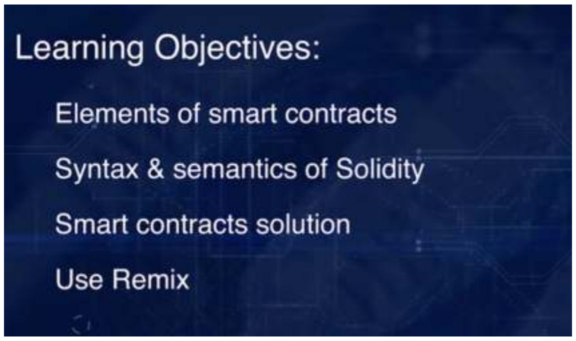
Smart Contracts
Course 2 of the specialization

Contents

Chapter 1

1.1 Smart Contract Basics: Why Smart Contracts?

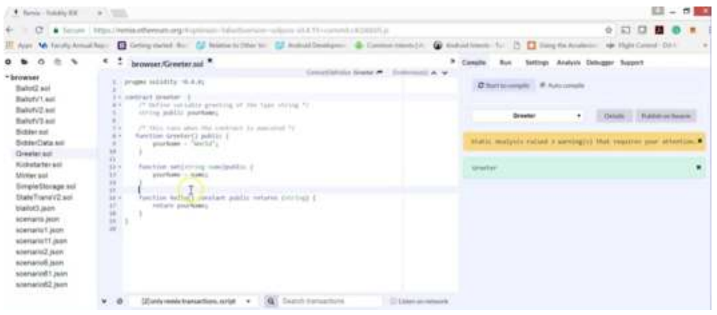
Welcome to the second course in the blockchain specialization titled Smart Contract. This course is completely dedicated to smart contract, the computational element of the blockchain technology. Let's begin with the origins of a smart contract. In the first course of the specialization, blockchain basics, we discussed how proven algorithms and techniques for encryption, hashing and peer to peer networks have been creatively applied to the innovation of blockchain, a decentralized, trusted, distributed, immutable ledger. The concept of the smart contract was there well before the advent of the Bitcoin. Computer scientist Nick Szabo detailed his idea of cryptocurrency Bit gold as a sort of a precursor for Bitcoin. He also outlined the concept of smart contract in his 1996 publication. In fact, Szabo coined the term smart contract more than 20 years ago. Smart contract is a centerpiece and main thrust of Ethereum blockchain. It is the good, bad and the ugly of the blockchain technology. It's a powerful feature. Improper design and coding of a smart contract, resulted in significant failures such as DAO hack and Parity wallet lockup. It is such a dominant feature of blockchain, which is why we've devoted the second course of this specialization to help you design, code, deploy and execute a smart contract. As we discussed in course one, Bitcoin has a script feature that includes rules and policies. Linux Foundation's Hyperledger blockchain has a smart contract feature called Chaincode. The Chaincode is written in go language, and executed in a docker environment. Docker is a lightweight container technology for executing programs. You can find more about these in the resource section. Many variations of smart contracts are prevalent in the blockchain context. We have chosen to discuss Ehereum implementation of smart contracts since Ethereum is a general mainstream blockchain, and it is being used as a reference blockchain for many others. At the end of this course, you'll be able



Learning Objectives:

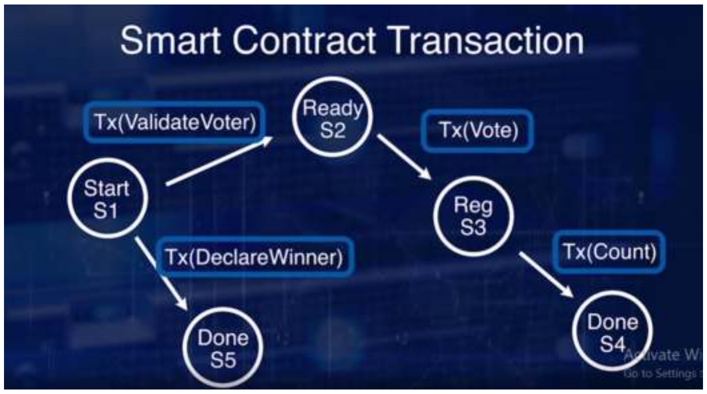
- Elements of smart contracts
- Syntax & semantics of Solidity
- Smart contracts solution
- Use Remix

to explain the elements of a smart contract, discuss the syntax and semantics of a smart contract programming language Solidity, solve a problem and design a smart contract solution, use remix development environment for building and testing smart contracts, and deploy the smart contract using remix, and invoke the contract from a simple web interface. In this course, it is imperative that you try the various concepts related to smart contract in a test environment in order to understand and apply these concepts. We'll use remix integrated development environment IDE, which is a web interface for hands-on explorations. At this time, please make sure you're able to access this interface at remix.ethereum.org. Be warned that remix is only a development environment, and it keeps changing as new features are being added, even as frequently as weekly. This course is one in which you will experiment with smart contract. Then, in the next course, we'll develop complete end-to-end applications using a smart contract. On completion of this module, you will be able to explain the elements of a smart contract, and explain the types of problems a smart contract can solve, define the structure of a smart contract, apply

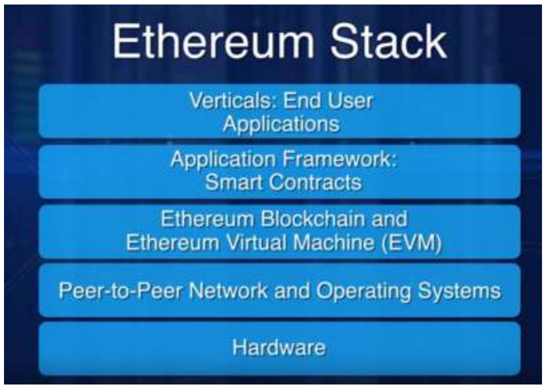


```
pragma solidity ^0.4.0;

contract Greeter {
    /* Define variable greeting of the type string */
    string public yourName;

    /* This runs when the contract is executed */
   function Greeter() public {
       yourName = "World";
    }

    function set(string name)public {
       yourName = name;
    }

    function hello() constant public returns (string) {
       return yourName;
    }
}
```

this knowledge to understand what the real smart contract written in Solidity language, use a web development environment remix to invoke and interact with a smart contract. As we begin, let's remember Bitcoin blockchain is primarily meant for transferring digital currency. Bitcoin added a simple conditional transfer of value through an embedded script. Recall from the blockchain basics course, course one, that this is a conditional feature that was bootstrapped as a softfork in Bitcoin. The script was limited in its capabilities. It enabled simple conditional transfers. After Bitcoin, evolved Ethereum. The founders of Ethereum developed smart contract keeping Nick Szabo's idea of a smart contract from over 20 years ago in mind. A significant contribution of Ethereum is a working smart contract layer that supports any arbitrary code execution over the blockchain. Smart contract allows for user-defined operations of arbitrary complexity. This feature enhances the capability of Ethereum blockchain to be a powerful decentralized computing system. Why would you want to transfer currency? Cryptocurrencies such as Bitcoin, enable transfer of values such as money or currency from peer-to-peer without any intermediaries. For what? To gift somebody, to buy a product, maybe even renew a driver's license, or send flowers to someone. Let us elaborate further. We may want the gift to be delivered on a certain date. Buy a product of a particular color and quality. We may need some credentials verified for renewing the license. And we may need a specific tulip bouquet to be delivered to Buffalo. This introduces conditions, rules, policies beyond that of which a simple money transfer cryptocurrency protocols can handle. Smart contract addresses this need for application specific validation for blockchain applications. Smart contract has some advantages including, a smart contract facilitates transaction for transfer of assets other than value or cryptocurrency. Smart contract allows specification of rules for an operation on the blockchain. It facilitates implementation of policies for transfer of assets in a decentralized network. It also adds programmability and intelligence to the blockchain. The smart contract represents a business logic layer, with the actual logic coded in a special high level language. A smart contract embeds function that can be invoked by messages that are like function calls. These messages and the input parameters for a single message are specified in a transaction. Let us compare Bitcoin transaction and a smart contract transaction. As you can see in Bitcoin, all the transactions are about send value. In the case of a blockchain that supports a smart contract, a transaction could embed a function implemented by a smart contract. Here we have a voting smart contract. The functions are ValidateVoter, Vote, Count, Declare Winner. Smart contract provides a layer of computation logic that can be executed on the blockchain, thus availing the features enabled by the blockchain framework. Recall the layers of a decentralized application that

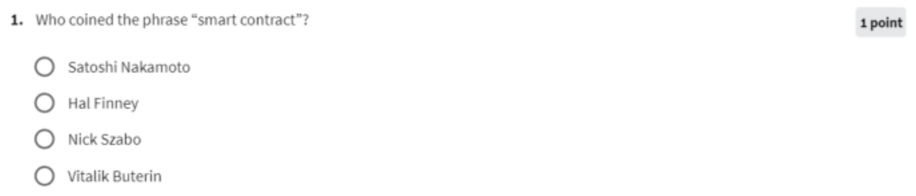


we discussed in course one, module two. Observe that smart contract providing the application framework for a domain application. For example, consider home mortgage application. Smart contract could embed all the business logic and the intelligence for the rules and regulation, to allow for automatic computation and initiation of operation.


Ethereum Stack
Verticals: End User Applications
Application Framework: Smart Contracts
Ethereum Blockchain and Ethereum Virtual Machine (EVM)
Peer-to-Peer Network and Operating Systems
Hardware

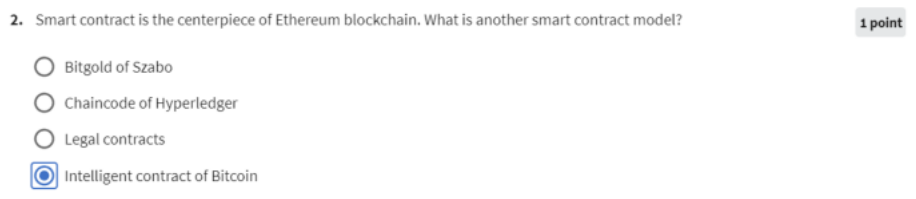
How might this be different from existing systems? You may ask. Here, all the operations are transparent and are recorded on the blockchain. Customers can directly access the tools without an intermediary like a bank. It is like the ATM for mortgage initiation. You are holding the assets, it's therefore an intermediary. Now that we have reviewed the advantages of a smart contract, let's look at what problem or problems a smart contract can solve. Typically, currency transfer is used to buy a service, a product, or a utility from a person or a business. There may be other conditions besides availability of funds while executing a transaction. For example, a business transaction may involve rules, policies, laws, regulations and governing contexts. Smart contract allows for these other real world constraints to be realized on a blockchain, thus a smart contract enables a wide variety of decentralized application of arbitrary complexity to be implemented on the blockchain. This can run the spectrum from supply chains to disaster recovery. It is likely many of the applications for the blockchain technology have not yet been conceived. It is predicted that online shopping will overtake retail shopping for the first time this holiday season, 2017-2018. Some couldn't have even imagined online shopping, mobile application or Uber 20 years ago. Smart contract is ushering the next generation blockchain that goes beyond the transfer of value into a visionary realm. Smart contract allows for implementation of rules, policies and with the help of blockchain, supports the methods for governance and provenance. Next in the upcoming lessons, we'll explore how to design and implement a smart contract.

1.1.1 Practice Quiz



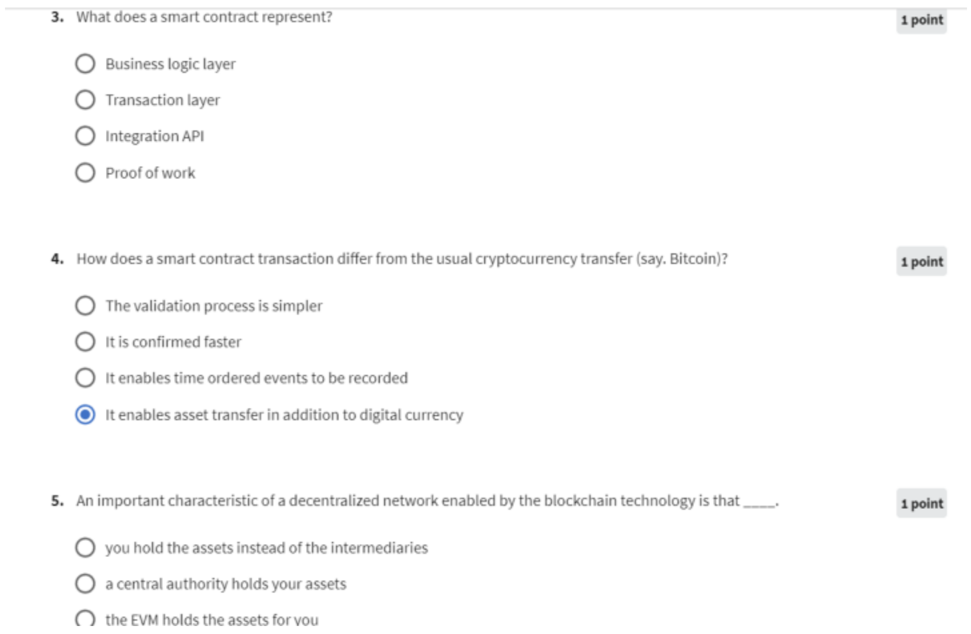
1. Who coined the phrase "smart contract"? 1 point

   - Satoshi Nakamoto
   - Hal Finney
   - Nick Szabo
   - Vitalik Buterin



2. Smart contract is the centerpiece of Ethereum blockchain. What is another smart contract model? 1 point

   - Bitgold of Szabo
   - Chaincode of Hyperledger
   - Legal contracts
   - [✓ Intelligent contract of Bitcoin]


3. What does a smart contract represent? 1 point

- ○ Business logic layer
- ○ Transaction layer
- ○ Integration API
- ○ Proof of work

4. How does a smart contract transaction differ from the usual cryptocurrency transfer (say. Bitcoin)? 1 point

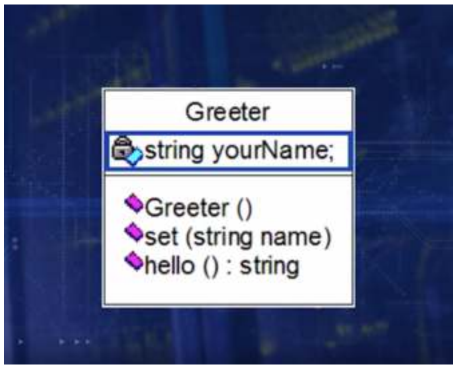
- ○ The validation process is simpler
- ○ It is confirmed faster
- ○ It enables time ordered events to be recorded
- [✓ It enables asset transfer in addition to digital currency]

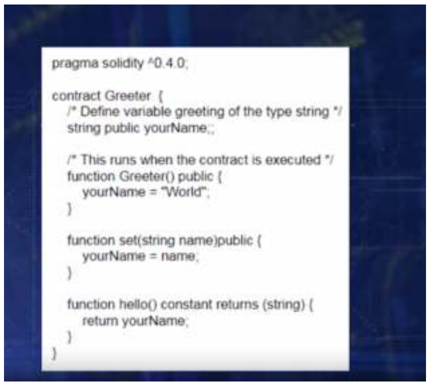
5. An important characteristic of a decentralized network enabled by the blockchain technology is that ____. 1 point

- ○ you hold the assets instead of the intermediaries
- ○ a central authority holds your assets
- ○ the EVM holds the assets for you

1.2 Smart Contracts Defined (Part 1) (Remix IDE and Greeter Demos)

For currency transfer, the verification and validation, we're just checking the existence and validity of UTXOs, the balances, and the structural characteristics. When a chain is thrown open for an arbitrary decentralized application requiring trust and immutable recording, conditions to be verified and validated become application-specific. At the completion of this lesson, you will be able to define the structure of a smart contract, apply this knowledge to understand with a real contract written in Solidity language, use a web development environment Remix to invoke and interact with a smart contract. Structural and meta-level attributes of a transaction are verified at the blockchain protocol level. How were the application-specific constraints? The answer is in the critical role played by the smart contract. Smart contract work with the application-specific semantics and constraints of the transaction and verify, validates, and executes them. Most of all, since it is deployed on the blockchain, the smart contract leverages the immutable recording and trust model of the blockchain. Since a smart contract is deployed in the blockchain, it is an immutable piece of code, and once deployed, it cannot be changed. We will have to redeploy the code as a new smart contract, or somehow redirect the calls from a old contract to the new one. Smart contract can store variables in it called state variables. We could retrieve how these variables change over the blocks. Contract in the Ethereum blockchain has pragma directive, name of the contract, data or the state variable that define the state of the contract, collection of function to carry out the intent of a smart contract. Other items, we'll discuss in the later lessons. Identifiers representing these elements are restricted to ASCII character set. Make sure you select meaningful identifiers and follow camel case convention in naming them. We'll learn these concepts using simple contracts written in high level language called Solidity. We will use a web integrated development environment, IDE, called Remix to create, deploy, execute, and explore the working of few representative smart contracts. Welcome to Remix IDE. This is available at Remix.ethereum.org. Here is the environment that is available as a web interface. On the left side, you see the file browser, where you can see all the smart contracts that you have created. You can create a new one, and it'll have an entry here. In the middle is the editor window, where you'll type in the smart contract. At the bottom is a console or the output window. On the right side, you have the tools compile, run, settings, analysis, debugger, and support. At the bottom will be the web interface, and ability to create a smart contract will be somewhere in this middle. With all these features, Remix is indeed a one-stop environment to develop, deploy, and test a smart contract. We will examine two very simple smart contracts, greeter and one integer storage, simple storage. These two examples are modified versions of the example given in Solidity documentation. Our goal is to get an overview of the structure of a smart contract without getting into the details of the Solidity language. However, we'll explain every item in the smart contract we plan to explore. Let's look at three steps in the development of a smart contract: design, code, and test. Here is the design of the Greeter contract. This is the Hello World of smart contract. Greeter has a string variable named yourName, the constructor Greeter, a set function to set the name, and a hello function that returns a string name so that you can use it to greet the world. Here, we see the code for the Greeter contract. It begins with the pragma that provides a version number so that the compiler knows which version of Solidity this was developed in. You also see the name of the contract Greeter, the state variable yourName. Note that it is in camel notation.





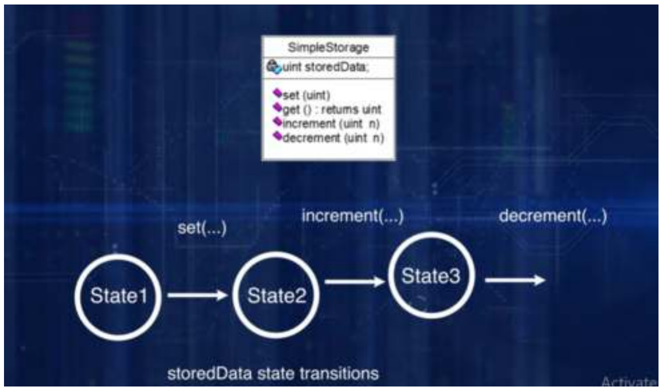
```
pragma solidity ^0.4.0;

contract Greeter {
    /* Define variable greeting of the type string */
    string public yourName;;

    /* This runs when the contract is executed */
    function Greeter() public {
        yourName = "World";
    }

    function set(string name)public {
        yourName = name;
    }

    function hello() constant returns (string) {
        return yourName;
    }
}
```

This is followed by functions, the constructor Greeter that initializes yourName variable, set function that sets a variable to a name supplied by the users message as a parameter, and a hello function that retrieves the name for use by the invoking application. Remix is where we test. Now, let's look at the Greeter contract. Here is a Greeter contract. At the first line, you see pragma Solidity. This provides the version of the Solidity. Then, it starts with the contract name, followed up by the data or the state variables, followed up by the functions. There are three functions here: Greeter, which is a constructor, and a set function which is setting the data variable, and then, a hello function which extracts the values of the state variable and returns it. The state variable here is yourName and it has a public visibility modifier, and the function Greeter, the constructor initializes yourName to World, and the function set initializes yourName to the name provided by the sender. Hello returns whatever value that was set in the state variable. Let's run it and see. I'm going to start by compiling it, and then, running it. When I run, I have to deploy the smart contract on a JavaScript VM. I'm going to change it to JavaScript VM, and I'm going to create it. When I create it, this is the web interface where you will see all the public variables. YourName is a public variable that's available there. Hello is a public function that is available there, and set is another public function that is available there. When I just simply say yourName, the current yourName given is World that shows up. I'm going to set the name to Buffalo. If I set the name to Buffalo and I click on yourName, yourName shows up at Buffalo. If I click on the function hello, hello will return the current yourName that happens to be Buffalo, and it will show up.

