an invalid **pure** function because pure functions cannot call a non-pure function, even if that function is a **view** function

this is a valid **pure** function since it doesn't read and/or modify any state

Function Modifiers

A video discussing all of this material is available here: https://youtu.be/thADMg9cKPM

Function Modifiers are used for:

- restricting write access
- input validation
- reentrancy guard

Function modifiers are reuseable code that can be attached to a function; this reuseable code can be executed before and or after the function itself is executed

when the contract is deployed, it sets the owner to **msg.sender**

```
pragma solidity ^0.5.3;

contract FuncModifier {
  address public owner;

  constructor() public {
    owner = msg.sender;
  }

  modifier onlyOwner() {
    require(msg.sender == owner, "Not owner");
    _;
  }

  modifier validAddress(address _addr) {
    require(_addr != address(0), "Not valid address");
    _;
  }

  function changeOwner(address _newOwner) public onlyOwner validAddress(_newOwner) {
    owner = _newOwner;
  }

  uint public x = 10;
  bool locked;

  modifier noReentrancy() {
    require(!locked, "Locked");

    locked = true;
    _;
    locked = false;
  }

  function decrement(uint i) public noReentrancy {
    x -= i;

    if (i > 1) {
      decrement(i - 1);
    }
  }
}
```

the "_" is a special character that you can only use in a function modifier; it tells Solidity to execute the rest of the code inside the parent function

if the message sender is not the current owner, the transaction fails

You attach a modifier to a function by declaring it here in the function signature

reentrancy guard

this modifier prevents recursive calls by first setting the locked to **false**, then when decrement is called, here it sets the locked to **true**, and it starts executing its code here, and next it calls itself again here but since locked is set to **true** at this point, the require statement fails here and the transaction therefore fails so as to prevent a reentrancy hack

Inheritance (Constructor-2)

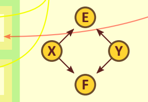
A video discussing all of this material is available here
this section starts at time: 3:26 / 8:09

Below we see in what order parent constructors will be called and executed; whereas on the right (Constructor-1) we detail how to pass input variables in a child constructor to a parent constructor

the order in which the parent constructors are called is determined by the inheritance order here and not by the order in which the parent constructors are called on these lines here

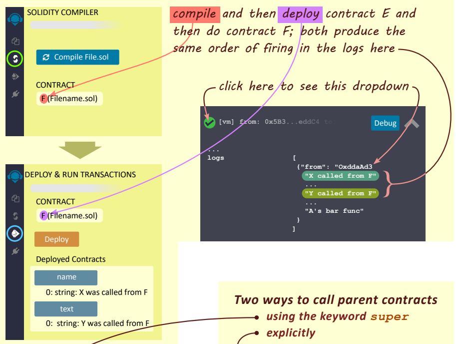
```
pragma solidity ^0.5.3;

contract X {
  event Log(string message);
  string public name;
  constructor(string memory _name) public {
    name = _name;
    emit Log(_name);
  }
}

contract Y {
  event Log(string message);
  string public text;
  constructor(string memory _text) public {
    text = _text;
    emit Log(_text);
  }
}

contract E is X, Y {
  constructor() X("X called from E") Y("Y called from E") public {
  }
}

contract F is X, Y {
  constructor() Y("Y called from F") X("X called from F") public {
  }
}
```

NOTE: constructors inherit opposite of functions



compile and then deploy contract E and then do contract F; both produce the same order of firing in the logs here

click here to see this dropdown



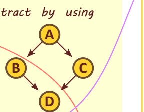
Two ways to call parent contracts
- using the keyword **super**
- explicitly

Inheritance (Super)

A video discussing all of this material is available here
this section starts at time: 8:42 / 12:37

```
pragma solidity ^0.5.3;

contract A {
  event MyLogEvent(string message);

  function foo() public {
    emit MyLogEvent("A's foo func");
  }

  function bar() public {
    emit MyLogEvent("A's bar func");
  }
}

contract B is A {
  function foo() public {
    emit MyLogEvent("B.foo called");
    A.foo();
  }

  function bar() public {
    emit MyLogEvent("B's bar func");
    super.bar();
  }
}

contract C is A {
  function foo() public {
    emit MyLogEvent("C's foo func");
    A.foo();
  }

  function bar() public {
    emit MyLogEvent("C's bar func");
    super.bar();
  }
}

contract D is B, C {
}
```

This example on the left shows how the parent (more baselike) functions will be called when using the keyword super; contracts A, B, & C first emit an event then call the parent contract by using the keyword super

