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

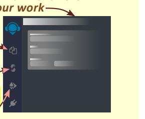
variables declared inside a function are only available during a call of that function, and such variables are not stored on the blockchain

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

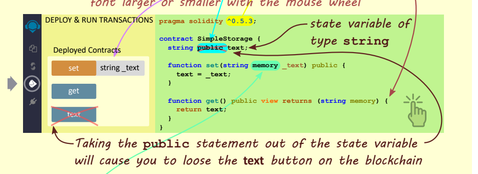
STEP 1: Write a smart contract in this file explorer

STEP 2: Compile the smart contract with compiler version compatible 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

view

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

returns

pure

Two ways to get state variables
1.) write your own function
2.) let Solidity compiler write it

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

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;
  }
}
```

This example shows a 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!

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

Ether and Wei

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

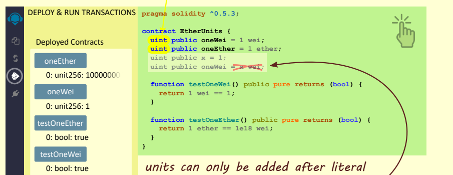
The currency used within Ethereum is Ether; it can be used to:
- pay block reward
- pay transaction fee
- transfer between accounts

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;
  uint public x = 1;
  uint public oneWei = ~~x wei~~;

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

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

giga-wei

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

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 prince

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 is defined in the ethereum white paper

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 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 the whitepaper; here's what would happen:

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

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

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

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

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

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

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

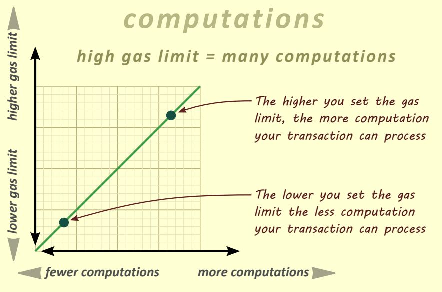
after forth iteration
(- 1,800 gas, 0 gas left)

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

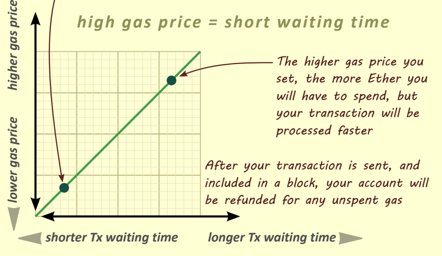


The higher you set the gas limit, the more computation your transaction can process

The lower you set the gas limit the less computation your transaction can process

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



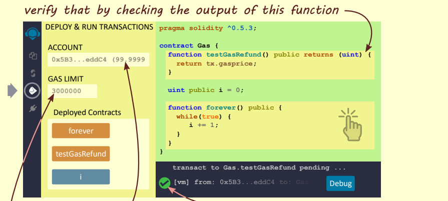
The higher gas price you set, the more Ether you will have to spend, but your transaction will be processed faster

After your transaction is sent, and included in a block, your account will be refunded for any unspent gas

time

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



ACCOUNT 0x5B3...eddC4 (99.9999
GAS LIMIT 3000000

```
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;
    }
  }
}
```

```
transact to Gas.testGasRefund pending ...
[vm] from: 0x5B3...eddC4 to: Gas
```

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 reccomended (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[9][9] arr2DFixed;
  uint arr2D;

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

  function multiDimFixedSizeArrayOutput() public view returns (uint[9][9] 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

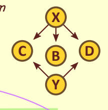
In Solidity, constructors are always executed in a certian order. You cannot override a state variable that is declared in a parent contract by simply redeclaring it in a child contract

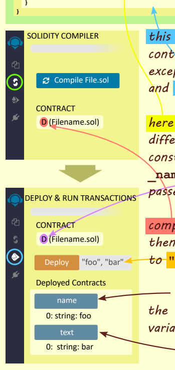
```
pragma solidity ^0.5.3;

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 it 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)

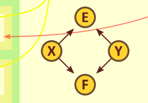
A video discussing all of this material is available here
this section starts at time: 3:26 / 8:09

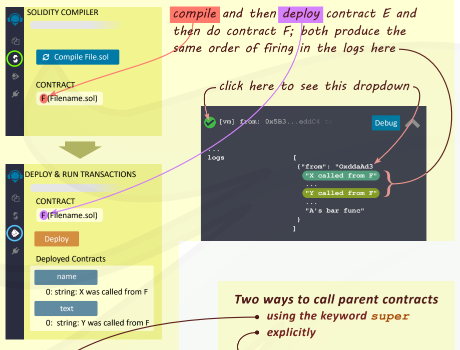
Below we see in what order parent constructors will be called and executed; whereas on the right (Constructor-1) we detail how to pass input variables in a child constructor to a parent constructor

```
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

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