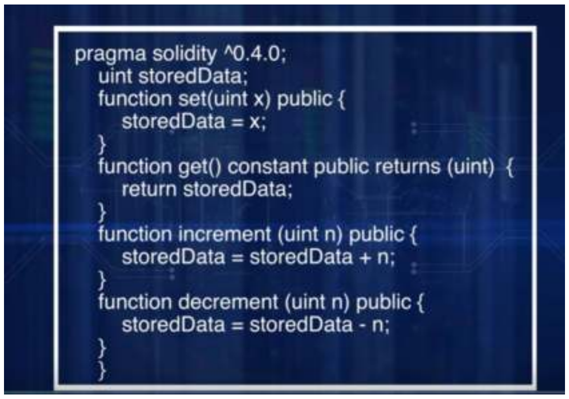
1.3 Smart Contracts Defined (Part 2) (Simple Storage Demo)

Now onto the second example. Here is the design representation of simple storage. Observe that it looks exactly like a regular class definition in a unified modeling language class diagram. There are a few syntactic differences.

Also, note that the state transition that result from execution of a smart contract functions. Recall that the state and the state hash in a black header, we discussed in course one. These are those states. Initial state



is state one, is updated by the message set and the state changes to state two. Next, the execution of increment message results in state three and the decrement message or function execution will transition to the next state and so on. Now let's look at the code. Open up remix environment and enter the solidity program shown here.

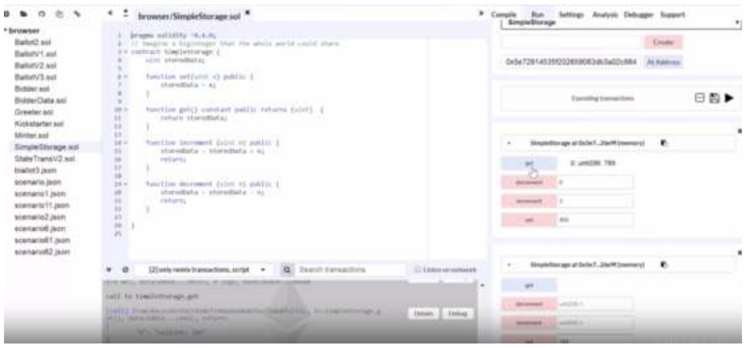


```
pragma solidity ^0.4.0;
    uint storedData;
    function set(uint x) public {
        storedData = x;
    }
    function get() constant public returns (uint) {
        return storedData;
    }
    function increment (uint n) public {
        storedData = storedData + n;
    }
    function decrement (uint n) public {
        storedData = storedData - n;
    }
    }
```

Please understand that it is a minimal environment that provides simple commands, we're adding files, compiling, deploying them for testing. About this particular smart contract, Imagine a big number that a whole world could share. A simple de-centralized use case is world population. Count to something that is distributed all over the planet. This particular smart contract has get, set, increment and decrement as functions. You may expand these to more sophisticated operations such as, dig the soil and evaluate the acidity, identification of an isolated Ebola patient, Work with a remote supply of rare mineral, management of land deeds and any number of decentralized application.

Now let's test.

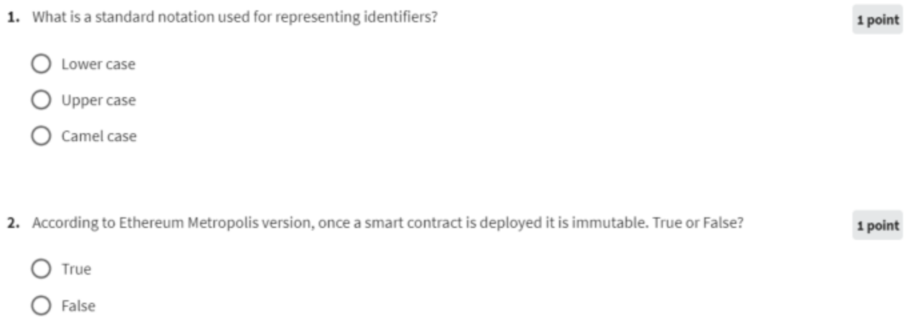
Next I'm going to close the greeter sol and let's move on to the Simple Storage. Simple Storage. This is another smart contract. Again, it starts up with pragma solidity and the contract name. There is only one single state variable store data. There are several function set, get, increment, and decrement. You notice that there is no explicit constructor but a constructor is automatically by default created for the smart contract. I'm going to compile it, and I'm going to run it and by doing that I can close any of the previous smart contracts that I've created here and I'm going to go here and create the Simple Storage and all the public variables and public methods are displayed in the web interface. You can see that set, get, increment, and decrement are available. Since stored data was not a public data, it's not available in this public web interface. And now, if I say Get, there is no value set so you'd return to zero. And let's set it to a value 456. I'm going to set it and if I get it, you can see 456 returned. I'm going to increment it and if I increment it by some value, let's say three, and increment it and if I get the value, I get 459. Let say we decrement it by eight and decrement it i should get 451. Yes I got the 451. That is the Simple Storage. Understand that I can create this simple storage, Smart contract again by using the smart contract address. I'm going to do that. I'm going to copy the Smart Contract address and put it in the local contract address. Paste. Then at that address, I'm creating one more smart contract. Here I have two smart contracts and they have the same address and I can- whatever I do in this smart contract will be available throughout, globally, everywhere you would see that. So let me set this to some 789 and I set it and I go back to the smart contract here and I get it, I should get 789. That's how the smart contracts all over the world are in consistent state. You can see that in the simple global storage that we have here. In summary, we examined two



```
pragma solidity ^0.4.0;
// Imagine a big integer that the whole world could share
contract SimpleStorage {
    uint storedData;

    function set(uint x) public {
        storedData = x;
    }

    function get() constant public returns (uint) {
        return storedData;
    }

    function increment (uint n) public {
        storedData = storedData + n;
        return;
    }

    function decrement (uint n) public {
        storedData = storedData - n;
        return;
    }
}
```

simple smart contracts and learn the high-level structure of a smart contract. We also got an introduction to the remix integrated development environment for working with smart contracts. In the next module, we'll learn the solidity language itself.

1.3.1 Practice Quiz



1. What is a standard notation used for representing identifiers? 1 point

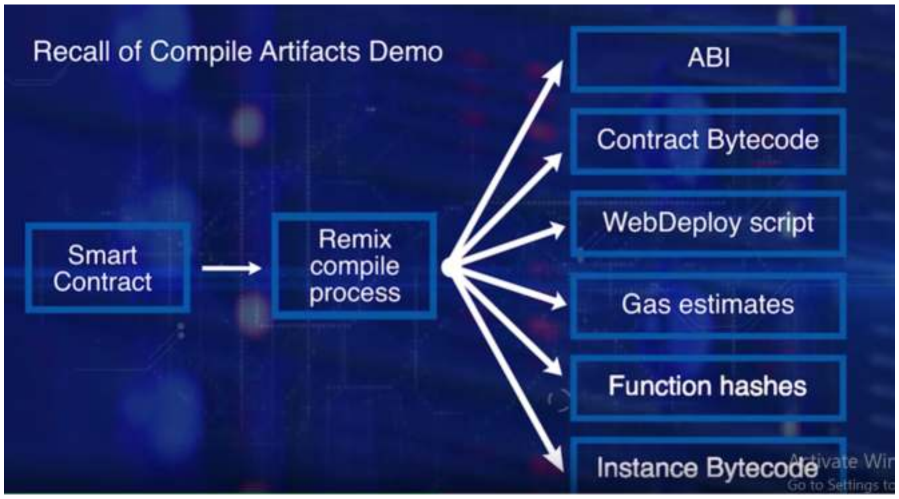
   - Lower case
   - Upper case
   - Camel case

2. According to Ethereum Metropolis version, once a smart contract is deployed it is immutable. True or False? 1 point

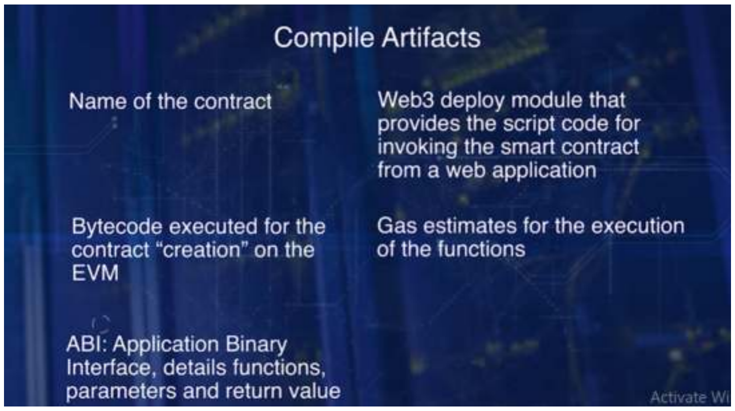
   - True
   - False

1.4 Processing Smart Contracts (Compile Artifacts Demo)

On completion of this lesson, you will be able to explain the address of a smart contract, list the compilation artifacts generated by the compiler, byte code, ABI, web3deploy script, function hashes, and gas estimates. A smart contract can be created, on behalf of an externally owned account, by application programmatically from the command-line interface and by a script of commands from high level applications and user interface or UI. It can also be created from inside a smart contract. We'll cover this approach in a later course. We need an address for the smart contract to deploy it and invoke its functions. The address is computed by hashing the account number of externally owned account UI and the nonce. Let us review these steps with the Greeter smart contract on Remix. Let's go back to the Greeter.sol and find out how the compiler provides you many artifacts. I'm going to compile, and then click on the Details button, and you can see there is the name of the contract, the metadata of the contract, the byte code of the smart contract that executes on the Ethereum virtual machine. The Application Binary Interface is here, and if we can expand on it, you can see the various details. Web3deploy script is here. This is used by the web application to interface to the smart contract and the metadata hash. There are few other details. The function hashes, the hashes by which the functions are called, and the gas estimates when the functions are executed, and the runtime bytecode, and then the assembly code. As you can see, a contract needs to be compiled. Various artifacts are generated including; the bytecode for deploying the contract, and the Application Binary Interface, ABI for the application that smart contract interact with a deployed bytecode. Remix compiler script generates many artifacts in one shot. You can view these items generated by clicking on the Compile button, and then the Details button just below the Compile button. A pop-up screen appears with many details. Here are some of the artifacts generated by the Remix smart contract compile process and their use.



ABI, Application Binary Interface, the interface schema for a transaction to invoke functions on the smart contract instance bytecode. Contract bytecode, this is the bytecode that is executed for instantiating a smart contract on the EVM. Think of it like executing a constructor of a smart contract to create an object. WebDeploy script, this as two items; json script to web application to invoke smart contract function, script for programmatically deploying a smart contract from a web application. Gas estimates, this provides a gas estimates for deploying the smart contract and for the function invocation. Function hashes, first four byte of the function signatures to facilitate function invocation by a transaction. Instance bytecode, the bytecode of the smart contract instance. We will learn more about these and how to use them in building blockchain application in the upcoming lessons. Summarizing, Remix solidity compiler generate several artifacts as discussed. Here are some important ones: name of the contract, bytecode executed for the contract creation on the EVM, ABI, Application Binary Interface, details functions, parameters and return values, Web3 deploy module that provide the script for invoking the smart contract from web application, gas estimates for execution of a function, actual run-time bytecode of the smart contract. Explore this list on Remix ID.



Compile Artifacts

Name of the contract

Bytecode executed for the contract "creation" on the EVM

ABI: Application Binary Interface, details functions, parameters and return value

Web3 deploy module that provides the script code for invoking the smart contract from a web application

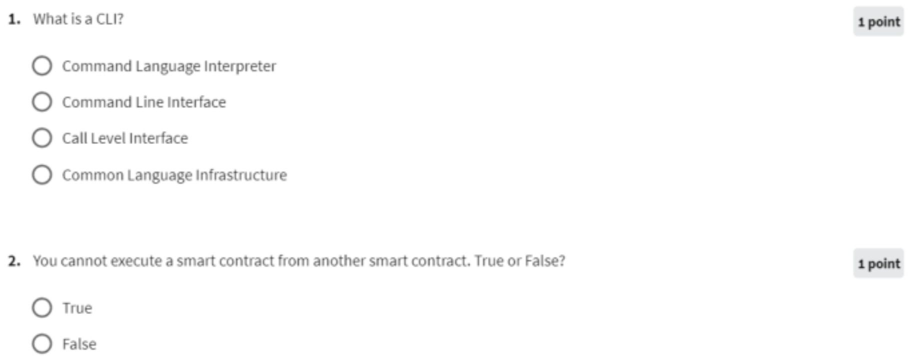
Gas estimates for the execution of the functions

If you have Remix window open and the smart contract in it, click the Compile, and then, the Details button.

1.5 Practice Quiz

1.6 Deploying Smart Contracts

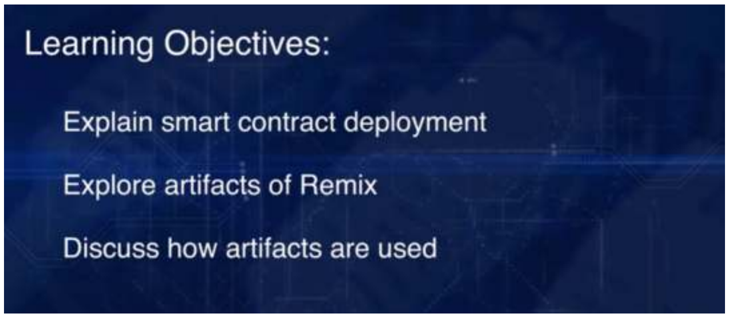
Model one lesson four. Deployment and execution of a smart contract. On completion of this lesson, you will be able to explain the smart contract deployment process, explore the artifacts generated by Remix compile process, discuss how these artifacts are used to deploy and interact with the smart contract. Let us start by getting into the smart contract deployment process. First, a smart contract solution is written in high-level language and


1. What is a CLI? 1 point

- ○ Command Language Interpreter
- ○ Command Line Interface
- ○ Call Level Interface
- ○ Common Language Infrastructure

2. You cannot execute a smart contract from another smart contract. True or False? 1 point

- ○ True
- ○ False

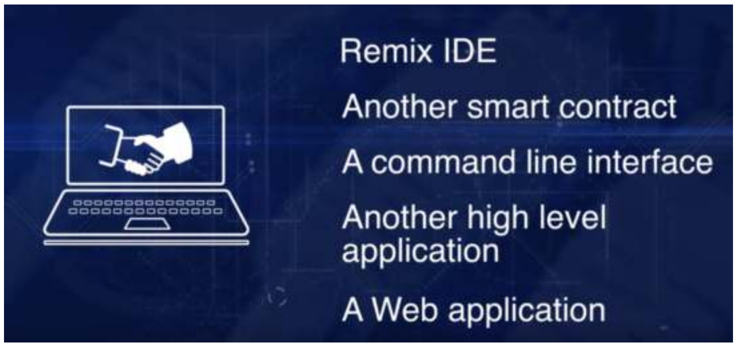

Learning Objectives:

Explain smart contract deployment

Explore artifacts of Remix

Discuss how artifacts are used

compiled bytecode. An ABI is also generated for high-level language application. Example, Web Apps to interact with the binary smart contract. Recall that we discussed about etherium virtual machines in the last course. EVM provides execution environment for a smart contract bytecode. The smart contract requires an address for itself so that transaction can target it for invocation of its function. The contract address is generated by hashing the sender's account address and its nonce. A unique target account is reserved for smart contract creation and deployment. Target account zero. If a target's address is zero or null, it is meant for creating a new smart contract using its payload feed. The payload of a transaction contains the bytecode for the smart contract. This code is executed as a part of the transaction execution to instantiate the bytecode for the actual smart contract. Similar to how a constructor creates an object, the execution of a smart contract creation transaction results in the deployment of this smart contract code on the EVM. It is permanently stored in the EVM for future invocation. This transaction goes through all the regular verification and validation specified in the etherium blockchain protocol. Block creation, transaction confirmation by the full nodes deploys a the same contract on all the nodes. This provides consistent execution when the regular transaction with function messages are invoked on the smart contract. We have now explained the fundamental process of deployment of the smart contract but there are many other approaches for deploying this smart contract. They can be deployed from Remix IDE, another smart contract, a command line interface, another high-level language application or Web application. We're going to


Remix IDE

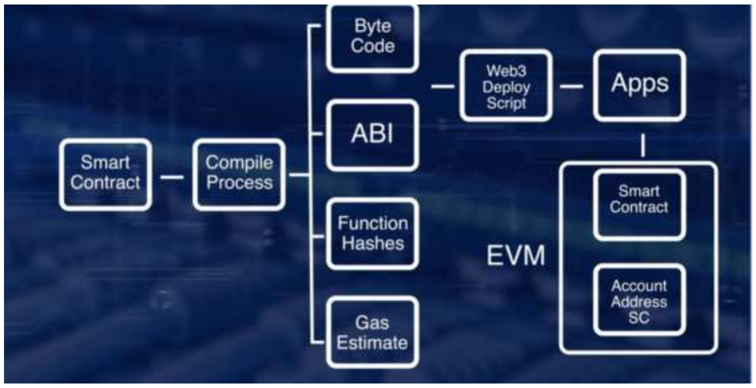
Another smart contract

A command line interface

Another high level application

A Web application

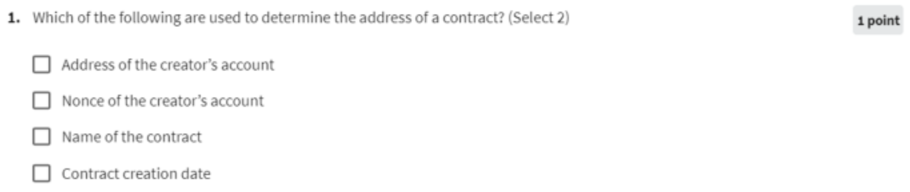
review the deployment using only the Remix IDE. Here is the complete process. You enter the smart contract code in the Remix IDE and compile. Remix generates several artifacts as discussed earlier and as shown in



the picture. For the ease of deployment, Remix provides us with the Web3 deployment script which contains a bytecode. Application Binary Interface, ABI, and account detail. To deploy the smart contract, we could just execute the script. Once the deployment is done, the address is generated by hashing creator's account number and nonce. To interact with the smart contract, we'll use the smart contract address, ABI definition, and the function hashes. Summarizing. We learned the purpose of the smart contract and its critical role in transforming blockchain technology for enabling decentralize systems. We explored the structure and basic concepts of a smart contract through examples.

We illustrated Remix, Web ID, for deploying and interacting with the smart contract. How do we design and program a smart contract? What are the elements of the solidity language so that we can drive the smart contract for our applications. We do that in the next module. Onto the next module, solidity language for smart contracts.

1.6.1 Practice Quiz



1. Which of the following are used to determine the address of a contract? (Select 2) **1 point**

- ☐ Address of the creator's account
- ☐ Nonce of the creator's account
- ☐ Name of the contract
- ☐ Contract creation date

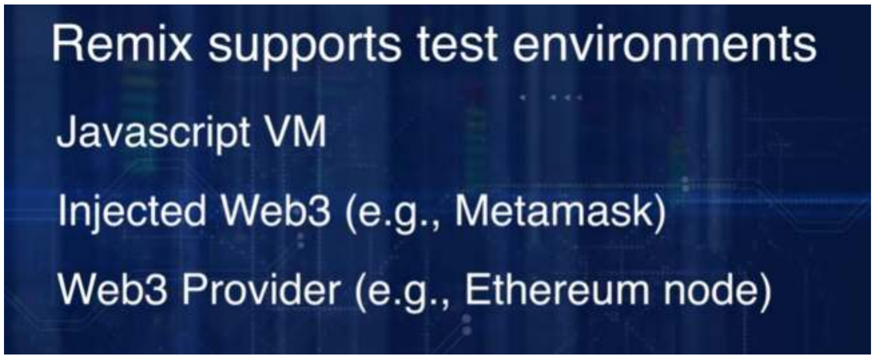
Chapter 2

2.1 Solidity: Structure

Welcome to the second module, Solidity, the second course in the blockchain specialization. We will use Solidity as a high level language for implementing a smart contract. Solidity is a high level language that is a combination of JavaScript, Java, and C++. It is specially designed to write smart contracts and to target the Ethereum Virtual Machine.