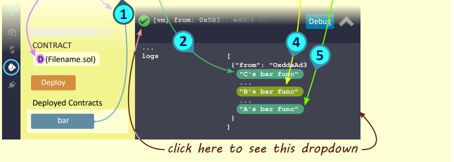


super

in this example here the parent contract is called directly whereas here, **super** is used to call all parent contracts in the order of inheritance because the **bar()** function in contract A is being overiden by the **bar()** functions in contracts B and C

emit

here, X, Y, & Z are hypothetical

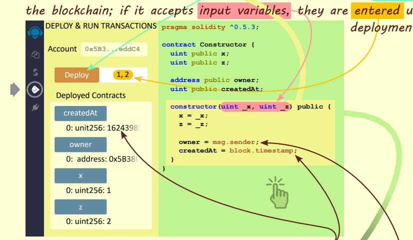


click here to see this dropdown

Constructor

A video discussing all of this material is available here: https://youtu.be/HpJZZ9tASGs

a **constructor** is an optional function that is executed only once when the contract is initially deployed to the blockchain; if it accepts input variables, they are entered upon deployment



```
pragma solidity ^0.5.3;

contract Constructor {
  uint public x;
  uint public y;

  address public owner;
  uint public createdAt;

  constructor(uint _x, uint _y) public {
    x = _x;
    y = _y;

    owner = msg.sender;
    createdAt = block.timestamp;
  }
}
```

this will give you the unix timestamp for when the contract was deployed to the blockchain

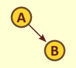
constructor() has access to special variables like **msg.sender** & **block.timestamp**

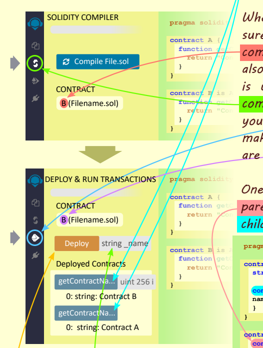
Inheritance (Override-1)

A video discussing all of this material is available here: https://youtu.be/GntHR8-1o8

contract B is inheriting from contract A, and they both have a function with the same name; normally, the function in contract B would override the one in contract A if they have the same name, but here, contract B's function accepts an input variable whereas contract A's function does not; therefore, contract B will have two functions of the same name available to it when deployed·

```
pragma solidity ^0.5.3;

contract A {
  function getContractName() public pure returns (string memory) {
    return "Contract A";
  }
}

contract B is A {
  function getContractName(uint i) public pure returns (string memory) {
    return "Contract B";
  }
}
```





When using the Remix IDE, make sure to specify that you are compiling contract B (you can also compile contract A, but it is unnecessary here) within the compiler window; likewise, when you go to the deploy window, make sure to specify that you are deploying contract B

One way to pass variables from a parent constructor to a child constructor is like this ···

```
pragma solidity ^0.5.3;

contract A {
  string public name;

  constructor(string memory _name) public {
    name = _name;
  }
}

contract B is A {
  constructor() A("Contract B") public {
  }
}
```

···but another way to pass such variables is like this

```
contract B is A {
  constructor(string memory _name) A(_name) public {
  }
}
```

···the later case is used when you want to set the variable upon the initial deployment of the smart contract such that you would define the variable here upon deployment - this input field will not exist if you compile the version of contract B shown here

Inheritance (Order)

A video discussing all of this material is available here: https://youtu.be/Q5_Gmm_IZSI

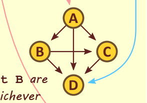
```
pragma solidity ^0.5.3;

contract A {
  function foo() public pure returns (string memory) {
    return "A";
  }
}

contract B is A {
  function foo() public pure returns (string memory) {
    return "B";
  }
}

contract C is A, B {
  // foo returns "B"
}

contract C is B, A {
  // foo returns "A"
}

contract D is C, B, A {
  // foo returns "A"
}
```

most baselike (doesn't inherit)

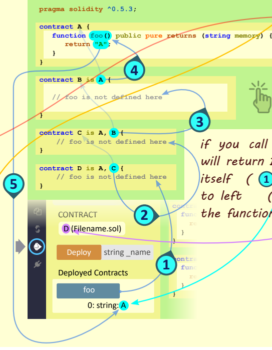
most derived (inherits the most)

The order of inheritance determines which contract is inherited; according to the docs, parent contracts must be ordered from most baselike, to most derived (the reverse in Python)

meaning that baselike = less inheritance while derived = more inheritance; this gives you this and vice-versa

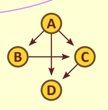


In the example above, contract A and contract B are both equally baselike, so contract C inherits whichever one is listed last in the order; if this order has contracts listed out of order (from most base-like to most derived) then the Solidity compiler might throw an error, otherwise the order of inheritance will go from right to left for functions, but left to right (the exact opposite) for any constructors which inherit from left to right



depth first

Solidity will perform a depth-first search when inheriting, such that in the example on the left, if you call **foo()** on **contract D** it will return A because it will first look in itself (①) then search from right to left (② ③ ④) until it finds the function that it is looking for (⑤)



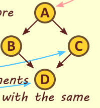
Inheritance (Events)

