TABLE OF CONTENTS

MEHMOOD SHEIKH

MEHMOOD SHEIKH

SECTION 1: CREATING SPACE DOGGO

VARIABLE TYPES

State variables are used to store information on the blockchain. They can also be manipulated by the functions within the contract.

Solidity is a statically typed programming language, meaning that each variable must have its type specified. Examples of the main data types:

Booleans

Can only have one of the two following values: true or false. The keyword for booleans is bool.

Integers

Integers can be split into main groups: regular integers (can store both positive and negative values) and unsigned integers (can only store values that are 0 or higher)

Regular integer has keywords from int8 to int256. The number signifies the maximum number of bits it can store (thus limiting the maximum value), and it can be any number between 8 and 256 as long as it incremented in steps of 8 (e.g., int16 is valid but int17 is not). The int keyword alone would be understood as int256.

Unsigned integers follow the same logic. The only difference is the keyword which ranges from uint8 to uint256. Keyword uint can also be used instead of uint256.

Addresses

The address keyword is used to hold Ethereum addresses. If you're planning to store an Ethereum address, you will need to use the address keyword.

Strings

The string variable is used to store text information. If you need to store a variable that should contain text information use the string keyword.

MEHMOOD SHEIKH

ARITHMETIC OPERATIONS

Integers can be used for arithmetic operations:

Addition x + y

Subtraction x - y

Multiplication x * y

Division x / y

Remainder x % y

Exponentiation x ** y (x to the power of y)

STRUCTS

A struct is a special data type that allows the programmer to group a list of variables.

Structures are defined like this:

```
struct Car {
    string make;
    string model;
    uint16 year;
    uint16 horsepower;
}
```

MAPPINGS

Now that you've defined your Doggo, we'll learn how to create a variable that will later map your Doggo to blockchain - a special list where all other Doggos are stored.

Mappings

Mappings allow the programmer to create key-value pairs and store them as a list. Concepts like this also are known as hash tables.

Mappings are defined like this:

MEHMOOD SHEIKH

mapping(key_type => key_value) mappingName;

key_type should be replaced by a data type. Two commonly used variable types for mapping keys that we already know about are address and uint. It is important to note that not every data type can be used as a key. For instance, structs and other mappings cannot be used as keys.

Similarly, key_value should be replaced by the value type. Unlike with keys, Solidity does not limit the data type for values. It can be anything, including structs and other mappings.

A real-world example of a mapping:

```
mapping(address => uint256) balance;
```

This mapping could hold the bank account balance in uint256 for the given address.

FUNCTION DECLARATIONS

Functions are pieces of code that perform a specific task. Functions can take any number of variables as an input, and it can return a specific value back. In case of Solidity, you can even return multiple values from one function.

Here is a simple function that does not return any data but simply adds two values together and assigns the result to a state variable sum:

```
uint sum;

function add(uint a, uint b) {

    sum = a + b;

}
```

Function inputs are optional, but if there are any, they need to be comma-separated

If we want to return data and avoid changing the contract's state, we can rewrite the function like this:

```
function add(uint a, uint b) returns(uint)  {

    return a + b;

}
```

MEHMOOD SHEIKH

Note: that a new keyword returns() was used to specify the type of the function output. Because we added two uints together, the result will also be uint. To return a value back, you need to use the return statement.

CREATING STRUCTS

Structs are created by specifying the values for each struct property.

```
struct Car {
    string make;
    string model;
    uint16 year;
}
function makeCar(uint16 _year) {
    Car myCar = Car({
        make: "Nissan",
        model: "Micra",
        year: _year
    });
}
```

Please note how you can assign both constant values and values from variables. In this case, you can see how model value is a constant and year value comes from function argument _year.