Recall from our course one that the format of a smart contract is like the class definition in the object-oriented programming style.

In this course, we learn the important features of Solidity and the development of a smart contract through two specific problems. To create and view the smart contracts, we'll use Remix Web IDE for Solidity. There are many IDEs available. We chosen to use Remix, which provides not only a compiler, but also a runtime to test a smart contract you will design. Most of all, it works without any software insulation on your computer. Remix supports free runtime test environments, JavaScript VM, Injected Web3, like Metamask, and Web3 Provider, for example, your locally running Ethereum node. We'll use only JavaScript environment for this module.



Remix supports test environments

Javascript VM

Injected Web3 (e.g., Metamask)

Web3 Provider (e.g., Ethereum node)

Our goal is to master the Solidity basics in this course. In the next module, we'll explore our approaches to deploying and invoking a smart contract on a test blockchain based on Injected Web3 and Web3 Provider. [MUSIC] Recall that we examined the structure of a simple smart contract in the last lesson.

Here is a more detailed structure of a smart contract. After this lesson, you will be able to discuss the elements of Solidity, a high level language for writing smart contract. Illustrate data types, and data structures, and functions, enum, modifiers, and events using Solidity code. Apply these concepts to design, develop, deploy, and test a smart contract.

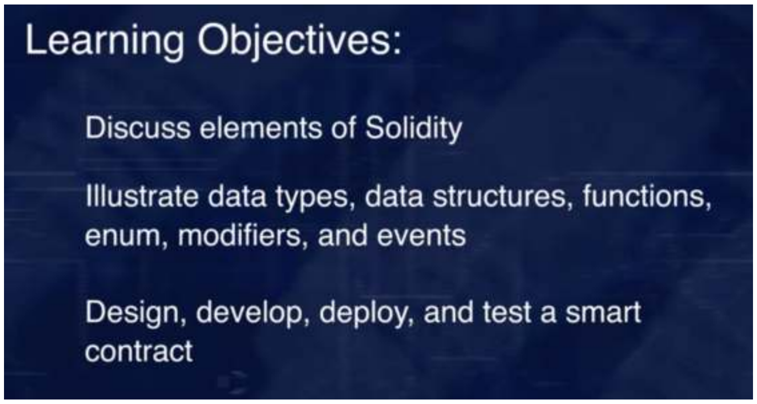
Here is am ore detailed structure of a smart contract.

Data or state variables, functions, there are several types of functions allowed. Constructor, default or user-specified, only one, meaning it cannot be overloaded. Fallback function, there's a powerful feature of an anonymous function

that we'll discuss in later courses.

View functions, pure functions, no state change, it computes under terms of value, example math functions. Public functions, accessible from outside, two transactions, state changes recorded on the blockchain. Private function, accessible only with the current contract.

Internal function, accessible inside the current contract and inherited contracts. External functions can be accessed only from outside the smart contract. User defined types in struct and enums. Modifiers and, finally,

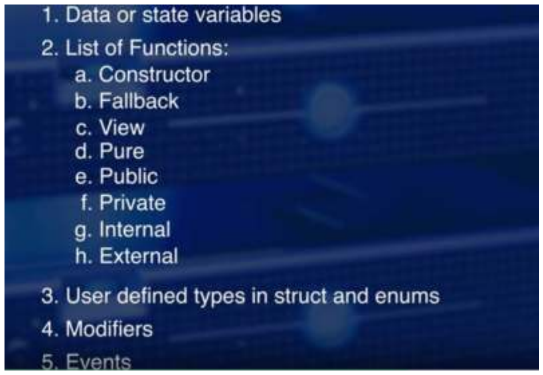

Learning Objectives:

Discuss elements of Solidity

Illustrate data types, data structures, functions, enum, modifiers, and events

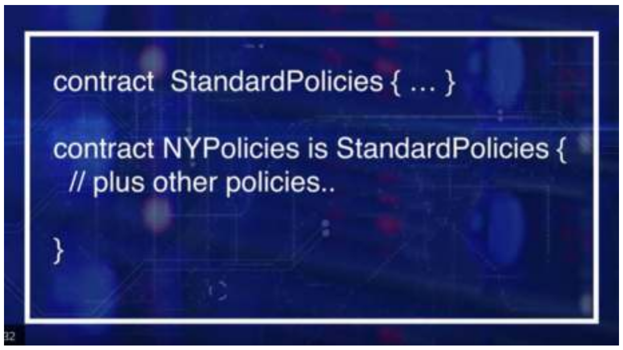
Design, develop, deploy, and test a smart contract

events. Besides its explicit content, a smart contract can also inherit from other smart contract, as shown in the


1. Data or state variables
2. List of Functions:
   a. Constructor
   b. Fallback
   c. View
   d. Pure
   e. Public
   f. Private
   g. Internal
   h. External
3. User defined types in struct and enums
4. Modifiers
5. Events

following example.

Here standard policies contract defines a basics policies that apply to all states, which is inherited by NYPolicies, smart contract, where more policies can be added. Now, we step into the smart contract and look at


```
contract  StandardPolicies { ... }

contract NYPolicies is StandardPolicies {
  // plus other policies..

}
```

the function definition.

Function definitions are similar to functions in any other high level language. Function header, followed by the code, within curly brackets. Function code contains the local data and statements to process the data and return the results of processing. Function header can be as simple as an anonymous noname function to a complex function header loaded with a lot of details. Here is more explanation of each of the items of the function header.

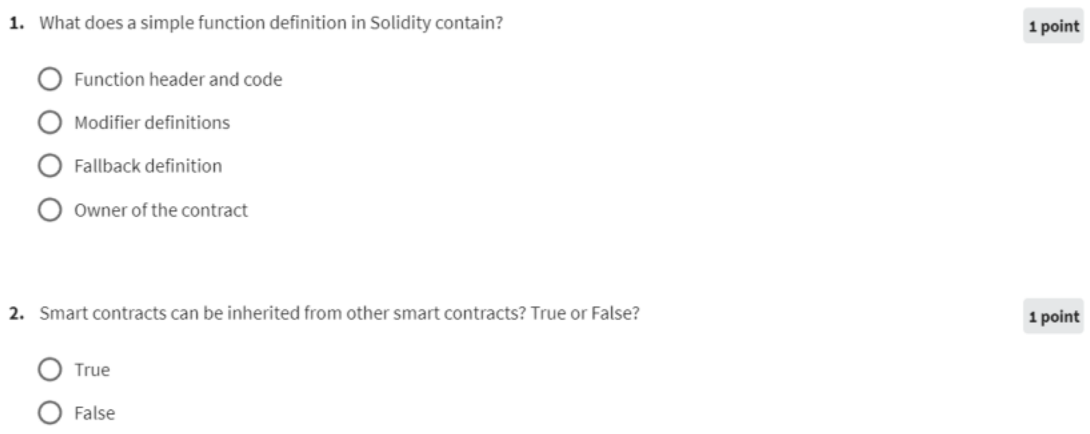
Function is a keyword at the beginning of all functions. Parameters are any number of pairs type identifier, example, UInt count. returnParameters, return values can be specified as pair type identifier or just type. When only type is specified, it has to be explicitly returned using a return statement.

If type and identifier are specified in the return statement, any statechain that happens to the identifier within the function is automatically returned.

Any number of values can be returned, unlike common programming languages that allow only one return value. For example, multiple variables, age and gender, can be assigned return values from a function getAgeGender.

In summary, a smart contract for Ethereum can be specified using Solidity defined structure and functions. A variety of function types are provided for expressing smart contract operations. A smart contract in Solidity can inherit its attribute and functionality from another smart contract.

2.1.1 Practice Quiz



1. What does a simple function definition in Solidity contain? 1 point

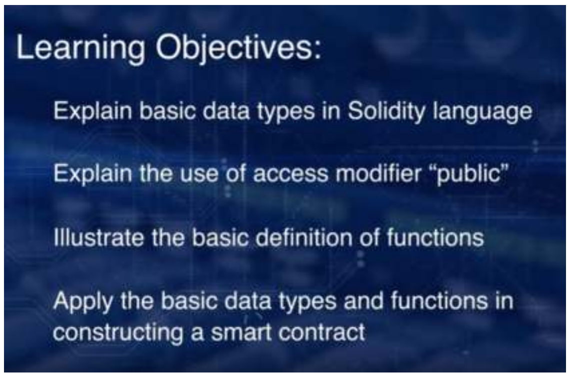
- ○ Function header and code
- ○ Modifier definitions
- ○ Fallback definition
- ○ Owner of the contract

2. Smart contracts can be inherited from other smart contracts? True or False? 1 point

- ○ True
- ○ False

2.2 Basic Data Types Statements (Bidder Data Functions Demos)

On completion of this lesson, you will be able to explain the basic data types and Solidity language, explain the use of visibility modifier public, illustrate the basic definition of functions, and apply the basic data types and functions in constructing a smart contract.

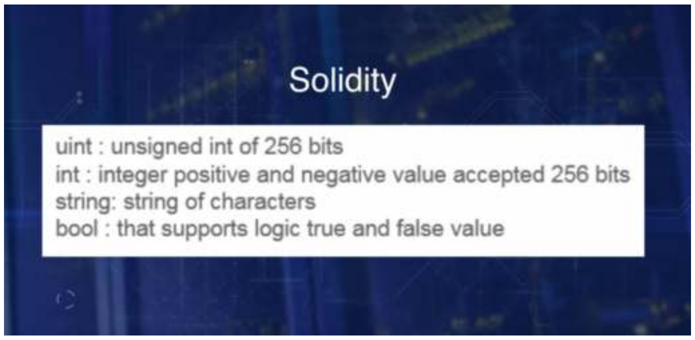


Learning Objectives:

- Explain basic data types in Solidity language
- Explain the use of access modifier "public"
- Illustrate the basic definition of functions
- Apply the basic data types and functions in constructing a smart contract

Recall the concept that we discussed in course one. Externally owned account, contract account, transactions, and blocks. We use all of these in this lesson. We'll also use gas or crypto fuel that is paid for transaction execution, and an operation step as set by Ethereum Protocol. Remember, one ether is 10 power 18 Wei. Since operations consume gas, choosing appropriate and efficient data structures are important steps in constructing your smart contract. Solidity supports many of the basic types of a high-level language.

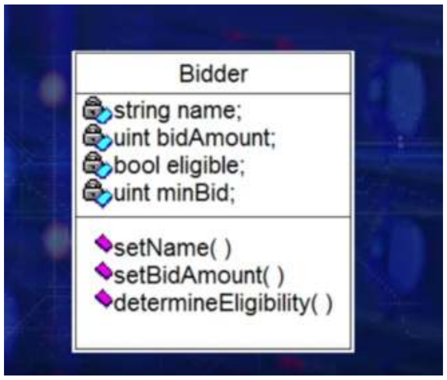
We have given just a few. Default modifier is private. You explicitly state the public modifier, if that is what is intended. For every data declared public, accessor or getter function is automatically provided. Let us examine these concepts with a simple bidder example. The common statements available in any high-level language are available in solidity with very little variation. Assignment statement, if-else, why, for, etcetera. We learn them as we add code elements into the examples. We will develop bidder smart contract in incremental steps starting with the basic design representations shown here. Always design before you code. Let's move to



Solidity

uint : unsigned int of 256 bits
int : integer positive and negative value accepted 256 bits
string: string of characters
bool : that supports logic true and false value

the remix environment to work on this example. The bidder smart contract design is shown as a class diagram. It has three items; name of the contract, the data or states, the functions. The first version of the bidder contract, we add the data. The next thing that we want to do is look at the bidder data, and only the data. We're going



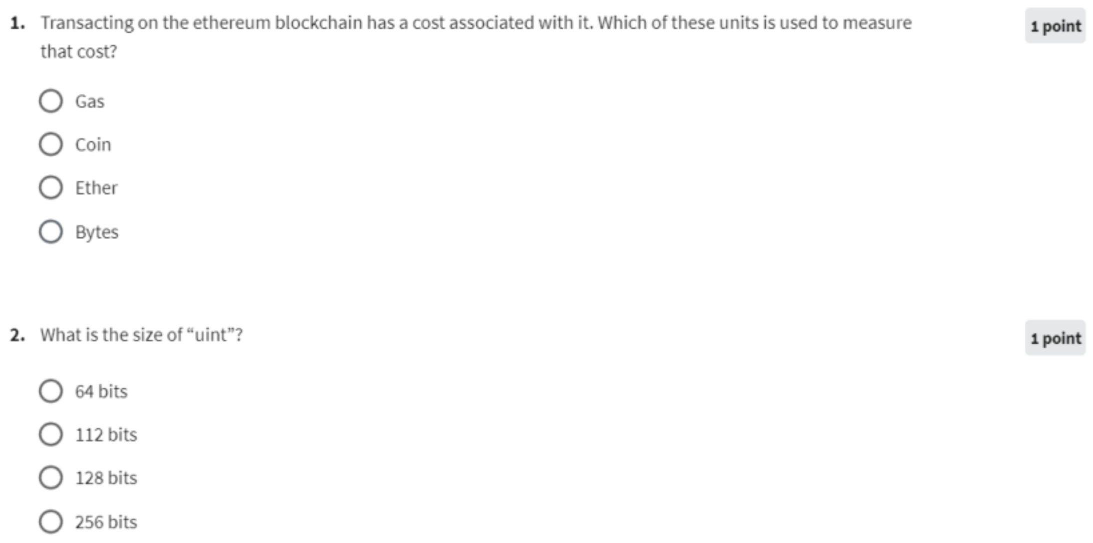
to do some incremental development here. So, I have here the bidder contract and I have only the data, and you see that many of these state variables or data are public, and so, we should be able to access them from the web interface even without any functions. So let's compile and run them and create or deploy the smart contract, and I'm going to do that. I'm going to say create and you can see that it was created by this account number and the smart contract account number is here, and it is different from the account of the creator. The public variables are available here. You can see when you click on the bid amount, and you can see the 20,000 that was the initial value for the bid amount. Nothing else is eligible, it's initialized to false, name, we don't know the name. So, let's add the functions to this basic dataset of the state variables and complete the bidder contract.

Now that we have added the data, let's add the functions listed in the class diagram. We'll implement the two setters and the one that determined eligibility to bid. The code for this demo is in the resource section.

I'm going to go to the bidder sol that I pre-created for this. I added the functions here. The function setName that sets the name of the bidder and bid amount sets the bid amount. Determine the eligibility is a function that based on the state variable determines whether this bit is eligible or not. Let's go through and run it. I'm compiling it, running it, and this time I'm going to close the previous smart contract. I'm going to deploy one and new one and by clicking on the create button. You can see lot more functions here because we have added the public functions were set and determine eligibility. I'm going to chain the name of the set name to let's say "Amherst". So, I'm going to change the name and I'm going to obtain the bid amount to 330,000 and set amount setName, and let's look at and see whether these have been stored in the state variable. I'm clicking on the name. Buffalo has been changed to Amherst and I'm clicking on the bid amount and you can see that it has been changed to 30,000. We want to find out if this bid amount is eligible, and I'm clicking on the function "Determine eligibility," and that sets the variable eligible which is a public variable. Automatically, it has been given a getter function. I click on it, I get true. Yes, 30,000 is indeed higher than 1,000 and we got a true value. Let change the bid amount now to 300. Set the bid amount to 300, and determine the eligibility and then click, look at the bid amount it's 300. Look at eligible. It says false, and so, that illustrates the simple smart contract of the bidder.

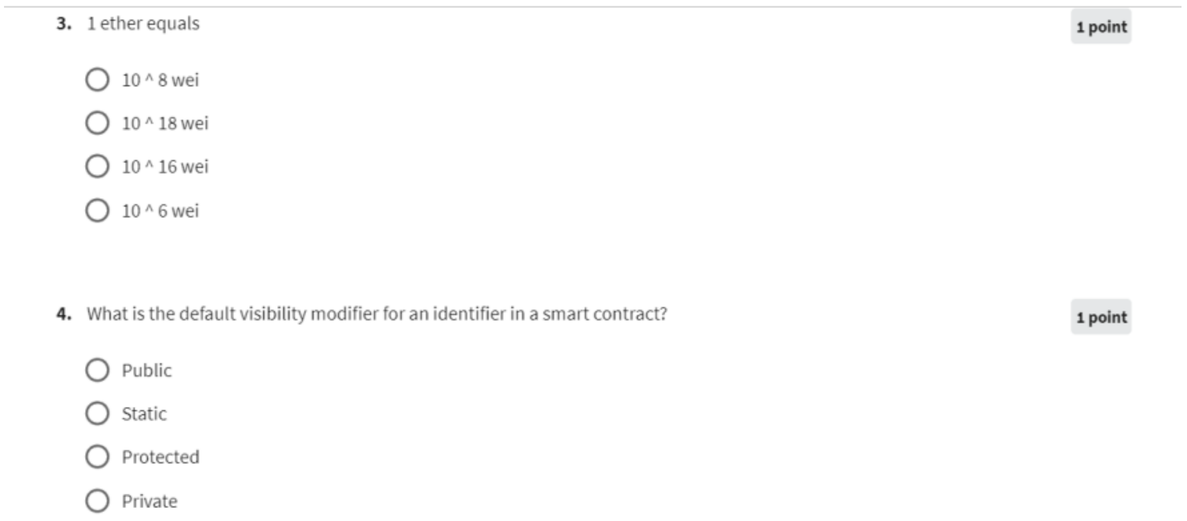
We looked at a few simple data types and their usage through an example. This example should give enough background knowledge to select a type for your data, and use it in your application design. For more of the simple data types refer to solidity documentation in the resources.

2.2.1 Practice Quiz



1. Transacting on the ethereum blockchain has a cost associated with it. Which of these units is used to measure that cost? **1 point**
   - Gas
   - Coin
   - Ether
   - Bytes

2. What is the size of "uint"? **1 point**
   - 64 bits
   - 112 bits
   - 128 bits
   - 256 bits

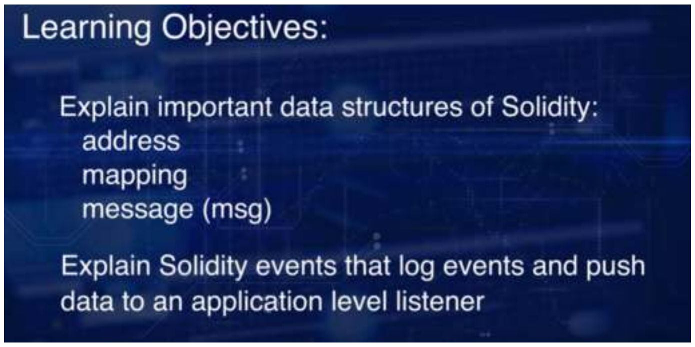


3. 1 ether equals **1 point**
   - 10 ^ 8 wei
   - 10 ^ 18 wei
   - 10 ^ 16 wei
   - 10 ^ 6 wei

4. What is the default visibility modifier for an identifier in a smart contract? **1 point**
   - Public
   - Static
   - Protected
   - Private

2.3 Specific Data Types (Part 1) (Coin Demo)

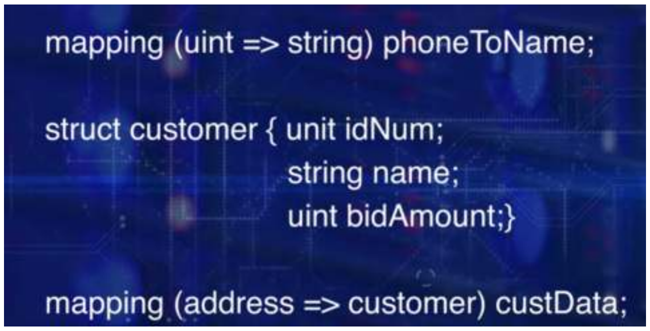
On completion of this lesson, you will be able to explain important data structures of Solidity: address, mapping and message; explain Solidity events that logs events and pushes data to an application level listener. Address is a special Solidity define composite data type. It can hold a 20-byte ethereum address. Recall address is a reference address to access a smart contract. Address data structure also contains the balance of the account in Wei. It also supports a function transfer, to transfer a value to a specific address. Mapping is a very versatile data structure that is similar to a key value store, it also can be thought of as a hash table. The key is typically a secure hash of a simple Solidity data type such as address and the value in key-value pair can be any arbitrary type. Here we illustrate the idea of mapping with two examples. We can use phone number to name mapping.