A video discussing all of this material is available here
this section starts at time: 6:56 / 12:37

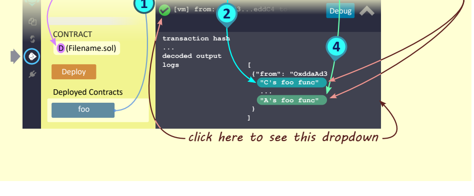
Solidity uses events to log things on the blockchain, and applications can be notified when events have been emitted so that those same applications don't have to constantly monitor the contract (on the blockchain) to see if such changes have occured

emit

Events are declared with the **event** keyword and are capitalized by convention; you can pass parameters that you want the event to log in the most baselike contract where the event is found and those parameters will get passed to the more derived contracts downstream



Unlike **return** statements to a function call, emitted events with the same name will get called twice, and not be overidden



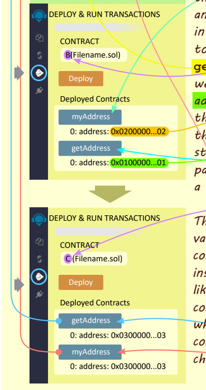
click here to see this dropdown

Inheritance (Override-2)

A video discussing all of this material is available here
this section starts at time: 5:05 / 8:09

Here is an example of how not to override a variable in a parent contract

```
pragma solidity ^0.5.11;

contract A {
  address public myAddress = 0x010...01;

  function getAddress() public view returns (address) {
    return myAddress;
  }
}

contract B is A {
  address public myAddress = 0x020...02;
}

contract C is A {
  constructor() public {
    myAddress = 0x030...03;
  }
}
```



If we deploy contract B and then call **myAddress** here, we get the address that is redefined here in contract B, and not the original definition found here in contract A - this is what we want to get, but if we were to then call **getAddress()** from contract B here, we would get original definition for address found in contract A, instead of the redefined address in contract B this is because you cannot override a state variable that is declared in a parent contract by redeclaring it inside of a child contract

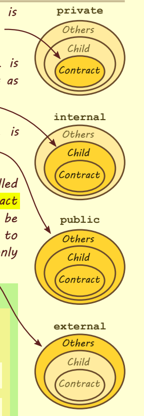
The correct way to override state variables (that are defined in a parent contract) is to redefine such variables inside a child contract's **constructor()** like this so when you deploy such a contract, the redefinition will hold whether or not it is called in a parent contract's function like this or in the child contract like this; in both cases the

Visibility

A video discussing all of this material is available here: https://youtu.be/NBzQVJ6OrrQ

These are the 4 key words that determine the visibility of state variables and functions:

```
private - visible inside the contract
internal - visible inside the contract and child contract(s)
  public - visible to everyone
external - visible to other contracts and accounts
```



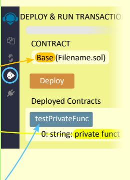
A state variable or function declared **private** is only accessable to **the contract** that defines it

A state variable or function declared **internal** is only accessable to **the contract** that defines it as well as any **child contracts** it has

A state variable or function declared **public** is accessable by any **contract** and any **account**

A function declared **external** can only be called by other contracts and accounts, not the contract it is defined in; a state variable like this cannot be declared **external** since it doesn't make sense to have a state variable that stores data that is only accessable to other contracts and accounts

```
pragma solidity ^0.5.11;

contract Base {
  string private privateVar = "my private variable";
  string internal internalVar = "my internal variable";
  string public publicVar = "my public variable";
  string external externalVar = "my external variable";

  function privateFunc() private pure returns (string memory) {
    return "private function called";
  }

  function testPrivateFunc() public pure returns (string memory) {
    return privateFunc();
  }

  function internalFunc() internal pure returns (string memory) {
    return "internal function called";
  }

  function testInternalFunc() public pure returns (string memory) {
    return internalFunc();
  }

  function publicFunc() public pure returns (string memory) {
    return "public function called";
  }

  function externalFunc() external pure returns (string memory) {
    return "external function called";
  }

  function testExternalFunc() public pure returns (string memory) {
    return externalFunc();
  }
}

contract Child is Base {
  function testInheritedFunc() public pure returns (string memory) {
    return privateFunc(); internalFunc();
  }
}
```



external public private internal

If you deploy this contract, you can not call this private function here from outside the contract; but if you deploy the same contract with a private function nested inside another public function like this; then that private function will be accessable here from outside the contract·

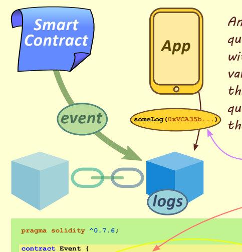
Although **private state variables** are only accessable inside the contract, you should not store sensitive information inside them because everything on the blockchain is visible to everyone; the same could be said for **internal state variables**·