```
[vm] from: 0x5B3...eddC4 to:
...
logs         [
              {"from": "OxddaAd3
                "X called from F"
                ...
                "Y called from F"
                ...
                "A's bar func"
              }
             ]
```

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

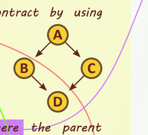
Two ways to call parent contracts
- using the keyword super
- explicitly

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 X, Y, Z {
  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

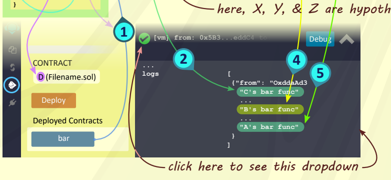


super

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 overidden by the bar() functions in contracts B and C

emit

here, X, Y, & Z are hypothetical



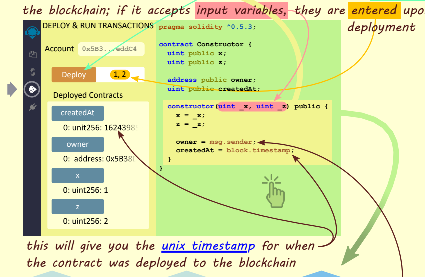
```
[vm] from: 0x5B3...eddC4
...
logs         [
              {"from": "OxddaAd3
                "C's bar func"
                ...
                "B's bar func"
                ...
                "A's bar func"
              }
             ]
```

click here to see this dropdown

Constructor

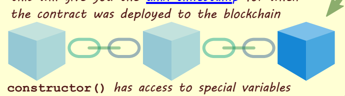
A video discussing all of this material is available here:
https://youtu.be/HpjZZ9tASGs

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 z;

  address public owner;
  uint public createdAt;

  constructor(uint _x, uint _z) public {
    x = _x;
    z = _z;

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

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



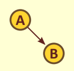
constructor() has access to special variables like msg.sender & block.timestamp

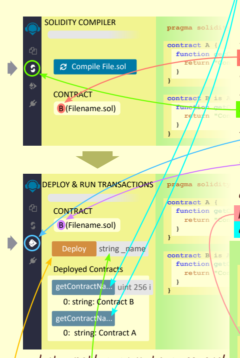
Inheritance (Override-1)

A video discussing all of this material is available here:
https://youtu.be/GnztHR8-1o8

contract B is inheriting from contract A, and they both have a function with the same name; normally, the function in contract B would overide 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

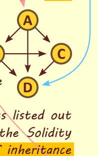
```
pragma solidity ^0.5.3;

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 derrived = more inheritance; this gives you this and vice-versa

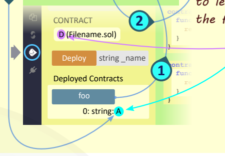


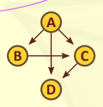
In the example above, contract A and contract B are both equally baselike, so contract C inherits whichever one is listed last in the order; if this order has contracts listed out of order (from most base-like to most derived) then the Solidity compiler might throw an error, otherwise the order of inheritance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

```
pragma solidity ^0.5.3;

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

contract B is A {

  // foo is not defined here

}

contract C is A, B {
  // foo is not defined here
}

contract D is A, C {
  // foo is not defined here
}
```

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)

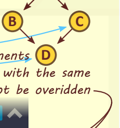
A video discussing all of this material is available here
this section starts at time: 6:56 / 12:37

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

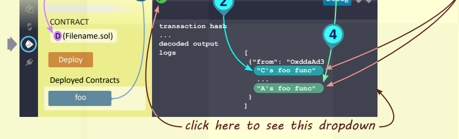
```
pragma solidity ^0.5.3;

contract A {
  event MyLogEvent(string message);

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

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

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

contract D is B, C {

}
```

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



```
[vm] from: 0x5B3...eddC4
transaction hash
...
decoded output
logs         [
              {"from": "OxddaAd3
                "C's foo func"
                ...
                "A's foo func"
              }
             ]
```

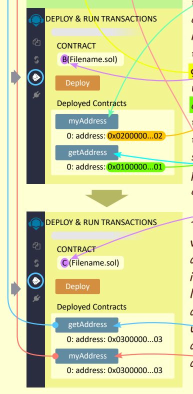
click here to see this dropdown

Inheritance (Override-2)

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

```
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;
  }
}
```

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


DEPLOY & RUN TRANSACTIONS
CONTRACT
B (Filename.sol)
Deploy
Deployed Contracts
myAddress
0: address: 0x0200000...02
getAddress
0: address: 0x0100000...01
DEPLOY & RUN TRANSACTIONS
CONTRACT
C (Filename.sol)
Deploy
Deployed Contracts
getAddress
0: address: 0x0300000...03
myAddress
0: address: 0x0300000...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

YouTube Tutorial References:

Learn Solidity (0.5) by Examples
Smart Contract Programmer

github.com/Richard-Burd/solidity-sandbox
last updated @ 5:21pm on 21/July/2021 by Richard Burd rick.a.burd@gmail.com

Solidity Illustrated