Learning Objectives:

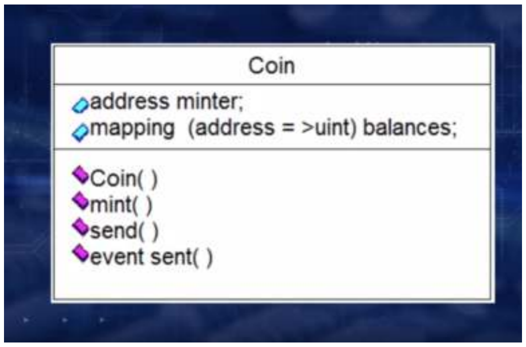
Explain important data structures of Solidity:
- address
- mapping
- message (msg)

Explain Solidity events that log events and push data to an application level listener

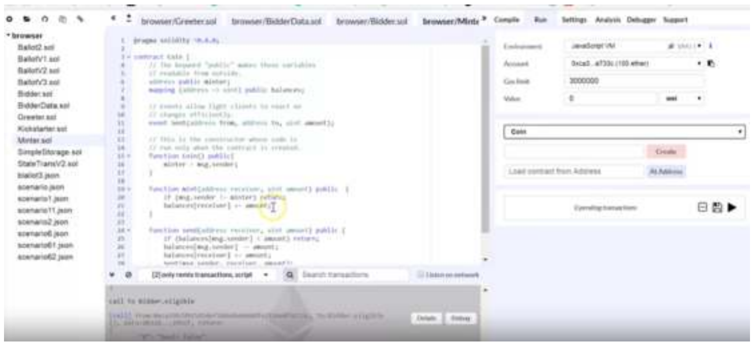


```
mapping (uint => string) phoneToName;

struct customer { unit idNum;
                  string name;
                  uint bidAmount;}

mapping (address => customer) custData;
```

That's a common utility function on our phone system. As a second example, you can have a struct of all customer data and can use mapping to map an account address to customer data as shown. Let's now transition to message. Message is a complex data type specific to smart contract. It represents the call that can be used to invoke a function of a smart contract. It supports many attributes of which we are interested in two of them now. You can always look up other message details in the Solidity documentation, msg.sender that holds the address of the sender, msg.value that has the value in Wei sent by the sender. Now, you can write statements that verify and validate the address and the amount to make sure that the application specifics are met. Let's use these data structures and mint some money. Let's look at the smart contract for coin as specified in Solidity documentation. Recall, we always start with the design, say a class diagram whether we are analyzing a program or coding an application. Here is a class diagram for coin contract. Also make a note of the statements for, if-else assignment. Coin uses the two Solidity features address and mapping. Address holds the 20-byte ethereum address. Recall, address is the base for a smart contract. Let's review this code in Remix IDE.



The next contract that we're going to look at is the coin. I created this under Minter.sol. The name of the contract is Coin and this is also from the Solidity documentation, and we have here one state variable, public minter. You can see that it has a data type of address that is Solidity specific. They are also illustrating the mapping function here mapping table where address is used to map on to the balances that we have. It also illustrates the event, an event has been defined as sent with three parameters: from whom the money was sent, to what address the money was sent, and the amount of money that was sent. We have the constructor where the message sender or the person who's creating this contract is defined as the minter and message sender. We're also illustrating the use of message and the sender data of the message and initializing the minter public data that we have in this contract. Next comes the function mint, where the coins are minted and in this case, only the sender,

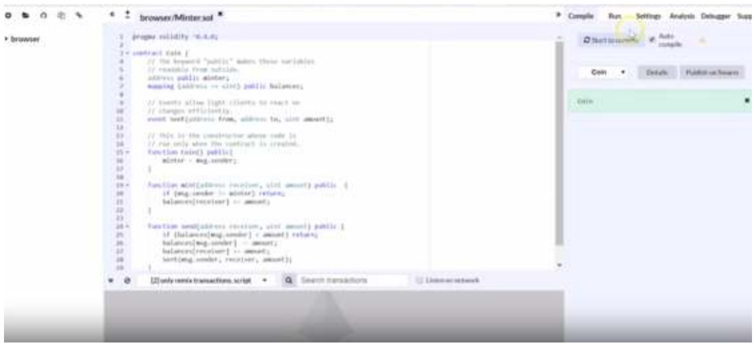


```
pragma solidity ^0.4.0;

contract Coin {
    // The keyword "public" makes those variables
    // readable from outside.
    address public minter;
    mapping (address => uint) public balances;

    // Events allow light clients to react on
    // changes efficiently.
    event Sent(address from, address to, uint amount);

    // This is the constructor whose code is
    // run only when the contract is created.
    function Coin() public{
        minter = msg.sender;
    }

    function mint(address receiver, uint amount) public  {
        if (msg.sender != minter) return;
        balances[receiver] += amount;
    }

    function send(address receiver, uint amount) public {
        if (balances[msg.sender] < amount) return;
        balances[msg.sender] -= amount;
        balances[receiver] += amount;
        Sent(msg.sender, receiver, amount);
```

```
call to Bidder.eligible
```

only the minter should be able to mint the coins. Only the owner of the smart contract should be able to mint the coin. So, I have here a simple command that tests the condition that whoever is the message sender for mint, is indeed the minter who created this contract. At this time, I'm going to close this bidder, we don't need that anymore. Once that is so, once a mint is requested by the owner of the contract and I mint those newly created coins and add to the receiver whose address was sent as a parameter. Finally, I have a send function where the coins created can be sent to a specific address. You specify the address of the receiver and the amount to be sent as a parameter and this is also a public function as you can see. If the balance of the sender of this message is less than the amount to be transferred, the function is returned. This is not completed. Otherwise, the balance of the sender is decremented, balance of the receiver is incremented. There is also an event log as what happened here? The sent is the event and it is pushed, it is logged with message sender and the receiver is the coin and the amount has three parameters. Three is the maximum number of parameters that is allowed for an event, recall that from our lesson, okay. This invokes that event and that gets logged.

2.4 Specific Data Types (Part 2) (Coin Demo cont.)

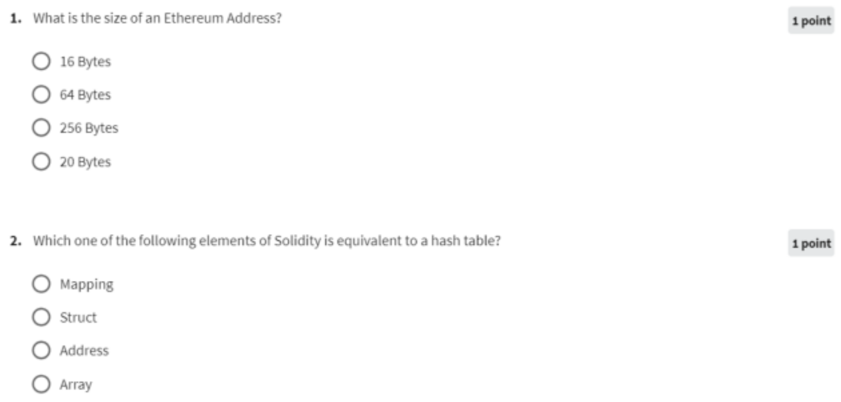


```
pragma solidity ^0.4.0;

contract Coin {
    // The keyword "public" makes those variables
    // readable from outside.
    address public minter;
    mapping (address => uint) public balances;

    // Events allow light clients to react on
    // changes efficiently.
    event Sent(address from, address to, uint amount);

    // This is the constructor whose code is
    // run only when the contract is created.
    function Coin() public{
        minter = msg.sender;
    }

    function mint(address receiver, uint amount) public  {
        if (msg.sender != minter) return;
        balances[receiver] += amount;
    }

    function send(address receiver, uint amount) public {
        if (balances[msg.sender] < amount) return;
        balances[msg.sender] -= amount;
        balances[receiver] += amount;
        Sent(msg.sender, receiver, amount);
    }
```

We looked at the structure of coin, now let's look at it in operation. And let's mint some coins and send it between accounts. Compile, you have a clean compile. Run, and I'm going to create. And that deploys a smart contract with public data and public methods. And I'm going to check the balance of the mentor.
None of that column should have any coins because we just started. So I'm going to say balance and it's going to be zero. If you check the other accounts also, it would be zero, and I'm going to mint some coins for the minter, the creator of the.
I'm going to say 2,000, mint, and we have here, let me go back and check if we have 2,000. Now I'm going to go back to another account, copy the account address, and put it in the send. Make sure it is copied and it's the right address. And I'm going to send 600 of the coin that I created from the minter's account to the new account. And let's go back here, let's go back here, and see whether we have that particular account has that 600 coins that we sent. So here we have balance 600. So we minted 2,000 coins, we sent 600 coins. And now I'm going to go back here to the minter's account. Minter's account should show 1,400 because it has sent 600 of them, balance, yes. We know that it's nicely balanced.
In summary, it is impossible to enumerate every possible language element of solidity.
However, in this lesson we've established a method you can follow to learn and practice new constructs for building smart contract. We learned the concept of address, mapping, and message, and applied these to analyze a simple

contract coin. One more thing. Always design before you code. We did that with the class diagram of the coin example.
And explore the coding examples by uploading them to the remix and map.

2.4.1 Practice Quiz


1. What is the size of an Ethereum Address? 1 point

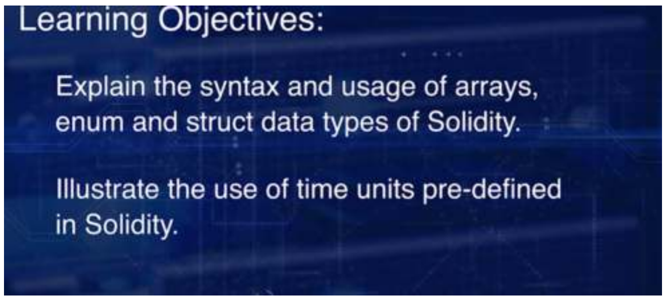
   - ○ 16 Bytes
   - ○ 64 Bytes
   - ○ 256 Bytes
   - ○ 20 Bytes

2. Which one of the following elements of Solidity is equivalent to a hash table? 1 point

   - ○ Mapping
   - ○ Struct
   - ○ Address
   - ○ Array

2.5 Data Structures (Part 1) (BallotV1 Demo)

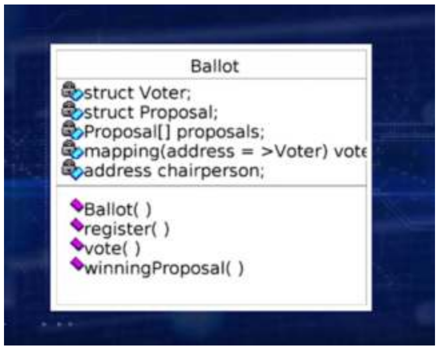
On completion of this lesson you will be able to explain the syntax and usage of arrays, enum, struct data types of Solidity, illustrate the use of time units pre-defined in Solidity. We will begin with a simple smart contract as an


Learning Objectives:

Explain the syntax and usage of arrays, enum and struct data types of Solidity.

Illustrate the use of time units pre-defined in Solidity.

example. We'll develop this example to illustrate the concepts for this lesson, struct, array, enum, and time units. Let us begin our analysis of the code base with a class diagram that visually represents an example, a modified version of the balanced smart contract specified in the Solidity documentation. The smart contract creator is the



chairperson who gets a weight of two for her vote. Others get a weightage of one for their one vote. Each voter has to be registered first by the chairperson before they can vote. They can vote only once. A constant function is included to enable the client applications to call to obtain the result. The constant modifier of the function prevents it from changing any state of the smart contract.

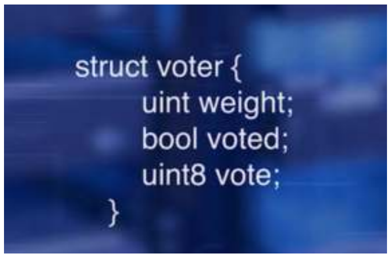
Ballot Smart Contract:

A constant function is included to enable the client applications to call to obtain the result.

The constant modifier of the function prevents it from changing any state of the smart contract.

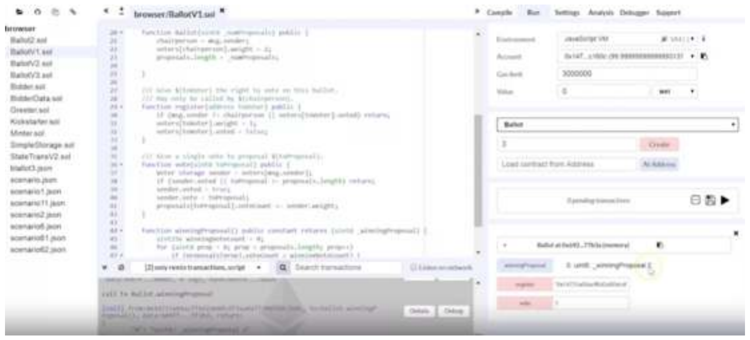
This call comes directly to the smart contract and not via a transaction, so it is not recorded on the blockchain.

More importantly, this call comes directly to the smart contract not via a transaction, so it is not recorded in the blockchain. Since it does not change the state of the smart contract, there is no need. Struct is a composite data type of a group of related data that can be referenced by a single, meaningful, collective name. Individual elements of the struct can be accessed using the dot notation. Here is a struct in ballot smart contract representing a single voter with three attributes: weight of the vote, whether this person has voted boolean, and which proposal the person voted for.



```
struct voter {
        uint weight;
        bool voted;
        uint8 vote;
    }
```

Voter struct is used to define a mapping of address. Another example for struct is a proposal data with just one element proposal number. We could also add another data to the struct, say proposal name that is of string type. Set of proposals is represented by an array of proposals. Let us now review the code on remix to check the usage of the struct and the arrays in the functions: constructor, register, vote, and winningProposals. Let's examine ballot version one. Once again, this is from Solidity documentation, and we have modified it somewhat, we have dropped one of the functions, and we have made it a simple one. It starts with the name of the ballot, and we have several data variables. One is a struct for the voters that has got the weightage of the vote, whether they voted or not, boolean and the vote itself, which proposal they vote it to. Then we have a structure proposal that has got the vote count for each one of them. The proposal array maintains the votes for each one of the proposals. The chairperson address is specified by a state variable and there is a mapping from address to voter that is specified by the variable voters. There are several functions, even though we dropped one of them, there are a couple of functions there to note. One is a function ballot which is a constructor, and then I have a register where that chairperson registers the voter and the person or the people who registered vote for it, vote for the proposals that are there and finally, there's a function winningProposal that can be called by the application. It doesn't come through a transaction. This will determine the winningProposal and it can be obtained by a client application. Let's run it and see. You can see that there are several conditions specified, and all the code is available inside the function. We'll not go into the details, but well, let's compile. We know, we understand the meaning of voting and deciding the proposal that wins. Let's compile and run. I'm going to go into JavaScript VM, and let's wait for a few minutes so that all the addresses are generated. These are the addresses that are available to you, and we will be using several of them. For the sake of demo, I'm just going to use one of them as a chairman, as you can see the ca, the one that start with ca. 0x states that there's a hexadecimal address, ca, the one that starts with ca3 is the chairperson. I'm just going to use just one other person for the sake of demo. But if you want to try it on your own, you can try any of these addresses as other people who may be interested in voting. You can try it at your leisure. So, let's create the smart contract. But in this case, when you're creating a smart contract you have to tell how many proposals are there since the constructor requires



more than certain number of proposals. So, I'm just going to say three for the sake of simplicity. This is the chairperson. Chairperson is going to create the smart contract, and I did that. Here we have a simple interface since we had just three functions, and these functions are available in the web interface for us to use. First, I'm just going to simply press the winningProposal. Remember, winningProposals shouldn't show anything or shouldn't be displaying anything, should not be executing at all when nobody has voted, nobody has registered. But we're going to click on the winningProposal, and it shows it as zero. This is an error or a issue with the design that we have right now. Let's move on. So, I'm going to register one person and start voting here. So, in order to register you have to be the chairperson here. I'm going to go into the other person to address. This is the person, other voter and I'm going to copy the address of this person and put it in here.

Addresses have to be within quotes and remix, so I'm just going to put it within quotes. These are all big numbers because they are 256 bits and I got to go back to the chairperson because only the chairperson can register. So I'm issuing this register from the chairperson's address, and this transaction goes from the chairperson to register this person. So, I'm going to register that person. One person has been registered. Now, let's vote. We have one chairperson and one voter. Just for the sake of understanding the balloting process, we are going to use only these two people. I'm going to vote for, let's say the chairperson wants to vote for the proposal number 01 or two. The chairperson votes were two. So this is the chairperson, the chairperson is initiating the vote transaction and number two is a parameter, the chairperson is voting for two. I'm going to press, and that's going through. You can see all the transaction going through here. Then I'm changing the address of the originator of the message to the next person who has registered. Then I'm going to say, "Okay, the voter who is not a chairperson is voting for one." So, understand what happened here. The chairperson voted for two. The regular voter voted for one and we want to find out the winningProposal. Remember, chairperson has a weightage of two. So obviously, it has to be two. So you can see the winningProposal is two because the chairperson, even though there was only one vote weightage was two and the winningProposal was two. So that is a very simple demo of the ballot contract that is given in the Solidity, a documentation, and we modified it somewhat by dropping a function. But we will take this base contract and improve on it by adding enums, stages, time, require, modifiers, and other kinds of check modifiers and other kinds of assertions so that it is a robust solution. On Remix IDE, you can save the list of transactions and view it to explain the sequence of operation. Here is a sample transaction. Can you guess what is wrong with this transaction? The transaction did execute consumed gas, but did not provide the result expected because it was from account two. Remember, nobody except the chairperson account zero can register. It has to be from account zero. We made a mistake in not selecting that right from address for registering. Check out the other correct transaction to understand this further.

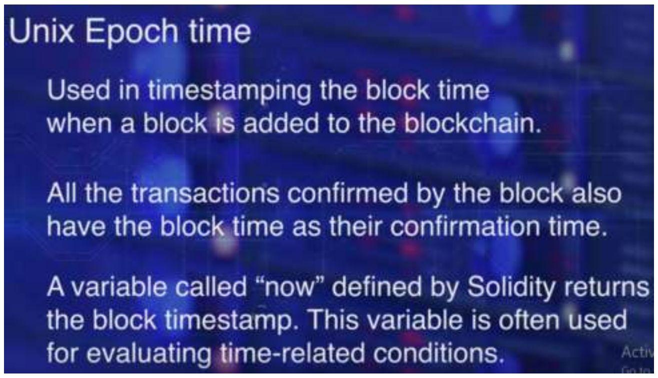
2.6 Data Structures (Part 2) (StateTransV2 Demo)

Let's discuss time units in solidity. In a blockchain application, all the participants and nodes have to synchronize to one universal time. For this purpose, blockchain proposals include a time server that serves the Unix Epoch time or time since January 1st 1970 in seconds.

See this link for a sample conversion of a real date time to Unix Epoch time. This time is used in timestamping the block time. When a block is added to the blockchain, all the transaction confirmed by the block also have the same block time as their confirmation time. A variable called "Now" defined by solidity, returns the block timestamp.

This variable is often used for evaluating time related conditions. In other words, now variable in a function is not the time at which function transaction was initiated, but it is the time when it was confirmed. Time is defined as unit time; seconds, minutes, hours, days, weeks and years. Little numbers can be used in specifying

EpochConverter
Epoch & Unix Timestamp Conversion Tools
The current Unix epoch time is 1519848950
www.epochconverter.com

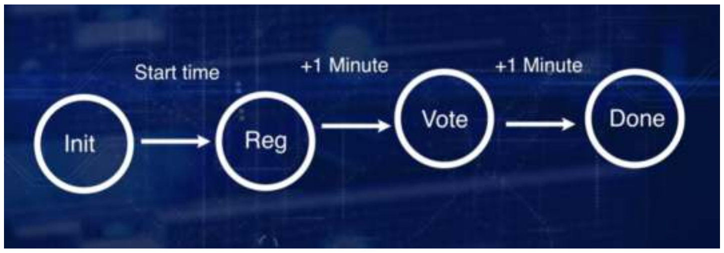

Unix Epoch time

Used in timestamping the block time when a block is added to the blockchain.