A child function can not inherit a private function from a parent contract because a private function is only visible in the contract it is defined in; an internal function can be inherited by the child contracts of the contract in which the function was defined

Events

A video discussing all of this material is available here: https://youtu.be/BduMOagAuKs

Smart contracts can log that something has happened on the blockchain by firing events; applications can be automatically notified when events happen so they do not have to monitor a contract on a blockchain to see what state changes have occured



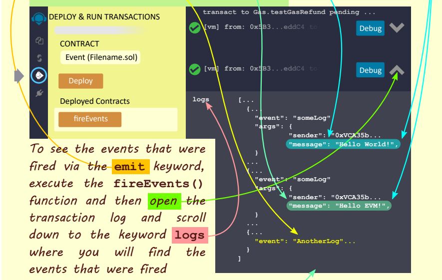
Anyone on the blockchain can query a smart contract's logs with input variable(s) if such variables are declared **indexed**· this keyword allows anyone to query a log for specific event(s) that match such criteria

this query would return these

indexed

It is possible to create an event where no parameter is passed

```
pragma solidity ^0.7.6;

contract Event {
  event someLog(address indexed sender, string message);
  event AnotherLog();

  function fireEvents() public {
    emit someLog(msg.sender, "Hello World!");
    emit someLog(msg.sender, "Hello EVM!");
    emit AnotherLog();
  }
}
```



To see the events that were fired via the **emit** keyword, execute the **fireEvents()** function and then open the transaction log and scroll down to the keyword **logs** where you will find the events that were fired

These logs will be stored on the blockchain but smart contracts will not be able to access information on these logs; the purpose of emitting events is to store information on the blockchain that smart contracts do not need access to, because storing such information on logs is cheaper than storing it on smart contracts

Error

A video discussing all of this material is available here: https://youtu.be/k4mdv97up5M

Sometimes you want your function to fail if something is not right

In Solidity, there are 3 ways to throw an error which would stop the further execution of the function being called:

assert - used to check for conditions that should never be possible; should always evaluate to true
● uses up all the gas

require - used to validate inputs, state conditions before execution, and to check outputs of other functions
● does not use up all the gas

revert - will only accept one argument (the error message to be thrown) whereas **require** will accept two;
● does not use up all the gas

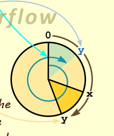
When a function throws an error, it will undo all of the changes that were made during the current call; Solidity errors **'bubble-up'** (like in Python or JavaScript) meaning that if function x calls another function y, and y throws an error, then x will also fail and all of the changes that were made during the call of x and y will be undone·

```
pragma solidity ^0.5.11;

contract Account {
  uint public balance;
  uint public constant MAX_UINT = 2 ** 256 -1;

  function deposit(uint _amount) public {
    uint oldBalance = balance;
    uint newBalance = balance + _amount;
    require(newBalance >= oldBalance, "Overflow");
    balance = newBalance;
    assert(balance >= oldBalance);
  }

  function withdraw(uint _amount) public {
    uint oldBalance = balance;
    require(balance >= _amount, "Underflow");

    if (balance < _amount) {
      revert("Underflow");
    }

    balance -= _amount;
    assert(balance <= oldBalance);
  }
}
```

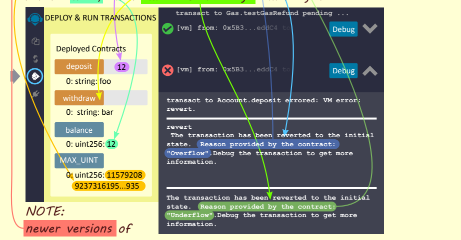
overflow

This is the largest number possible for the data type **uint** in Solidity; when you exceed it, the number 'overflows' back to 0 like on a clock· Likewise, a number 'underflows' when it reaches the lowest possible number for datatype uint, which is 0

underflow



If a number is deposited here and then the max number is added to it, than this error message will be generated; if the amount of money withdrawn here is greater than the balance here, then this error message will be generated



```
transact to Account.deposit errored: VM error:
revert.

revert
The transaction has been reverted to the initial
state. Reason provided by the contract:
"Overflow". Debug the transaction to get more
information.

The transaction has been reverted to the initial
state. Reason provided by the contract:
"Underflow". Debug the transaction to get more
information.
```

NOTE: newer versions of Solidity will have different error messages as they auto-detect discrepancies in Ether sent from an account vs· Ether remaining in that same account

Looping

A video discussing all of this material is available here: https://youtu.be/hzxdGvcsDvU

Loops in Solidity are structured just like they are in JavaScript, but in Solidity you want to make sure that your loops have an upper bound so they do not run for more iterations than you are expecting them to because that would use more gas than you were expecting to be used; this array has 100 elements, and the loop is limited to 100 iterations here, this is how loops in Solidity should be structured as opposed to leaving them open like this where you can have unlimited iterations

```
pragma solidity ^0.6.6;

contract Loop {
  uint public count;

  function loop(uint n) public {
    for (uint i = 0; i < n; i++) {
      count += 1;
    }
  }
}

contract Dividend {
  address[100] public shareholders;

  function pay() public {
    for (uint i = 0; i < shareholders.length; i++) {
      // send Ether to each shareholder
    }
  }
}
```

Arrays

A video discussing all of this material is available here: https://youtu.be/Vx0AoK7Vauw

arrays are declared with type, accessibility, and name; they can be either dynamically sized like this or fixed sized like this the actual size (for the latter case) is defined inside the brackets (in this case, size = 10) once the size of the array has been declared, it cannot be changed; the default value for each array element is 0 so this array will have 10 zeros in it; we can also create a dynamic array with default values stored in it like this and the number of elements in it can be changed

```
pragma solidity ^0.5.11;

contract Array {
  uint[] public myArray;
  uint[] public myArray2 = [1, 2, 3];
  uint[10] public myFixedSizeArray;

  function push(uint i) public {
    myArray.push(i);
  }

  function pop() public {
    myArray.pop();
  }

  function getLength() public view returns (uint) {
    return myArray.length;
  }

  function remove(uint index) public {
    delete myArray[index];
  }
  // [1, 2, 3, 4].remove(1) = [1, 0, 3, 4]
}

contract CompactArray {
  uint[] public myArray;

  function remove(uint index) public {
    myArray[index] = myArray[myArray.length - 1];
    myArray.pop();
  }
  // [1, 2, 3, 4].remove(1) = [1, 4, 3]

  function test() public {
    myArray.push(1);
    myArray.push(2);
    myArray.push(3);
    myArray.push(4);

    remove(1);

    assert(myArray.length == 3);
    assert(myArray[0] == 1);
    assert(myArray[1] == 4);
    assert(myArray[2] == 3);

    remove(2);
  }
}
```

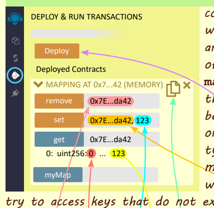
In Solidity, there are three built-in functionalities: **push()** is a function that adds an element to the end of an array, **pop()** is a function that removes the last element in the array; **.length** is a property that stores the length of an array

Solidity does not have an **unshift()** or or **splice()** method like JavaScript, so a custom function will be needed to **delete** an element at a specified index, but this would set the element's value to **0**; to keep arrays compact, we can grab the last element in an array and set it to the index we want deleted, then delete the last element after that

Mapping

A video discussing all of this material is available here: https://youtu.be/tO3vVMCOts8

```
pragma solidity ^0.5.11;

contract Mapping {
  mapping(address => uint) public myMap;

  function get(address _addr) public view returns (uint) {
    return myMap[_addr];
  }

  function set(address _addr, uint _i) public {
    myMap[_addr] = _i;
  }

  function remove(address _addr) public {
    delete myMap[_addr];
  }
}
```

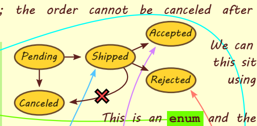


A mapping in Solidity is like a Dictionary in Python or a Map in JavaScript; however, in Solidity, you cannot iterate through a mapping and in order to create a mapping that you could interate on, you would have to use a combination of mapping and arrays wherein you store the keys in an array; you also cannot get the size of a mapping; the format is: **mapping(keytype, valuetype)** the keytype for a mapping has to be a built-in value type, a string, or bytes; the valuetype can be any type including an array or another mapping· Unlike arrays, mappings will will not return errors when you try to access keys that do not exist, instead the default value of 0 will be returned; as with arrays, the **delete** keyword does not remove an element in a mapping, but instead sets its value to 0; To see all this, compile and deploy this contract on Remix and then copy the contract address here and paste it here with a uint value here, then hit the set button· After that, hit the get button and you will see this uint value here, but if you copy the address and paste it here, then hit the remove button after that, when you hit the get button the value will be 0 instead of 123

Enums

A video discussing all of this material is available here: https://youtu.be/yJbx07N15j0

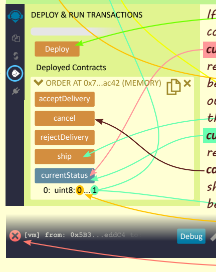
In Solidity, Enums are a data type used to create a custom-defined type; imagine we have an online bookstore in which the status of an order starts as pending, once it is pending, it is either canceled or shipped, if it is shipped the order can be either accepted or rejected after that; the order cannot be canceled after it is shipped·