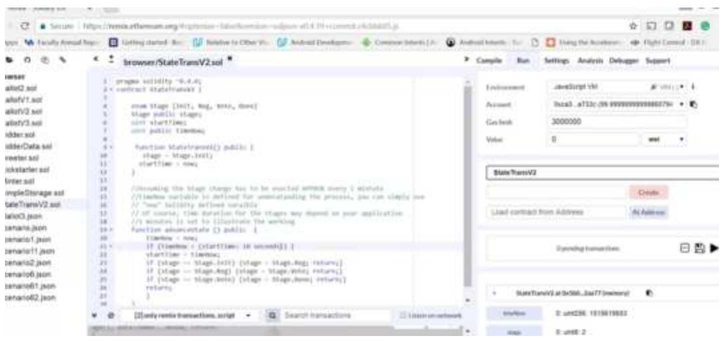
All the transactions confirmed by the block also have the block time as their confirmation time.

A variable called "now" defined by Solidity returns the block timestamp. This variable is often used for evaluating time-related conditions.

time related computations. For example; Assume one day is allocated for the duration of voter registration and if we allow 60 Minutes as the duration time to vote. Assign now value to creation time and after some elapsed time, if now is greater than the creation time plus one days registration status is done. After this If now is less than vote start time plus 60 minutes, then allow user to vote. Recall that now in smart contract is not the time at which the transaction was initiated, but it is the time when it was confirmed. Now, onto Enum data types. Enum or enumerator data type, allows for user defined data types with limited set of meaningful values. It is mostly used for internal use and are not supported currently at the ABI level of solidity. However, it serves an important purpose of defining states or phases of a smart contract. Recall that smart contract represent projects of contracts that may transition through radius spaces based on time or input conditions. Let us look at a simple example that combines time and the enumerated type elements. We define the enumerated type stage; Init, Reg, Vote, Done. And we transition between the stages based on elapsed time. Note that we have used one minutes or even less elapsed time, so that we can verify the execution of this contract in a finite time.



In reality, you can set the time to something realistic such as one day or thirty days.. Etc. Let's look at the state transition on Remix to reinforce this concept. Let's look at the state transition where we are considering only time-based state transitions. Here we have contract state transition version two. Here, I'm illustrating enum stage; Init, Reg, Vote and Done and we are defining the enum followed by the state variables stage. This is uppercase stage for the enumerated variable. That is the type and the stage itself is the data variable. I also have a start time variable and a time now variable, only the time now is public. And the function state transition version two is a constructor where I'm initializing the stage to Init and the start time to now. And now assuming there is a stage change has to be enacted approximately one minute, I'm going to reduce it for the sake of this demo to 10 seconds, because one minute takes too long a time for us to wait for the demo to elapse. So I'm just going to ten seconds. So instead of one minute, I'm changing it to ten seconds.



browser/StateTransV2.sol

```
pragma solidity ^0.4.0;
contract StateTransV2 {

    enum Stage {Init, Reg, Vote, Done}
    Stage public stage;
    uint startTime;
    uint public timeNow;

     function StateTransV2() public {
      stage = Stage.Init;
     startTime = now;
    }

    //Assuming the Stage change has to be enacted APPROX every 1 minute
    //timeNow variable is defined for understanding the process, you can simply use
    // "now" Solidity defined variable
    // Of course, time duration for the Stages may depend on your application
    //1 minutes is set to illustrate the working
    function advanceState () public  {
        timeNow = now;
        if (timeNow > (startTime+ 10 seconds)) {
        startTime = timeNow;
        if (stage == Stage.Init) {stage = Stage.Reg; return;}
        if (stage == Stage.Reg) {stage = Stage.Vote; return;}
        if (stage == Stage.Vote) {stage = Stage.Done; return;}
        return;
        }
    }
```

Environment: JavaScript VM
Gas limit: 3000000
Value: 0 wei
StateTransV2
timeNow 0: uint256 [illegible]
stage 0: uint8 2

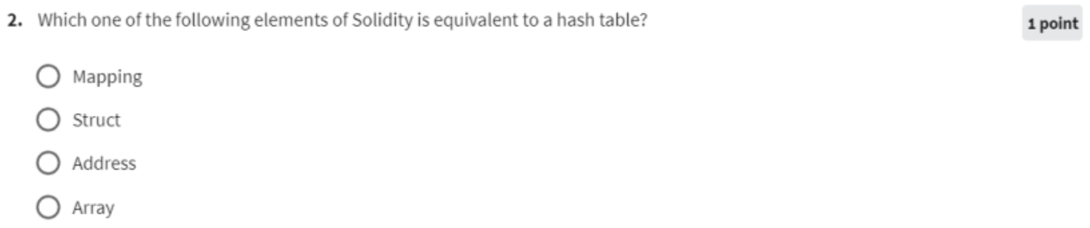
And you saw the little red button with an X up here that means something is wrong and you can change it. So I'm going to change the stage time to 10 seconds, so that we can go through the stages quite quickly for the demo purposes. Okay. Let's run it. Let's compile and run it. And you can see that this is simply going through the stages every 10 seconds. Compile, run it and I'm going to create the smart contract. It is deployed and the web interfaces available here on the web interface of the Remix ID. Time now, it is at zero, stage is set to zero. That is; it is at Init. InIt is zero, Reg is one, Vote is two, Done is three. Even though enum has these names, It is actually coded as zero, one, two and three. And I go to advanced state, and I look at the stage now, it is one. 10 seconds have elapsed and so it has moved on to one. And I'm going to advanced state again and look at the stage, it is still one. It is not at ten seconds. Now I get the time now, you can see it here. And this is the epoch time from January 1st 1970. Advanced state again and I can look at the stage two. Ten more seconds and advanced state, it is not at 10 seconds. Advanced state, and I should be getting now, advanced state I should getting it to stage three. See even though I've given ten seconds for the sake of demo and you can also look at the time it would advance approximately to the three ten seconds. Even though it is based on 10 seconds, In reality it would be a longer elapsed time for the smart contracts.

In summary, we have explored time and stages of contracts that are very important for many real world contracts based applications. We also illustrated Enum, user defined data type and more examples for struct and array data type.

2.6.1 Practice Quiz

1. What is the size of an Ethereum Address? **1 point**

- ○ 16 Bytes
- ○ 64 Bytes
- ○ 256 Bytes
- ○ 20 Bytes

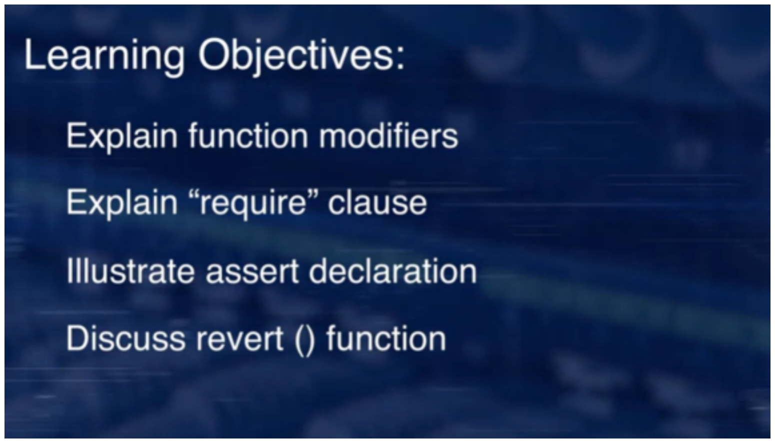


2. Which one of the following elements of Solidity is equivalent to a hash table? **1 point**

- ○ Mapping
- ○ Struct
- ○ Address
- ○ Array

2.7 Access Modifiers Applications

On completion of this lesson, you will be able to explain usage of function modifiers, explain the use of require clause for input validation, illustrate the assert declaration for post-condition checking, discuss reverting a transaction and reward function. The main intent of smart contract transaction is to execute a function. However,



Learning Objectives:

Explain function modifiers

Explain “require” clause

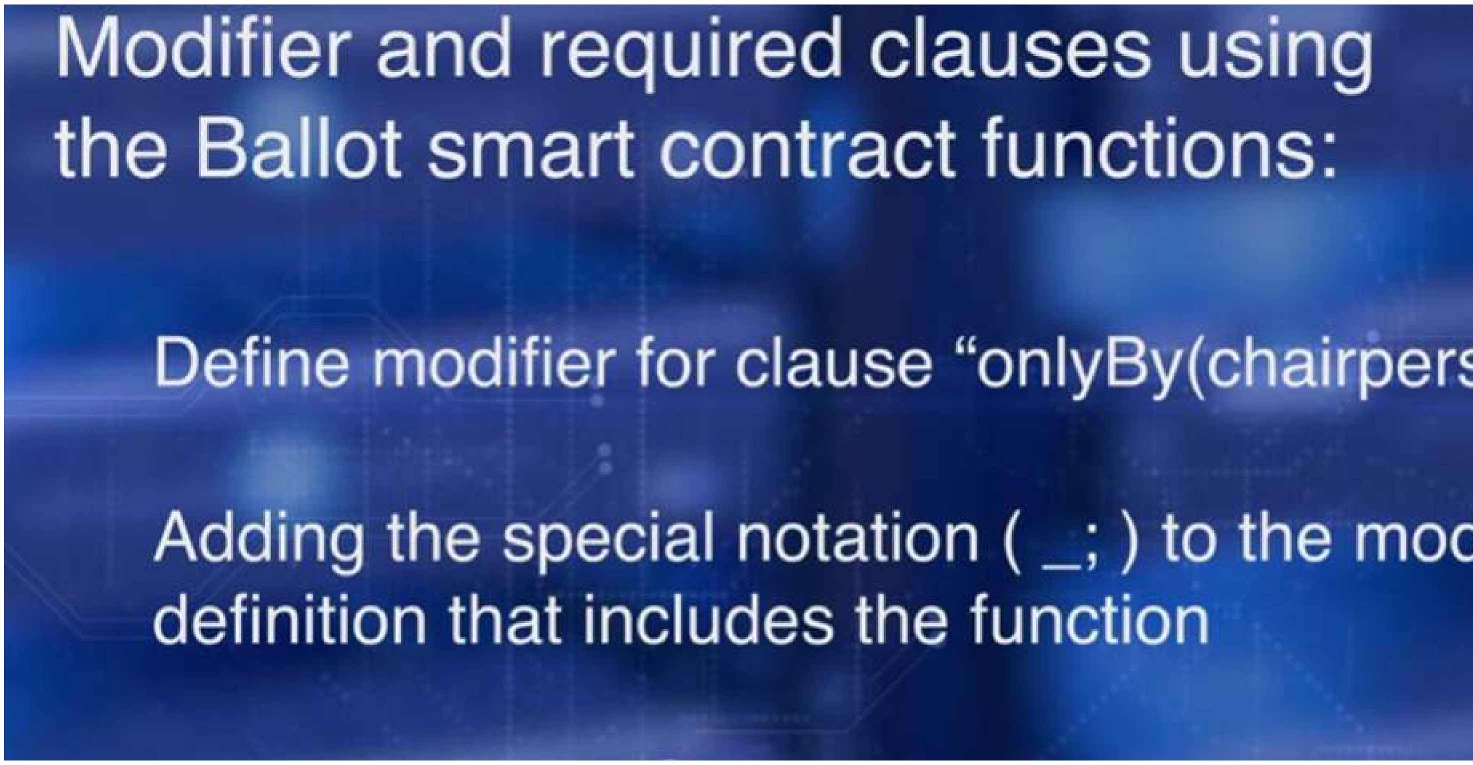
Illustrate assert declaration

Discuss revert () function

smart contracts often require control over who are what can execute the function, at what time a function needs to be executed, what are the precondition to be met before getting access to the function? It’s a good practice to validate input values to the function to avoid unnecessary execution and waste of gas. If there are any errors found during the input validation, these have to be handled appropriately. At the end of the function execution, you may want to assert that certain conditions are met to ensure the integrity of the smart contract. Let’s begin with an important feature of solidity, modifiers. They can address some of the concerns just mentioned. Modifiers can change the behavior of a function. That’s why this feature is referred to as a modifier. It is also known as a function modifier since it is specified at the entry to a function and executed before the execution of the function begins. You can think of a modifier as a gatekeeper protecting a function. A modifier typically checks a condition using a require and if the condition failed, the transaction that call the function can be reverted using the revert function. This will completely reject the transaction and revert all its state changes. There’ll be no recording on the blockchain. Let’s understand the modifier and require clauses using the ballot smart contract functions. We’ll add a modifier onlyBy(chairperson) to the register function. Will do that by the steps. Define modifier for the clause onlyBy(chairperson). Adding a special notation underscore semicolon to the modifier definition that includes the function.

Using the modifier clause in the function header. Step 1 is a definition of the modifier onlyBy, only the chairperson who created the smart contract valid to register any new orders. Step 2, shows how to make the modifier include a place holder for any function. Step 3, use the modifier clause in the header of the function definition. Now, let’s change the existing if statement of the ballot smart contract such that it reverse the transaction on input validation failure. We don’t need to waste blockchain resources for a failed transaction. Now, we’ll illustrate assert using a function payoff that computes and pays off bets. It is preferable that the assert not fail and that it requires checking the balance after every payoff in the above case. We are making sure bank has the reserves of 10,000 after all the payoffs. Under normal circumstance, assert should not fail. This is more for handling an anomalous, faulty or malicious condition. In this case, the exceptional condition is that bank balance somehow dipped below the reserves. The picture summarizes all the features we discussed. Modifiers and error handlers, and where they are typically used. The rules, laws, policies, and governance conditions are coded as modifiers. You can use the modifiers as gatekeepers for the functions. If your transaction to invoke the function does not meet the condition specified at the header of the function, your transaction will be reverted. Will not be recorded in the block chain. Modifiers can be used to validate rules outside the function such as describing the opening of the lesson, who has access to the function, at what time, and what condition, etc. Once the screening conditions are satisfied, input validation can be carried out inside the function using require declaration statement.

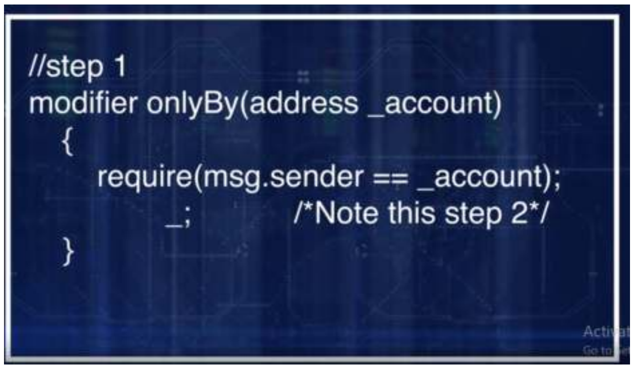
In this case, also the transaction will be reverted on failed validation. This is done before the execution of the function. There are anomalous faulty malicious code that may result in unexpected situations or exception. These can be caught usually at the end of the function or sometimes within the function using assert declarative statement, and the entire transaction including the function execution and the state change will be reverted. Here



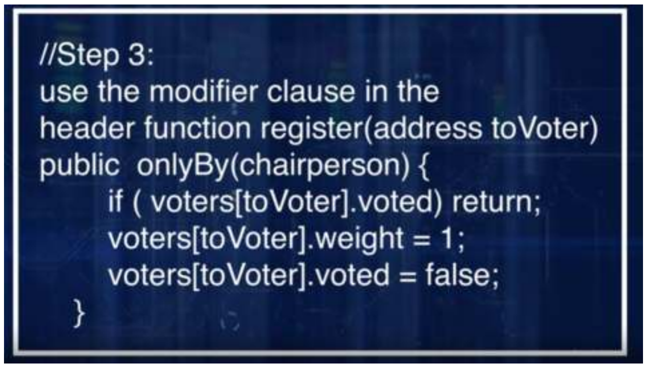
Modifier and required clauses using the Ballot smart contract functions:

Define modifier for clause "onlyBy(chairpers

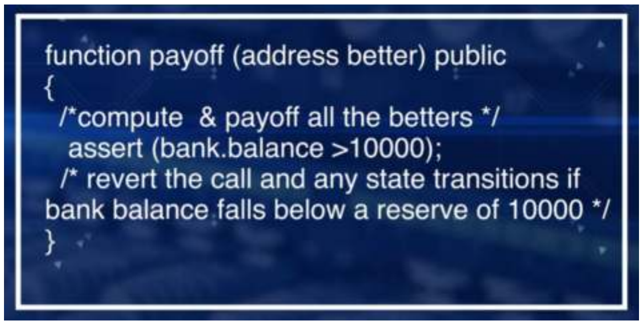
Adding the special notation (_;) to the moc
definition that includes the function



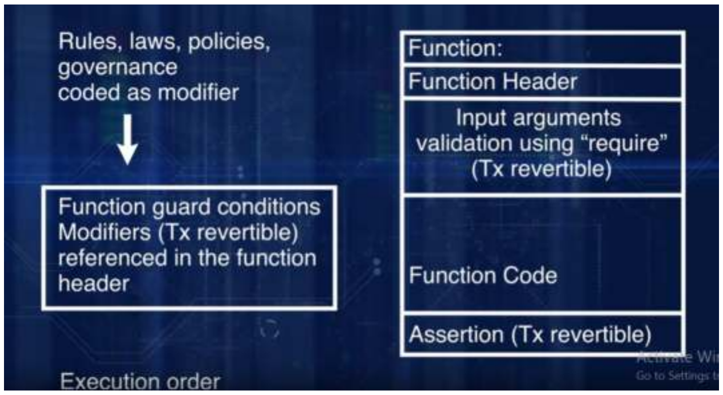
```
//step 1
modifier onlyBy(address _account)
   {
        require(msg.sender == _account);
              _;            /*Note this step 2*/
   }
```



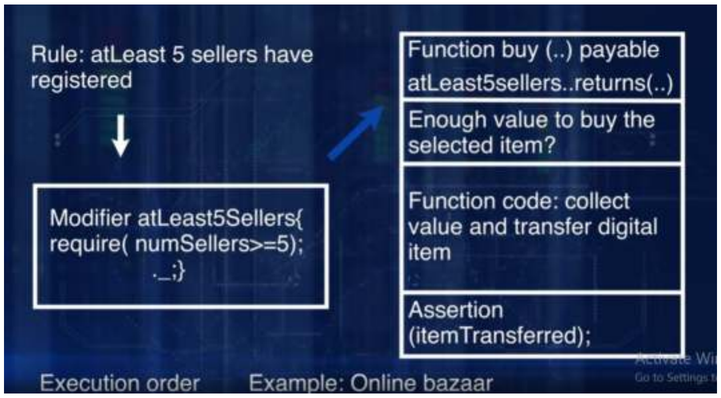
```
//Step 3:
use the modifier clause in the
header function register(address toVoter)
public  onlyBy(chairperson) {
        if ( voters[toVoter].voted) return;
        voters[toVoter].weight = 1;
        voters[toVoter].voted = false;
   }
```



```
function payoff (address better) public
{
  /*compute  & payoff all the betters */
   assert (bank.balance >10000);
  /* revert the call and any state transitions if
bank balance falls below a reserve of 10000 */
}
```

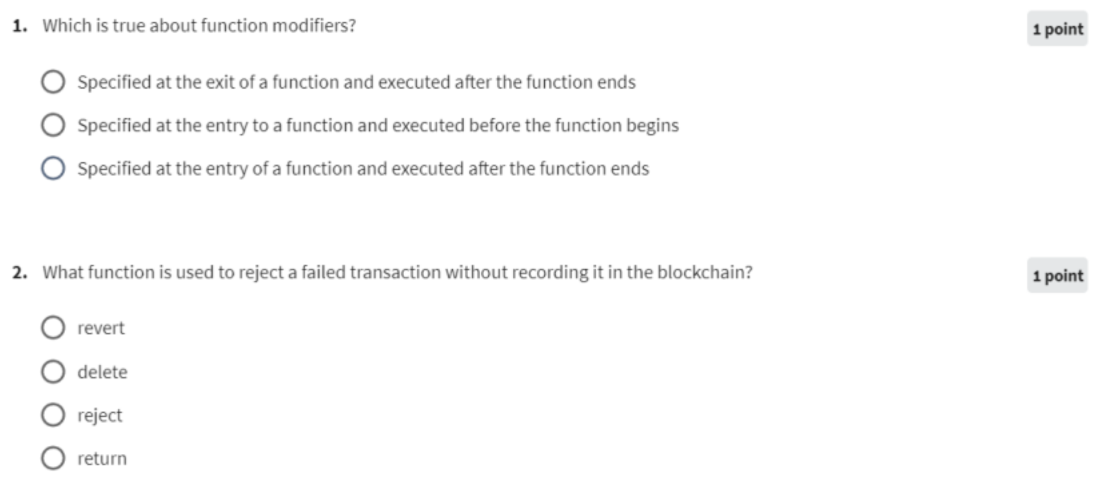

Rules, laws, policies, governance coded as modifier
Function guard conditions Modifiers (Tx revertible) referenced in the function header
Function:
Function Header
Input arguments validation using "require" (Tx revertible)
Function Code
Assertion (Tx revertible)
Execution order

is an example of an online digital media bazaar. Input condition as specified by a modifier is atLeast5Sellers. This enforces the condition that there should be atLeast5Sellers with their products before a buyer initiates a buy message through a transaction. This is a global condition for all buyers. So, it is appropriate to install it using a


Rule: atLeast 5 sellers have registered
Modifier atLeast5Sellers{
require(numSellers>=5);
_;}
Function buy (..) payable
atLeast5sellers..returns(..)
Enough value to buy the selected item?
Function code: collect value and transfer digital item
Assertion (itemTransferred);
Execution order
Example: Online bazaar

modifier declarative statement. Once there are enough sellers, a buyer can buy. Inside the function, we validate if the buyer has enough money to buy the specific item requested. If not, the transaction is reverted. When the validation is satisfied, money is transferred to the seller, item is transferred to the buyer. We have a simple assert declaration at the end that the bought item should have been transferred. After all, it's a digital item. This simply reached only if the item has not been transferred and could not be transferred due to exception reason. This provides a simple example of concepts learned in this lesson. In summary, function modifiers along with state reverting functions of revert, required, and assert, collectively supported robust error handling-approach for a smart contract. These declarative features can be used to perform formal verification and static analysis of a smart contract to make sure it implements the intent of a smart contract.