We can model this situation using enum

```
pragma solidity ^0.5.12;

contract Order {
  enum Status {
    Pending,
    Shipped,
    Accepted,
    Rejected,
    Canceled
  }

  Status public currentStatus;

  function ship() public {
    require(currentStatus == Status.Pending);
    currentStatus = Status.Shipped;
  }

  function acceptDelivery() public {
    require(currentStatus == Status.Shipped);
    currentStatus = Status.Accepted;
  }

  function rejectDelivery() public {
    require(currentStatus == Status.Shipped);
    currentStatus = Status.Rejected;
  }

  function cancel() public {
    require(currentStatus == Status.Pending);
    currentStatus = Status.Canceled;
  }
}
```

This is an **enum** and the name of this type is **Status**; we store this Status in a public state variable & declare a name for it to be stored in; the default value of the enum will be the first element defined in the enum; The functions on the left ensure the flows shown in the diagram above are mantained, in example; the status cannot be accepted or rejected if it is not currently shipped·



If you compile and deploy the contract on Remix then click on **currentStatus**, you will get 0 because the 0 is the first index of our first (and default) element in the enum; if you call ship and then **currentStatus** after that, the value returned will be 1; if you try to **cancel** an order after it has been shipped, the transaction will fail because an order cannot be canceled unless it is still pending per this require statement

Structs

A video discussing all of this material is available here: https://youtu.be/kYBHq7EmFBc

```
pragma solidity ^0.5.3;

contract Todos {
  struct Todo {
    string text;
    bool completed;
  }

  Todo[] public todos;

  function create(string memory _text) public {
    todos.push(Todo(_text, false));

    todos.push(Todo({
      text: _text,
      completed: false
    }));

    Todo memory todo;
    todo.text = _text;
  }

  function get(uint _index) public view returns (string memory text, bool completed) {
    Todo storage todo = todos[_index];
    return (todo.text, todo.completed);
  }

  function update(uint _index, string memory _text) public {
    Todo storage todo = todos[_index];
    todo.text = _text;
  }

  function toggleCompleted(uint _index) public {
    Todo storage todo = todos[_index];
    todo.completed = !todo.completed;
  }
}
```

initialization method 1

initialization method 2

initialization method 3

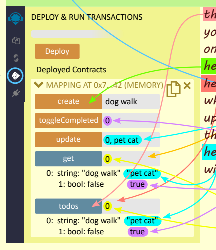
this gets the Todo's out of the array

this updates a selected Todo

this reverses the completed status

A struct is used to define a custom data type; the naming uses CapWords & these are called fields; here we create an array consisting of the custom **Todo** (struct) data type and this function will create a new instance of the **Todo** struct then it will **.push()** that new instance into this array: This order must match this order, but here you the order doesn't matter; here you can set only some field attributes and the rest will initialize with default values

You don't need to make a function like this because Solidity automatically creates getter functions like this when a struct gets created; If you compile and deploy the contract on Remix, you can create a struct here then get it returned here or here by querying the index (0) where it occurs in this array; updating the text here will change the text being returned to these here while toggling the boolean here will change the return values here



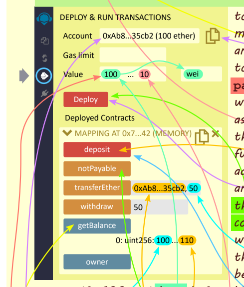
struct

Payable

A video discussing all of this material is available here: https://youtu.be/yD9EL1QN40Q

Contracts in Solidity can send and receive Ether by using the keyword **payable**; In order for this deposit function to be able to receive Ether into this (Wallet) contract, it must be declared **payable**; the sender is stored in a global variable called **msg.sender** and **msg.value** will hold the amount of Ether that was sent with this transaction, & the current balance is stored using **this** which is casted into an **address** data type, where the **this** refers to **this contract**

```
pragma solidity ^0.5.3;

contract Wallet {
  event Deposit(address sender, uint amount, uint balance);
  event Withdraw(uint amount, uint balance);
  event Transfer(address to, uint amount, uint balance);

  address payable public owner;

  constructor() public payable {
    owner = msg.sender;
  }

  function deposit() public payable {
    emit Deposit(msg.sender, msg.value, address(this).balance);
  }

  function notPayable() public {
    // cannot be paid because this is not "payable"
  }

  modifier onlyOwner() {
    require(msg.sender == owner, "Not owner");
    _;
  }

  function withdraw(uint _amount) public onlyOwner {
    owner.transfer(_amount);
    emit Withdraw(_amount, address(this).balance);
  }

  function transferEther(address payable _to, uint _amount) public onlyOwner {
    _to.transfer(_amount);
    emit Transfer(_to, _amount, address(this).balance);
  }

  function getBalance() public view returns (uint) {
    return address(this).balance;
  }
}
```



we only want the owner of this contract to be able to send Ether so we will use this modifier to ensure that is the case; and in order to do that we need to define the owner here as a **payable** state variable and then when the contract is deployed we assign the owner to the sender of the transaction here, this built-in function is only available to payable addresses like this; if you compile and deploy this contract on Remix, this button will be red because the constructor is payable which means we can send Ether when we deploy this contract; this button is red because its function is payable; if we specify 100 wei here before hitting deploy, the balance here will be 100; if we then enter 10 here and hit deposit, then hit the getBalance button, we will have 110 wei; if we enter any value of Ether here then click the notPayable button, the transaction will not process and an error message will occur in the log because this function is not defined as payable; only the owner of this contract can transfer Ether using **transferEther**; to do this select account #2 here then copy the #2 address here, then switch back to account #1 (the owner of this contract) here after that; next, paste in the account #2 address here and the amount you want to transfer here