2.7.1 Practice Quiz



1. Which is true about function modifiers? **1 point**

- ○ Specified at the exit of a function and executed after the function ends
- ○ Specified at the entry to a function and executed before the function begins
- ○ Specified at the entry of a function and executed after the function ends

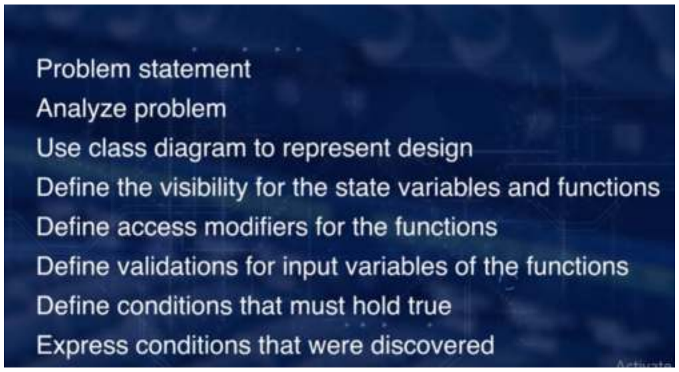
2. What function is used to reject a failed transaction without recording it in the blockchain? **1 point**

- ○ revert
- ○ delete
- ○ reject
- ○ return

Chapter 3

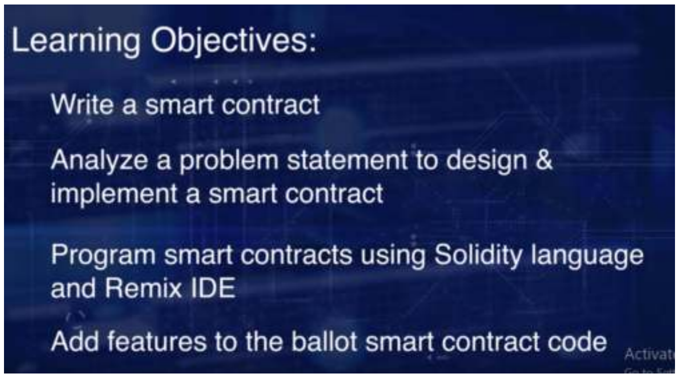
3.1 Putting It All Together: Developing Smart Contracts

We'll start this module with a discussion of a method for developing smart contracts. Begin with a problem statement, analyze the problem to come up with a basic design, its state variables and functions, recall our principle from the last lesson, design first, represent the design using a class diagram. Based on the problem statement, define the visibility for the state variables and functions. Based on the requirements, define the access modifiers for the functions, define validation for input variables of the functions, define conditions that must hold true on completion of critical operations within functions, declaratively express the conditions that were discovered in steps four to seven using access modifiers, visibility modifiers require an assert classes. We will use



Problem statement
Analyze problem
Use class diagram to represent design
Define the visibility for the state variables and functions
Define access modifiers for the functions
Define validations for input variables of the functions
Define conditions that must hold true
Express conditions that were discovered

solidity language and remix IDE to develop and test a smart contracts according to this method. Upon completion of this module, you will be able to apply the concepts learned in the last two modules to write a smart contract, analyze a problem statement to design and implement a smart contract, program smart contracts using solidity language and remix IDE. Incrementally, add features to the ballot smart contract code. Let's review the ballot



Learning Objectives:

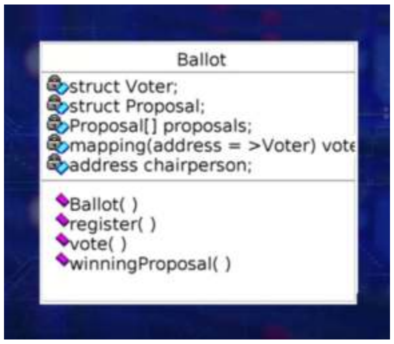
Write a smart contract

Analyze a problem statement to design & implement a smart contract

Program smart contracts using Solidity language and Remix IDE

Add features to the ballot smart contract code

problem. We'll use the ballot problem that is defined and solidity documentation. This problem will be used as a base problem on which we'll develop our solution. Recall that we discussed the ballot problem in the last module. We have chosen a familiar problem so that we can focus on designing and implementing the smart contract, rather than spending time on explaining the problem that is unfamiliar to you. Moreover, the concepts and the function used in the solution can be easily adapted to any other problem. Here is a formal problem statement for ballot. Problem description, version one: An organization invites proposals for a project, a chairperson organizes this process. The number of proposals is specified at the time of creation of the smart contract. The chairperson registers all the voters. Voters including the chairperson vote on the proposal. To make things interesting, we've given a weight of two for the chairperson's vote and a rate of one for the regular voter. The winning proposal is determined and announced to the world. We'll exclude the delegation of voting function that is present in the current valid smart contract in solidity documentation, we will not consider this function. Let's perform an analysis to determine the state variables. Let us now consider the data or state variables needed for the program. Details of a proposal and the set of proposals will keep track of only the proposal number and the vote for each proposal.

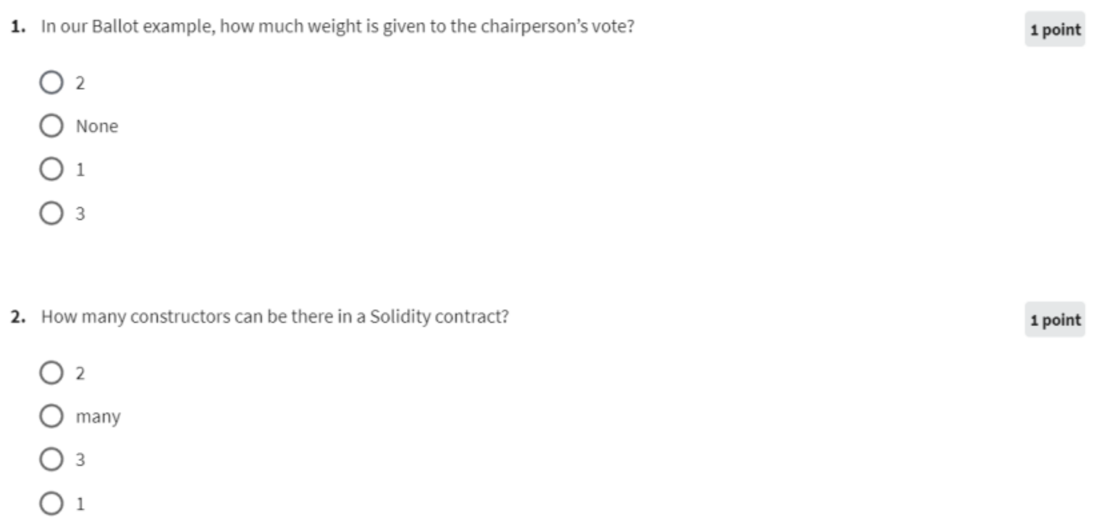


Auto details, the vantage of the vote, one for the regular voter, two for the chairperson. Voted or not, and to what proposal number they voted. Keep track of the addresses of the chairperson and other voters, address of a voter will be used as a key to map to the details of the voter. Major differences from the traditional object oriented analysis is in the smart contract specific data types, such as address and the message center. Let us now discuss the functions. The chairperson is the creator of the smart contract. He or she will be the only person who can register the voters. Here is a list of functions. Constructor is a function that is called to create the smart contract. In solidity, unlike a regular object oriented language, there can be only one constructor. Also, the constructor has the same name as a contract. The sender of the message invoking the constructor is the chairperson. The second function is the register function to register the voter. Only the chairperson can register a voter. The sender of the message for registration has to be the chairperson. The third function is the vote function. Voters including the chairperson. can vote for a proposal. The final function determines the winning proposal and that can be called by client application. Let's look at remix to explain the ballot version one. Let's recap what we did in remix ballot version one. Note that the visibility modifiers for the state variables and the functions represent the details of a single voter using a struct. Single proposal also using a struct possibly allowing for future expansion. A mapping of voters, mapping voter address, externally-owned accounts to voter details. An array of proposals and a chairperson address are defined. Also, review all the function. Try it for yourself.

3.1.1 Practice Quiz

3.2 Time Elements (Part 1)

Let us re-examine the ballot contract developed in the last lesson. In a typical voting process, voters are registered first. There is usually a deadline for registration, and also for the voting period. For example, for most states in the USA, you have to be registered 30 days before the voting day, and the ordering takes place on a single day for in-person voters. If that is the case, registration has to be completed before voting. The current ballot smart contract does not have these limitations. For example, the function register and the function vote can be called in any order. There are no rules such as the voters have to be registered before they can vote, voting is open only for a specified period, and the winning proposal can be decided only after all the voting is completed. Currently, if you call the winning proposal, it gives the zeroth proposal as the winner before anyone has registered or voted. Now, let us add the stages and the time duration of the stages to the ballot version one. Create a ballot version two, ballot version two.all on Remix, by copying ballots all we created earlier. We add the enum for the stages and logic for modifying the stages within the functions. We will compile and run it and make sure it works as expected. Let's define the stage as enum datatype. Stage is four distinct stages, Init, Reg, Vote, and Done. The stage is initialized to Init at the time of deployment of the Smart Contract. Then, in



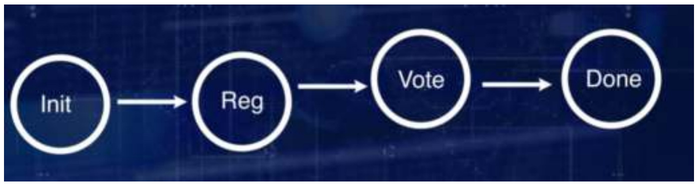
1. In our Ballot example, how much weight is given to the chairperson's vote? 1 point

- ○ 2
- ○ None
- ○ 1
- ○ 3

2. How many constructors can be there in a Solidity contract? 1 point

- ○ 2
- ○ many
- ○ 3
- ○ 1

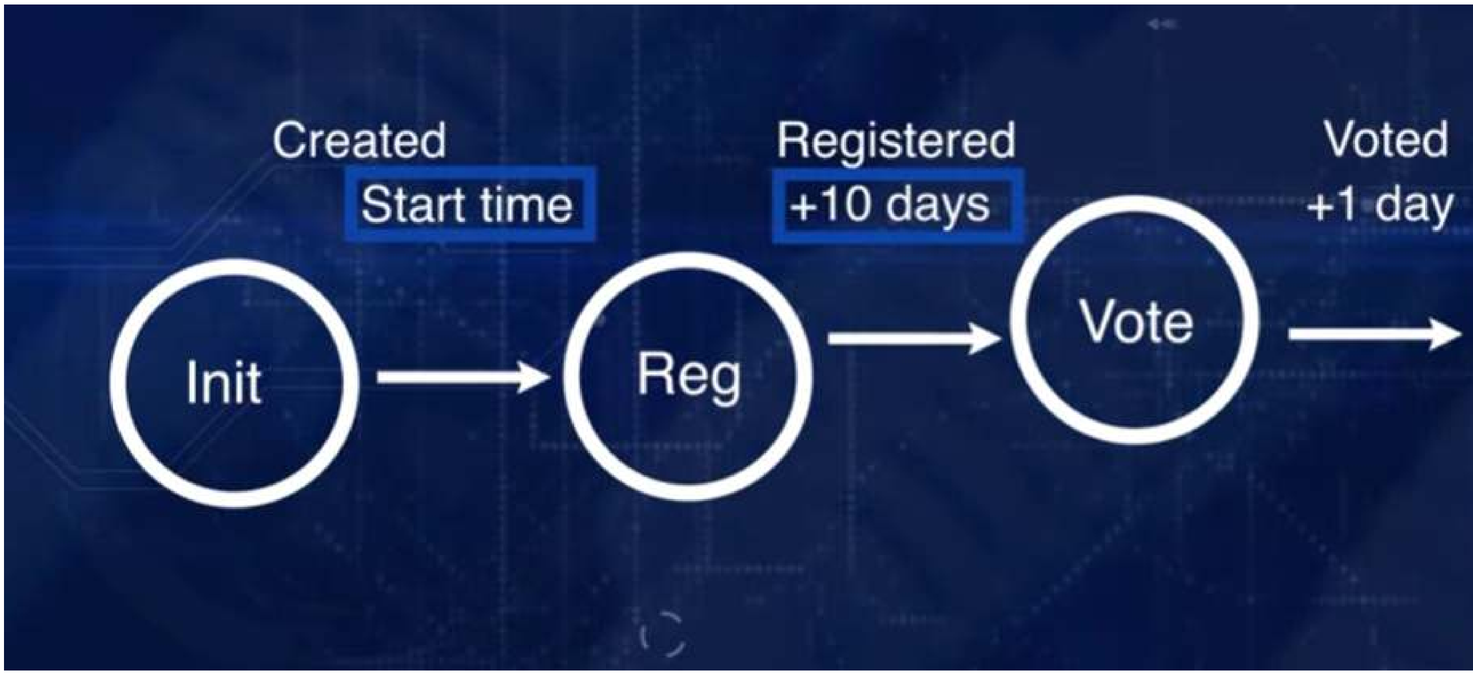
the constructor, the Init is changed to Reg stage. After the registration period is over, the stage changes to Vote. After the voting duration elapses, the stage is set to Done. Enum, stage, Init, Reg, Vote, and Done. Let's

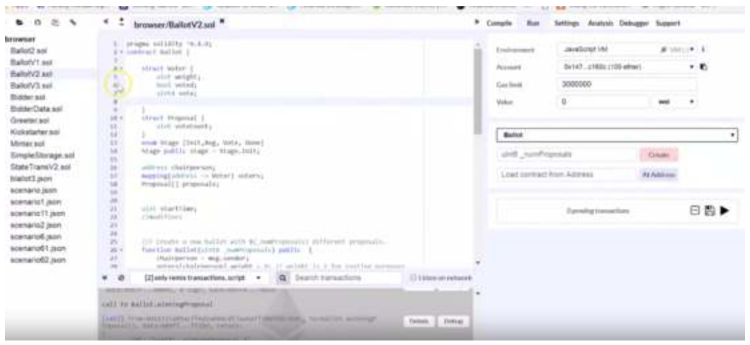


add this logic to the ballot two.soldsmartcontract, and use this to set the stages of the smart contract. We'll also add the time elements. Solidity defines a time variable "now", that is the current block timestamp. We'll add the state variable, startTime, uint startTime. startTime is initialized to now within the constructor. Then, change the stage of the ballot process based on the time allocated for registration and voting stage as shown. We have added the startTime variable, and the period for registration in this case is 10 days. We also added the duration for voting period, in this case it is one day. The now solidity variable is the timestamp of the block, block.timestamp function, in which the transaction is confirmed. Thus, now may not accurately reflect the elapsed time. For approximate intervals, and for testing simple concept, now is a convenient time attribute. In a realistic application, with specific deadlines, a better solution will be to pass the deadlines in a epoch time to the constructor of the smart contract at the time of creation, and compare it with the current block timestamp where needed. Recall that block timestamp is represented by the variable "now".

3.3 Time Elements (Part 2) (BallotV2 Demo)

This is Ballot version 2. Version 2 has a struct for Voter, it has a struct for Proposal. And has something new here, enum stage, that defines init reg vote and done stages for the smart contract.
And the variable stage is defined to be of this enum stage type, and it has been initialized to the Init stage. Actually Init is 0, Reg is 1, Vote is coded as 2, Done is coded as 3. Remember that when we are looking at the outputs in the web interface.
Address chairperson, chairperson is the only person who can register other voters. Mapping, address is mapped to the voter struct, and we have a Proposal and an array of proposals that has each one of the structs for the Proposal. That is just one data field called uint voteCount.
And we also define the time element uint startTime. These are some of the things that are new. We have the usual constructor, and the constructor has two more lines besides the previous code that is defining the stage to be reg, and also start startTime to be now. So, it initiates the registration stage.
And the register function happens only if the stage happens to be reg, otherwise it simply returns. This is a new





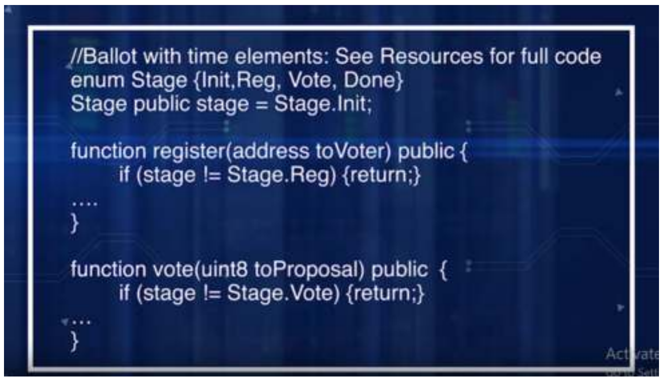
```
pragma solidity ^0.4.0;
contract Ballot {

    struct Voter {
        uint weight;
        bool voted;
        uint8 vote;

    }
    struct Proposal {
        uint voteCount;
    }
    enum Stage {Init,Reg, Vote, Done}
    Stage public stage = Stage.Init;

    address chairperson;
    mapping(address => Voter) voters;
    Proposal[] proposals;


    uint startTime;
    //modifiers



    /// Create a new ballot with $(_numProposals) different proposals.
    function Ballot(uint8 _numProposals) public  {
        chairperson = msg.sender;
```

Environment: JavaScript VM
Account: 0x147...c160c (100 ether)
Gas limit: 3000000
Value: 0 wei
Ballot
uint8 _numProposals
Load contract from Address

call to Ballot.winningProposal

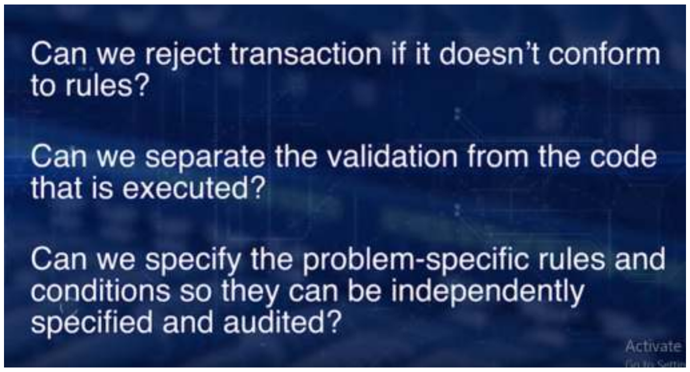
feature that we have added to the Ballot version 2, okay? It used to be just this code, now we have added one more check in the validation at the beginning of the function, okay. And that is based on the enum stage that we've created. The same way you can vote only if the stage happens to be, I'm going down here and you can see that if you can vote only if the stage happens to be vote, otherwise the function simply returns.
And finally, and the winning proposal returns something useful only if the stage is done.
Otherwise, it doesn't even go through computing. This winning proposal doesn't even execute if the stage is not done. All right, so we enforce some checkpoints here, some validations here from the first version of the Ballot. So let's execute it and see it running. I'm going to compile, it's already compiled, but anyway compile again. JavaScript, and I am going to change that address like you can see that we have several addresses and I am going to change that address to be the chairperson's address, that is arbitrarily decided. But I would like to have the ca3 to be chairperson's address every time, and I am going to create it once again with just three proposals, Create, and you can see the interface here, okay. So I have here vote register, and I also put the stage, stage also came up here, since I define the stage variable to be public. It is also coming and showing in the web interface, since it's a public variable, getters are automatically set for it, and so stages also here, so you can view the stage also. All right, so first thing is we need to register. So the chairperson is here sending, ready to send the message. And I want to find out who is to be registered first. This is the second person, this the first person who is to be registered and so I'm going to copy that address and then throw it in here. And remember, remix requires it within quotes. And I have to go back here and create or show the chairperson as the sender of this register message. Nobody else except the chairperson can register. So I set the Account address who's sending the register function to be the chairperson. And then this is the person to be registered, and I say, register.
Okay, now that person is registered so we have two people. One, the chairperson, and one more. I'm going to show you registration for one more person, let's see. I have here one more person, copy, and I can copy it here. And again, I want to emphasize that this is going to be within code, and I just want to make sure that I have a different address. You can see it's 4b and I have to change the sender to be the chairperson and then I'm going to register. We've registered two people here. Okay, let's register one more.
583, I'm going to copy, so you got the idea here right now, and this has to be within codes, and you can use a Ctrl+V to copy that, you could use copy and paste if you're a copy and paste person. Let's see, Ctrl+V, I have it here, all right. And so, I go back to the chairperson, and register. In this case I registered three people here. Okay, and the stage is two now, okay, so stage is two, that means elapsed time is ready to go and we can vote. So I'm going to go back to the chairperson. Chairperson votes for let's say 1, okay that means two votes are registered for 1 okay. And I'm going to do only one more vote because for the sake of I don't want copy. So I"m just going to, the second person votes for 2. Okay, the regular voter votes for 2. All right, and now let's look at the stage. Okay 3, it's ready to go, it's all done. So the winning proposal is 1, because remember, the chairperson voted for 1, the single voter voted for 2, the chairperson weighted is 2. So the winning proposal is one, all right? You can also keep checking the stages, every now and then, whether you are in the right stage. In this case I made the stage time to be very small so I could go through it quickly. But in reality, in a real smart contract it can be few days that it be allowed to vote, and so on. Right, that is the end of Ballot 2.

3.4 Time Elements (Part 3)



```
//Ballot with time elements: See Resources for full code
enum Stage {Init,Reg, Vote, Done}
Stage public stage = Stage.Init;

function register(address toVoter) public {
      if (stage != Stage.Reg) {return;}
....
}

function vote(uint8 toProposal) public  {
      if (stage != Stage.Vote) {return;}
....
}
```

The newer variable added to ballot version 1R, annum for stage, and start time for timekeeping. Advancing the stages, will now depend on the time as specified in the state diagram. Here is a solution ballot version two, with stage and time elements added. This is a traditional solution with the conditions validated by programmatic approach, using if else statement. Now, we are moving to the next question of the ballot. Consider some opportunities for improvement. Validation of time and stage are done inside the function code, programmatically. Because of this, the transaction is executed and recorded on the block chain, irrespective of whether the validation fails or succeeds. Note the if else statement, at the top of the function code of the ballot's smart contract. Further, we ask these questions, is there any way to reject the transaction? In a way similar to how transactions were rejected, at the block chain protocol level, if they don't conform to the rules, that is, if the problem specific conditions are not met toward the transaction. In this case, the transaction will not be recorded on the block chain, wasting effort and space. If there is a way to separate the validation from the actual code that the function executes, is there a way to specify the problem specific rules and condition declaratively, so that they can be independently specified, as well as audited to assure that the smart contract does what it's supposed to do? Auditing of the Smart Contract is specially critical, since a smart contract is expected to be autonomous and permanent, once deployed. We address these issues in the next lesson that is built around problem-specific

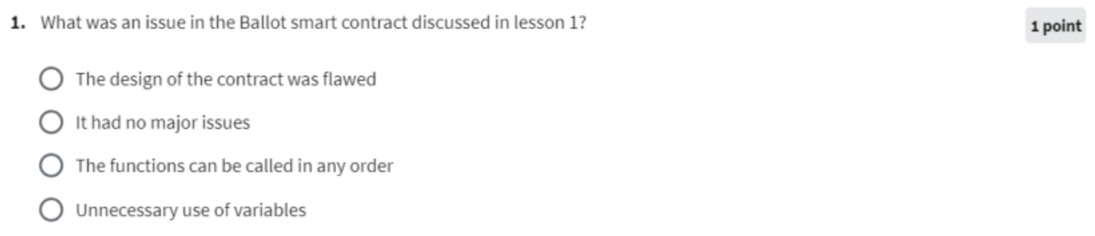

Can we reject transaction if it doesn't conform to rules?

Can we separate the validation from the code that is executed?

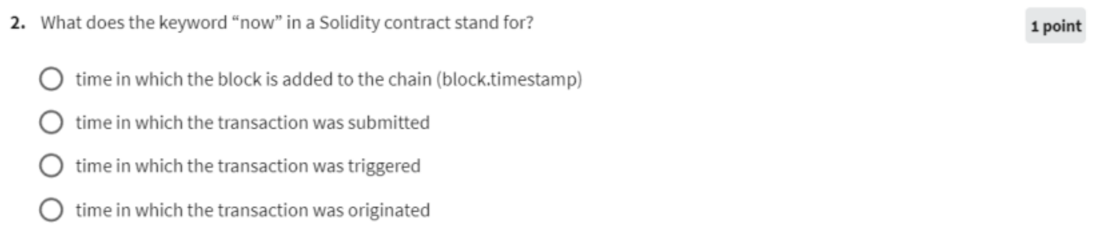
Can we specify the problem-specific rules and conditions so they can be independently specified and audited?

validation, by using function modifiers, required clause, revert and assert declarations.

3.4.1 Practice Quiz


1. What was an issue in the Ballot smart contract discussed in lesson 1? 1 point

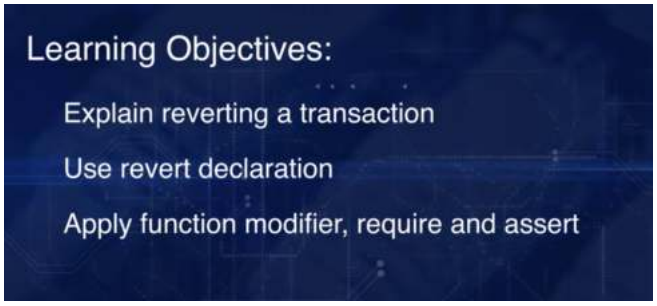
- ◯ The design of the contract was flawed
- ◯ It had no major issues
- ◯ The functions can be called in any order
- ◯ Unnecessary use of variables


2. What does the keyword "now" in a Solidity contract stand for? 1 point

- ◯ time in which the block is added to the chain (block.timestamp)
- ◯ time in which the transaction was submitted
- ◯ time in which the transaction was triggered
- ◯ time in which the transaction was originated

3.5 Validation Test (Part 1) (BallotV3 Demo)

Module three, lesson three: Problem-specific Validation and Testing. The learning objectives of this lesson are: explain the concept of reverting a transaction based on problem-specific validation and using a revert declaration, apply the concepts of function modifier; require; and assert. In this lesson, we will show the execution after



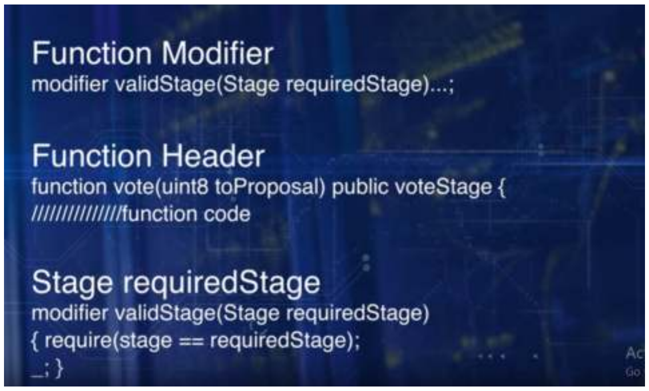
Learning Objectives:

Explain reverting a transaction

Use revert declaration

Apply function modifier, require and assert

modifying ballotversion2.sol using solidity-specific declarations and error handlers. Modifier, require and assert. Solidity features a function revert that results in state-reverting exception. This exception handling will undo all the changes made to the state in the current call and reverses the transaction and also flags an error to the caller. We will introduce a function modifier with the required stage as a parameter. First, we'll add the modifier to the



Function Modifier

```
modifier validStage(Stage requiredStage)...;
```

Function Header

```
function vote(uint8 toProposal) public voteStage {
//////////////function code
```

Stage requiredStage

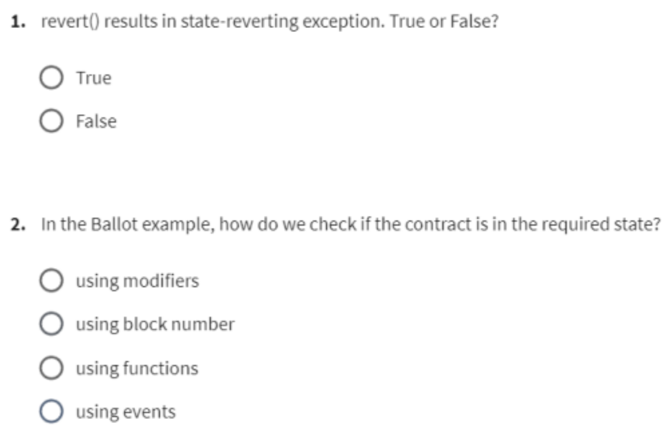
```
modifier validStage(Stage requiredStage)
{ require(stage == requiredStage);
_; }
```

function header. We'll add the modifier with the parameter, stage required stage. Let us review the code with the modifier added. Let us execute the ballot version three with all these improvement. Every now and then when you're working with Remix, it's good to refresh so that you get new account numbers and the space is clean, and so on. I've done that. And so you can see that I've been working on several of them, so they're all open here, and I'm going to choose the version three, ballot version three. And, here, you can see that I have, in the last one, I did not use any modifiers. Here, this one illustrates modifiers require assert, and all the gatekeeper elements that we discussed in the lectures. Okay, so let's go back. This is ballot version three. I have the regular struct for voter, struct for proposal, enum for stage, address for the chairperson, mapping the address to the voters in the proposal array. So, these are all quite familiar to you by now. And you can see how beneficial incremental development is. You take the base contract and put the essentials there and keep adding to it. That helps in better development and a good design. And now you also have the time element and start time, and we see some new things here. I have a single modifier, validStage. Instead of checking the stage inside using if-else statement, which requires the transaction to be executed inside the function, we can prevent that from happening right outside the function using the modifier as you can see here. So, let's look at the modifier. Modifier validStage has a parameters stage, and so every time you want to execute a function, register, vote, or winningProposal, we make sure we are in the right stage to do that based on the elapsed time. Once again, I given a very low elapsed time for the demo purposes. In reality, it's going to be higher. Let's review the code one last time before we go into run. This is the constructor. It has the same name as the ballot, and it requires the number of proposals as a parameter. Chairperson is the sender of this message, and we have designated chairperson voters two that I've been mentioning in the last few demos. And number of proposals is what you were passing in as parameters. For the sake of the demo, we are setting it at three. And more importantly, we are setting the stage to be the registration stage. We are finished with the init stage. Now, we are moving into registration stage and the start time is set as now. And when you go into the register, here, I have here the valid stage modifier specified as one of the modifiers for the function. Function, register, parameters, public is a visibility modifier. ValidStage is a access modifier, custom access modifier, that we defined. And we also have a parameter for that modifier. So at the outset, at the head of the function, we can see clearly, this function will not be executed if validStage is not reg. Okay, that's the beauty of the modifier. And once you go inside, I replace the if statement, the regular programmatic statement with that modifier. So, this was the previous version. Now, we made it at the header so the modifier stops it from going into the function itself. We were checking it inside the function, now at the header of the function, we're checking it. And then, the rest of the things are the regular code, except for the last one, where I'm checking if the elapsed time is over for the registration. If so, I'm moving on to the stage, vote, by setting the stage to the next stage. And the vote function. Here again, I have public, which is a visibility modifier, and I have an access modifier in validStage. As you can see, that was stage registration for registration. This has to be Stage.Vote in order for you to vote. Once again, I'm also keeping the if statement from last version so that you can see what replaced this. This if statement was replaced by the modifier, validStage, Stage.Vote. That is what we defined before. And inside, I have the regular code for counting the votes, and registering the votes, and other things. And at the end, if the elapsed time for the voting phase has moved, I set the stage to the done stage. I also have votingCompleted. This is an event, okay? We'll talk about that later. And I have a winningProposal that also executes only if the stage is done. Otherwise, it doesn't execute. It simply returns false or error so that we know that winning stage has not been arrived at, and we don't have any valid results yet.

3.6 Validation Test (Part 2) (BallotV4 Demo)

Here we see a familiar ballot contract with a few more items added. We have a modifier, and also we have the time and stage elements of that. You're allowed to vote only at certain times and you're allowed to only register at certain times. You can find out the winning proposal only at a certain stage. Okay I'm going to compile and it's already compiled. Run, and it's time dependent. I'm going to go through a little faster. And here I have all the public variables and methods. And I'm going to register just one person. And I just want to make sure that I'm in the stage 2 register. So I'm going to copy that address and you can see that it is being copied. And I have to go back to the base address or the address of the owner of the smart contract. I register, it goes through fine. Now I'm looking at the stage, it is not changed. So I'm going to register one more person, and copy and put it in there.
And I'm going to go back to the chairperson and register. Two people have been registered. So I'm in the stage, and now I can vote. So chairperson votes for 2. And I'm going to go back to one of the addresses and vote for 1. Understand that the chairperson has voted for 2. Now I go back to stage and, okay, so all right. So now I'm going to go back to the third person and vote for 2 again. So we have a lot of votes for 2. We are in stage, it should be 2. Okay, I'm looking at the winning proposal, and it's 2. And we've completed a simple demo. Let me just go back and see whether we can show a revert here. Because we are not in the right stage, so I'm going to now vote for 1. And you can see that it reverted because we are not anymore in the reverting stages, and we are not anymore in the voting stages. I'm going to register it reverted because we are not anymore in the register state. We are only in the final stage of winning proposal. And that is the demo of the ballot revert in case you are not in the right timeframe and you are not in the right stage terminal.
Please do explore this on your own so to understand the various components here.
Observe that we have given the time duration for the registration and voting processes one minutes so that we can complete the testing in a finite time. In reality, these times will be replaced by actual duration of each of the registration and voting activities.
Consider this issue. The base mark contract for the ballot given in solidity documentation will indicate proposal zero as the winning proposal even if no voters or votes are registered. That is by default the proposal that came first, numbered zero, will be chosen as the winning proposal. This is not intended.
How are we going to address this issue? We will address it by using an assert clause at the end of the function winningProposal. Next, we'll look at interfacing with client applications.

3.6.1 Practice Quiz



1. revert() results in state-reverting exception. True or False?

   ○ True

   ○ False

2. In the Ballot example, how do we check if the contract is in the required state?

   ○ using modifiers

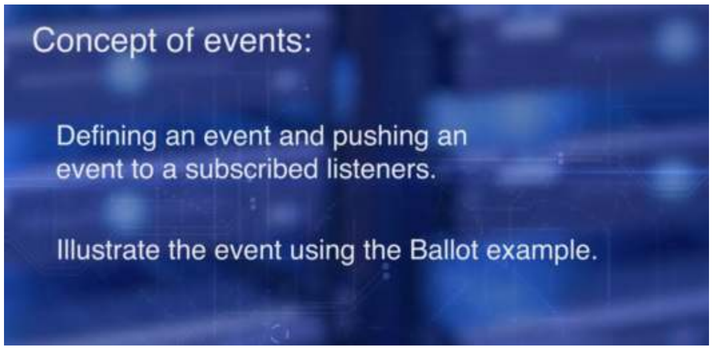
   ○ using block number

   ○ using functions

   ○ using events

3.7 Client Applications

We'll use solidity feature called Events to Interface with Client Application. We'll explain the concept of events: defining an event and pushing an event to a subscribed listener and illustrate the event using the Ballot example. First, a definition of event. A generic format is, event, name of the event, and parameters. For example,

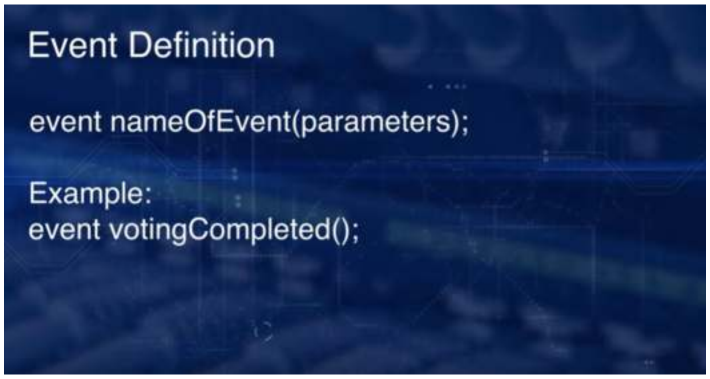


Concept of events:

Defining an event and pushing an event to a subscribed listeners.

Illustrate the event using the Ballot example.

event votingCompleted. Here there are no parameters. Invoking an event is by the name of the event and any parameters. In the function vote, when the state changes to done, we invoke the event. We indicate that by

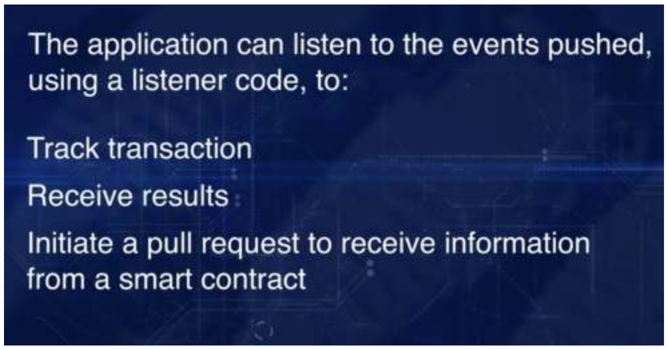


Event Definition

event nameOfEvent(parameters);

Example:
event votingCompleted();

invoking votingCompleted event. In the case of Ballot, we'll push this event at the end of the voting period. There are benefits to event logging. An event is pushed as opposed to regular function call that is pull to get an action performed. Typically, an event feature is to indicate to a client application, user interface or a transaction monitor that a significant milestone has been reached. The application can listen to the events pushed, using a listener code, to track transactions, to receive results through parameters of the event, initiate a pull request to receive information from the smart contract. We'll explore these event handlers in the next course when we



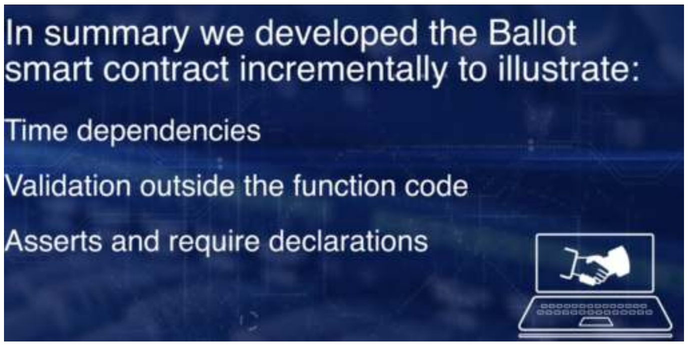
The application can listen to the events pushed, using a listener code, to:

Track transaction

Receive results

Initiate a pull request to receive information from a smart contract

discuss decentralized apps, dApps. Let's now review the entire smart contract with all these features added. In summary, we have developed the Ballot smart contract incrementally to illustrate various features including time dependencies, validation outside the function code before accessing the function using access modifiers, asserts and require declarations, and event logging.