Sending Ether

A video discussing all of this material is available here: https://youtu.be/_5vGaqgzlG8

There are 3 methods to send Ether from one contract to another contract: this is enough to emit a log, but not much else

transfer - forwards no more than 2300 gas, throws error

send - forwards no more than 2300 gas, returns boolean value

call - forwards all gas or sets gas, returns boolean value
recommended way after December 2019 because in the future, the cost of gas can change for different operations

NOTE: newer versions of Solidity require the use of the keyword **fallback** in lieu of **function** per the latest docs

Whenever a function receives Ether from a contract using any of the 3 methods above, it calls a special function called a **fallback** function & when the **transfer** or **send** methods are used, that **fallback** function (which can be declared like this) has at most 2300 gas to spend; such a function cannot have a name, take any inputs, or return any outputs; its visibility must be **external** and in order to receive Ether, it must be declared **payable** as well·

```
pragma solidity ^0.5.11;

contract ReceiveEther {
  function() external payable {}

  function getBalance() public view returns (uint) {
    return address(this).balance;
  }
}

contract SendEther {
  function sendViaTransfer(address payable _to) public payable {
    _to.transfer(msg.value);
  }

  function sendViaSend(address payable _to) public payable {
    bool sent = _to.send(msg.value);
    require(sent, "Failed to send Ether");
  }

  function sendViaCall(address payable _to) public payable {
    (bool sent, bytes memory data) = _to.call.value(msg.value)("");
    require(sent, "Failed to send Ether");
  }
}
```

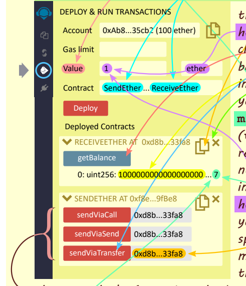
using the transfer method

using the send method

using the call method

this helper function lets us check the balance of Ether in the contract where **this** refers to this contract

These values are equal to the values specified here; if you deploy both contracts than copy this address and paste it here then specifying 1 Ether over here and then click here, if you click here after that, you will see a balance of 1 Ether (10^18 in gas) in the **ReceiveEther** contract; if you replace any of these **msg.value** statements with a **(uint)** number value like 7 and re-deploy the contracts, that number value will appear here instead of the value you specify up here, but on Remix, the first time you try this you will have to specify a value here or an error message will occur, after the first time however, you can call any of these methods (assuming they've all had **msg.value** changed to 7 in the code above) without specifying an amount here and the encoded value will show up here·



```
pragma solidity ^0.8.6;
...
contract SendEther {
  function sendViaDynamicCall(address payable _to) public payable {
    (bool sent, bytes memory data) = _to.call{value: msg.value}("");
    require(sent, "Failed to send Ether");
  }

  function sendViaHardCodedCall(address payable _to) public payable {
    (bool sent, bytes memory data) = _to.call{value: 7}("");
    require(sent, "Failed to send Ether");
  }
}
```

using the call method

using the call method

Newer versions of Solidity require a different syntax like this here but note that setting dynamic values in this way does not work as this **sendViaHardCodedCall()** function will fail because this does not work; most functions use **msg.value** as a matter of convention

this function returns 2 values, a boolean indicating success or failure and the values from calling the fallback function - this is an empty value since the fallback cannot return anything (i·e· 0 bytes)

Fallback Function

A video discussing all of this material is available here: https://youtu.be/3FY9FkJKSSI

Fallback functions like this have no names, inputs, or outputs and must be declared **external**; you can write regular code inside the fallback function like this but it is reccomended that you limit the amount of code in fallback functions because the function can fail if it uses too much gas which means you won't be able to send Ether to a contract using the **send** or **transfer** methods; fallback functions are called in one of two possible situations:

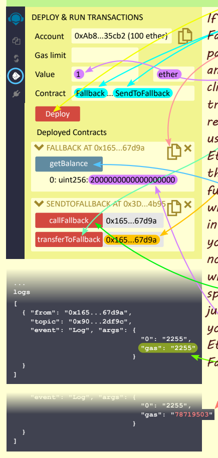
gasleft

1· When you call a function that does not exist in the host function

2· When you send Ether to the host contract using the **send**, **transfer**, or **call** methods

```
pragma solidity ^0.5.11;

contract Fallback {
  event Log(uint gas);

  function() external payable {
    emit Log(gasleft());
  }

  function getBalance() public view returns (uint) {
    return address(this).balance;
  }
}

contract SendToFallback {
  function transferToFallback(address payable _to) public payable {
    _to.transfer(msg.value);
  }

  function callFallback(address payable _to) public payable {
    (bool sent,) = _to.call.value(msg.value)("");
    require(sent, "Failed to send Ether");
  }
}
```