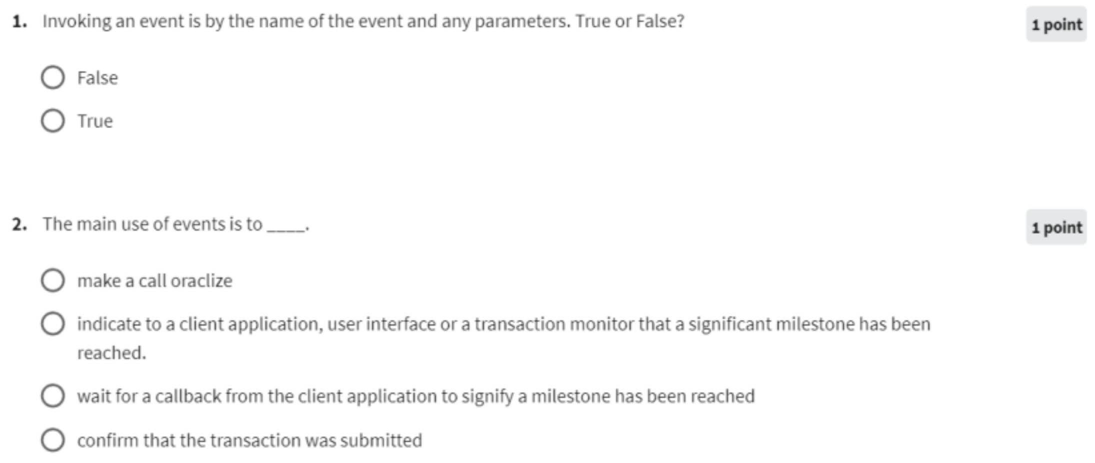
In summary we developed the Ballot smart contract incrementally to illustrate:

Time dependencies

Validation outside the function code

Asserts and require declarations

3.7.1 Practice Quiz



1. Invoking an event is by the name of the event and any parameters. True or False? **1 point**

- ○ False
- ○ True

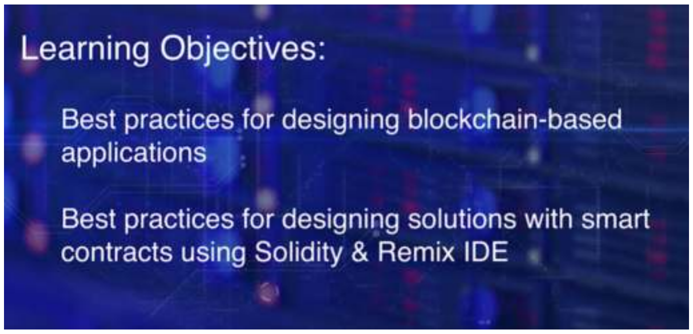
2. The main use of events is to _____. **1 point**

- ○ make a call oraclize
- ○ indicate to a client application, user interface or a transaction monitor that a significant milestone has been reached.
- ○ wait for a callback from the client application to signify a milestone has been reached
- ○ confirm that the transaction was submitted

Chapter 4

4.1 Best Practices: Evaluating Smart Contracts

We have discussed the design and development of a smart contract throughout the course thus far. This module will focus on the best practices. We'll begin with the cautionary note about evaluating whether a blockchain-based solution is suitable for your problem. We'll then discuss some of the best practices when designing Solidity smart contract focusing on data functions and their visibility modifiers and access modifiers. We will follow it up with best practices as it relates to Remix IDE. Upon completion of this module, you will be able to list the best practices when designing blockchain-based application. Illustrate the best practices for designing solutions with smart contract using Solidity and Remix IDE. Blockchain is not a solution for all applications. Make sure



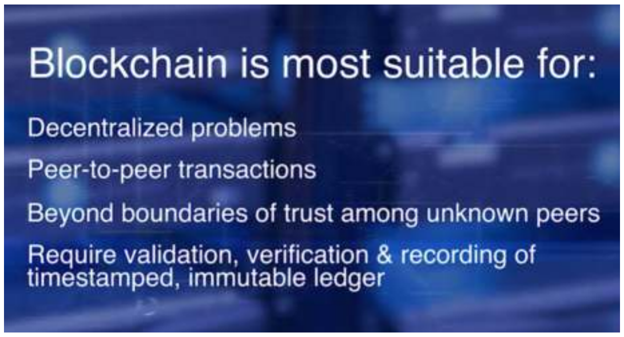
Learning Objectives:

Best practices for designing blockchain-based applications

Best practices for designing solutions with smart contracts using Solidity & Remix IDE

your application requirements need blockchain features. In other words, blockchain-based solutions and smart contracts are not a panacea for all problems you have. Then, what is it good for? Recall that we learned in course one that blockchain solution is most suitable for applications with these characteristics: Decentralized problems, meaning participant hold the assets and are not co-located.

In more, peer-to-peer transaction without intermediaries. Operate beyond the boundaries of trust among unknown peers. Require validation, verification, and recording on a universally timestamped, immutable ledger. Autonomous operations guided by rules and policies. Make sure you need a smart contract on blockchain for your



Blockchain is most suitable for:

Decentralized problems

Peer-to-peer transactions

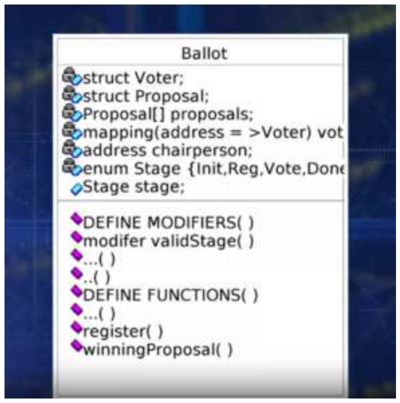
Beyond boundaries of trust among unknown peers

Require validation, verification & recording of timestamped, immutable ledger

application. Understand that smart contracts will be visible to all participants on the chain and will be executed on all the full nodes. You need a smart contract when you need a collective agreement based on rules, regulation, policies, or governance enforced, and the decision and the provenance for it must be recorded. Smart contract is not for single node computation. It does not replace your client server or inherently stateless distributed solutions. Keep the smart contract code simple, coherent, and auditable. Let it solve a single problem well to avoid design and coding errors. In other words, let the state variables and the functions specified in a smart contract be addressing a single problem. Do not include redundant data or unrelated functions. Make the smart contract functions auditable by using custom function modifiers instead of inline if-else code for checking pre and post conditions for a function execution. Smart contracts are usually part of a large distributed application; the part that requires the services provided by the blockchain. Blockchain is not a data repository. Keep only the necessary data in the smart contract. Of course, it is an immutable distributed ledger of transactions. Recall that the blockchain also manages the state transitions and maintains the state hash, transaction hash, and receipt hash in the header of each block. These are indeed application dependent overheads. Given these characteristics, it's a good practice to analyze the application data and separate them into on-chain an off-chain data. Design the state variables for the smart contract to be efficient storage for the on-chain data. Leave the off-chain data to be managed by higher level applications. As an example, instead of keeping and enter a 500 pages of a public legal document on the chain, keep only the metadata about the document, including a secure hash to protect the integrity of the document.

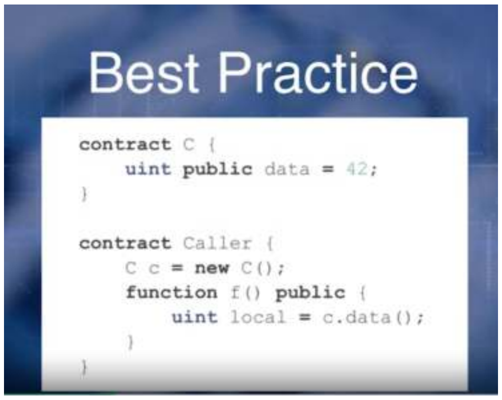
4.2 Designing Smart Contracts

Let's continue our discussion on best practices. Use appropriate data types, understand that Ethereum Virtual Machine, is a two 56-bit processor optimized for integer computations. It has a stack machine for execution with limited set of appcodes. Consider the ballot example discussed in the last module. For example, a variable proposal name of string type can be expensive, since a string in a smart contract is a dynamic sized variable. So, how can we address this issue? Instead of using a variable string name for the proposals, use an integer



identification for the proposal. This ID is used as an index into the array of proposals. This is the number returned by the function winning proposal. A user-level application outside the chain can then map the ID to proposal name if so desired. Make sure you use the integer arithmetic for most of your computational needs. 256-bit processor for the EVM, is indeed very large in comparison to your regular 64-bit processor, four times the sheer number of bits. With every doubling of the number of bits of the container size the range of values increases exponentially. Solidity also provides different sizes ranging from eight bits Uint8 all the way to 256 bits Uint256. Understand the public visibility modifier for data. All state variables are created as private. Any variable on the block chain is viewable to all, irrespective of the visibility modifier. You have to explicitly state that a variable is public. When a variable is declared public, Solidity compiler automatically creates a getter method to view the value of the variable.

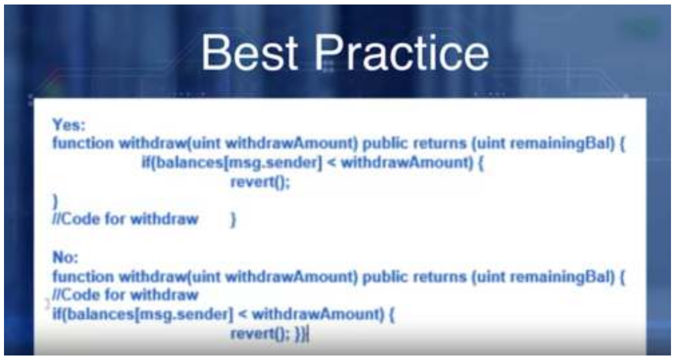
Internal to the contract, the variable is accessed as data, externally it is accessed as a function. Be aware of this difference in accessing the public data and the getter method. Maintain a standard order for different function types within a smart contract, according to their visibility as specified in Solidity docs. The recommended order for functions within a smart contract are; constructor, fallback function, external, public, internal, private. Within a grouping, plays the constant functions last. Functions can have many different modifiers, functions have visibility modifiers as well as predefined and custom access modifiers. The visibility modifiers for the function should come before any custom access modifiers. Multiple modifiers that apply to a function, by specifying them in a white space separated list and are evaluated in the order presented. Hence, if the output of one modifier



Best Practice

```
contract C {
    uint public data = 42;
}

contract Caller {
    C c = new C();
    function f() public {
        uint local = c.data();
    }
}
```

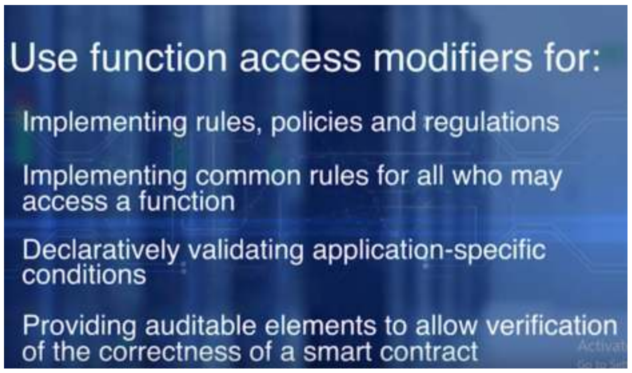
depends on the other, make sure you order them in the right sequence. For example, function buy, has three modifiers specified in the following order; payable, enoughMoney, item available. Use Solidity-defined payable modifier when sending value. Only through payable functions, can an account send value to another address. Payable is a reserved keyword, you may use payable as an addition to an existing function. In the following example, the bid function is to bid for an auction item, the transaction bid invoking bid can send either B only if the function is payable. In other words, deposit, register, and bid functions are allowed to send money to the target smart contract address. Pay attention to the order of statements within a function. The first withdraw function checks the condition first and then allows for withdraw, the second one has a withdraw code first and then condition check. Something similar to this, resulted in parody wallet losing several million ethers, when a smart contract was killed before moving the ethers.



Best Practice

```
Yes:
function withdraw(uint withdrawAmount) public returns (uint remainingBal) {
                if(balances[msg.sender] < withdrawAmount) {
                                revert();
}
//Code for withdraw      }

No:
function withdraw(uint withdrawAmount) public returns (uint remainingBal) {
//Code for withdraw
if(balances[msg.sender] < withdrawAmount) {
                                revert(); }}
```

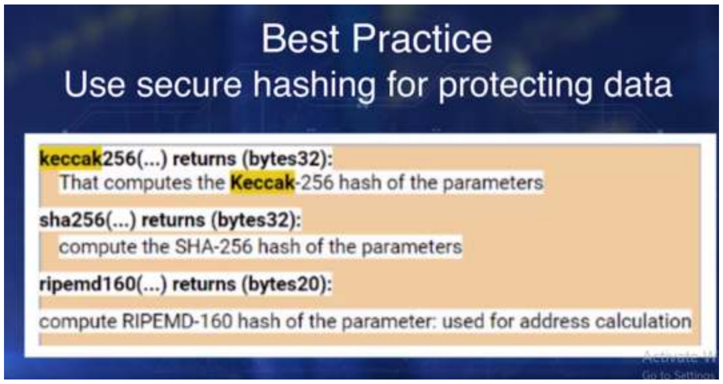
Use modifier declarations for implementing rules. Use function access modifiers for; implementing rules, policies and regulations. Implementing common rules for all who may access a function, declaratively validating application specific conditions and providing auditable elements to allow verification of the correctness of a smart contract.



Use function access modifiers for:

- Implementing rules, policies and regulations
- Implementing common rules for all who may access a function
- Declaratively validating application-specific conditions
- Providing auditable elements to allow verification of the correctness of a smart contract

Using events in smart contract. Use events to log important milestones, during the span of a smart contract execution especially long running ones. Events can carry at most three index parameters that can be used efficiently for searching through the events in the block chain. Instead of a client application, polling a smart contract using external functions, events can push information to the application that would have set up listeners to specific events. Beware of "now" time variable. Now is the alias for block timestamp of the block indicating the universal time when the block and the transaction within it are mined and recorded. Variable now can be used for approximate elapsed time comparison, as we illustrated in the ballot example. However, it is not a good practice to use it for computation within the application logic. Also, variable now provides time to the accuracy of seconds if fine granular time is required, you may have to resort to other mechanisms. Use secure hashing for protecting data. Recall that hashing is a very important function in a block chain. Data in the block chain is viewable by all. This means that we may want to secure hash, to protect its visibility. Solidity provides a variety of built-in functions for standard secure hash functions. Keccak, SHA-256, RIPEMD-160 are Solidity functions available to use for hashing application data.



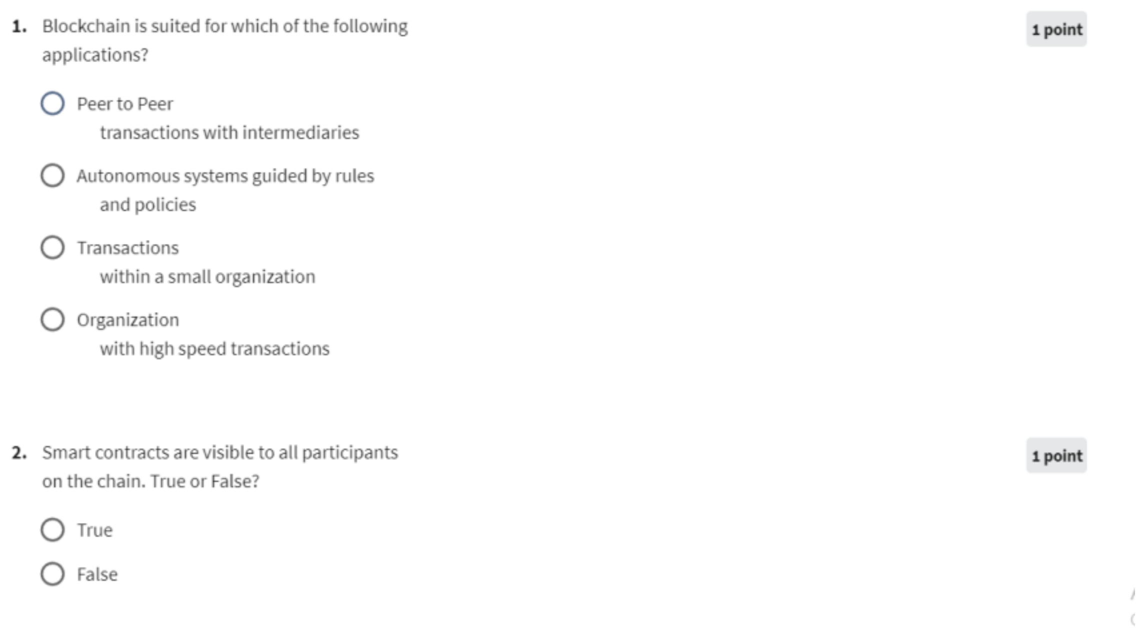
Best Practice
Use secure hashing for protecting data

keccak256(...) returns (bytes32):
That computes the Keccak-256 hash of the parameters
sha256(...) returns (bytes32):
compute the SHA-256 hash of the parameters
ripemd160(...) returns (bytes20):
compute RIPEMD-160 hash of the parameter: used for address calculation

4.3 Remix Web IDE

Starting with this best practice, the next few UrbanRemix environment. Remix keep changing quite frequently with weekly update. But it does provide a single web ID for all your development needs. Pay attention to remix static analysis. When compiling on remix, pay attention to static analysis warning it provides on the right panel. Many of these are quite useful for developing better smart contracts. For example, one of the warnings it offers is about time variable. Now, revisit the ballad example and check out the static analysis on variable now provided by remix environment. Pay attention to remix console detail. We began this course with a little history about smart contracts. We then discuss the basic structure of a smart contract, the development environment remix and the artifacts generated by the compiler process. We then illustrated the smart contract development and the solidity language with several smart contract greeter, bidder, mentor, and several versions of the ballot. We developed several versions of the ballot contract, adding features in incremental steps. We concluded the course by listing some best practices, and design, and development of smart contract. Now, here is your opportunity to apply all of the concepts learned in this course to proto-type blockchain solutions. Let's go ahead and do it. Review remix compile details. Remix compiler is a just-in-time compiler. As you're entering the code in the editor window, its syntax checks and flags any errors with a red button X. Click on it to get more details on the error. This is useful information that helps you learn the syntax as you go, and you're able to make corrections. After compile, when you click on the details button, you can see all the artifacts generated by the compiler process. Remix transaction log for debugging. The remix panel has a floppy disk icon. When you click on this, you can see the JSon version of all the executed transaction. This Json file can be saved and used for studying the transaction details, and also for debugging any run-time issues and logic problems with a smart contract code. To summarize, we reviewed a few best practices for different aspects related to smart contract. Of course, this list does not cover all the best practices. We'll collect and add some more to this list as we progress through the specialization. I'm glad that you joined me in this exploration of smart contracts. I encourage you to use the open discussion forum to share your experience with the design, development, testing, and execution of a smart contract on a blockchain.

4.3.1 Practice Quiz



1. Blockchain is suited for which of the following applications? **1 point**

- ◯ Peer to Peer transactions with intermediaries
- ◯ Autonomous systems guided by rules and policies
- ◯ Transactions within a small organization
- ◯ Organization with high speed transactions

2. Smart contracts are visible to all participants on the chain. True or False? **1 point**

- ◯ True
- ◯ False