Deploy both contracts here



If you copy the address of the Fallback function here and then paste it here then set the transfer amount to 1 Ether here and then click here, you will see that in the transaction log that 2255 gas were recorded here; calling this function used the transfer method to send Ether to the Fallback function and this in turn triggered the fallback function inside the Fallback contract which logged the gas left here that in turned displayed this here; If you repeat the steps above but now run this function instead, when you look in the log it will spend significantly more gas than just 2300; after doing all that, if you click here you should see 2 Ether (10^18 x 2 gas) in the Fallback contract here

Call

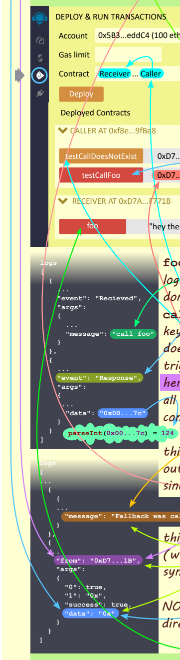
A video discussing all of this material is available here: https://youtu.be/mz10sUmEdsM

This example below shows **contract Caller** interacting with **contract Reciever** (sic) in a way that is not recomended, but nontheless demonstrates how to use the **call()** method; the Caller contract will be calling on this function inside the Reciever contract·

This event will track which functions were called; when the **foo()** function is called, the event here will send data to the logs

```
pragma solidity ^0.8.7;

contract Reciever {
  event Recieved(address caller, uint amount, string message);

  fallback () external payable {
    emit Recieved(msg.sender, msg.value, "Fallback was called");
  }

  function foo(string memory _message, uint _x) public payable returns (uint) {
    emit Recieved(msg.sender, msg.value, _message);

    return _x + 1;
  }
}

contract Caller {
  event Response(bool success, bytes data);

  function testCallFoo(address payable _addr) public payable {
    (bool success, bytes memory data) = _addr.call{value: msg.value, gas: 5000}(
      abi.encodeWithSignature("foo(string,uint256)", "call foo", 123)
    );

    emit Response(success, data);
  }

  function testCallDoesNotExist(address _addr) public {
    (bool success, bytes memory data) = _addr.call(
      abi.encodeWithSignature("doesNotExist()")
    );

    emit Response(success, data);
  }
}
```

the **call()** method is able to capture the return value



```
logs
[
  {
    ...
    "event": "Recieved",
    "args":
    {
      ...
      "message": "call foo"
    }
  },
  {
    ...
    "event": "Response",
    "args":
    {
      ...
      "data": "0x00...7c"
    }
  }
]
parseInt(0x00...7c) = 124

logs
[
  {
    ...
    "message": "Fallback was called"
  },
  {
    "from": "0xD7...1B",
    "args":
    {
      "0": true,
      "1": "0x",
      "success": true,
      "data": "0x"
    }
  }
]
```

this shows the **call()** method can receive the output of this function that was called here, where we started with 123 and added 1, which then gave us 124 in the logs here, but in hexadecimal notation; this can be translated into an integer in the browser console by using **parseInt();** here we are telling Solidity to only send 5000 gas; this special function gets called in lieu of the fallback function, and note that there is no space after this comma, and instead of uint, you must declare the full name like this, but after that we do not declare the type of output; these next 2 parameters passed are the inputs being passed to the **foo()** function; which get outputted to the logs here and here; This function shows we don't have to send Ether when we use the **call()** method as it has no **payable** keyword in its definition; because the function does not exist, the fallback function is triggered and we get these outputs here and here with the message here in the logs; to see all of this in Remix live, deploy these two contracts then copy this address and past it here, then click here, if you do the same thing for this function here you will see these outputs in the logs from the fallback function; since this function that was called does not exist· NOTE: the code shown here uses the depricated syntax featured in this video whereas the code shown above (which is available here) uses the updated syntax shown here

NOTE: you can call the **foo()** function directly here and see the outputs in the logs

```
(bool success, bytes memory data) = _addr.call{value: msg.value, gas: 5000}
```

new

```
(bool success, bytes memory data) = _addr.call.value(msg.value).gas(5000)
```

old(depricated)

YouTube Tutorial References:

Learn Solidity (0.5) by Examples
Smart Contract Programmer

github.com/Richard-Burd/solidity-sandbox
last updated @5:38pm on 16/August/2021 by Richard Burd rick.a.burd@gmail.com

Solidity Illustrated