STORAGE TYPES

Solidity has two places where it can store variable data: storage (which is the blockchain itself) or memory (a temporary place which is erased when it is no longer needed). Storage is expensive to use but memory is not. By default, structs and arrays reference to storage and all other variables use memory. If you want to make sure that a specific variable (a struct, for example) is stored in memory, you need to use the memory keyword. Like this:

```
Pizza memory favoritePizza = Pizza({type: "pepperoni"});
```

MEHMOOD SHEIKH

USING MAPPINGS

Now that you've created your Doggo and mapped it to the Doggos list, we will save your Doggo onto a list that's located within the blockchain.

Assigning values to a mapping

Attaching values to a mapping is pretty straight forward. The syntax is very similar to regular value assignment, the only difference is that you need to specify the key. It is done like this:

```
mapping(address => uint) score;
function updatePoints(address _person, uint _points) {
    score[_person] = _points;
}
```

SENDER ADDRESS

The functions on Smart Contracts (unless they are internal or private - more on that in upcoming lessons) will be called directly by accounts on Ethereum blockchain. Each account is a hexadecimal address that looks something like this:

```
0x1961b3331969ed52770751fc718ef530838b6dee
```

To retrieve the function caller's address, you can use special variable msg.sender.

For example:

```
score[msg.sender] = 500;
```

HASHES

To generate the space and planets we will need to turn two numbers that correspond to X and Y coordinates to a seemingly random code that we could use to determine the traits of the solar system and the planets that belong to it.

MEHMOOD SHEIKH

For this purpose, we will use Ethereum's KECCAK-256 algorithm that produces a 256-bit hexadecimal number. In Solidity, you can use the keccak256 function to get the hash. You can pass any number of arguments to this function.

For example:

```
keccak256(123, 321);
```

TYPECASTING

Sometimes you may need to do arithmetics with two variables that have different data types. Solidity will only allow operations between two variables that have the same type. Therefore you need to convert one of the variables to the type of the others. For example:

```
uint8 x = 8;

uint256 y = 10 ** 18;

uint256 z = y * uint256(x);
```

As you can see, we converted variable x from uint8 to uint256 by writing uint256(x).

SPACE DOGGO-CODE:

```
pragma solidity ^0.4.20;


contract SpaceDoggos {

    uint maxPlanetsPerSystem = 10;
    uint minPlanetsPerSystem = 3;


    uint planetCodeDigits = 7;
    uint systemCodeDigits = 7;
```

MEHMOOD SHEIKH

```
    uint planetCodeModulus = 10 ** planetCodeDigits;
    uint systemCodeModulus = 10 ** systemCodeDigits;

    struct Doggo {
        string name;
        uint8 breed;
        uint8 color;
        uint8 face;
        uint8 costume;
        uint coordX;
        uint coordY;
    }

    mapping(address => Doggo) doggos;

    function createDoggo(string _name, uint8 _breed, uint8 _color, uint8 _face,
uint8 _costume) {
        Doggo memory myDoggo = Doggo({
            name: _name,
            breed: _breed,
            color: _color,
            face: _face,
            costume: _costume,
            coordX: 0,
```

MEHMOOD SHEIKH

```
            coordY: 0
        });

        doggos[msg.sender] = myDoggo;
    }
    function getSystemMap(uint _coordX, uint _coordY) returns (uint) {
        return uint(keccak256(_coordX, _coordY));
    }
}
```

MEHMOOD SHEIKH

SECTION 2: CRYPTO ZOMBIE

MAKING THE ZOMBIE FACTORY

STATE VARIABLES & INTEGERS

State variables are permanently stored in contract storage. This means they're written to the Ethereum blockchain. Think of them like writing to a DB.

Example:

```
contract Example {

  // This will be stored permanently in the blockchain

  uint myUnsignedInteger = 100;

}
```

In this example contract, we created a uint called myUnsignedInteger and set it equal to 100.

Unsigned Integers: uint

The uint data type is an unsigned integer, meaning its value must be non-negative. There's also an int data type for signed integers.

Note: In Solidity, uint is actually an alias for uint256, a 256-bit unsigned integer. You can declare uints with less bits — uint8, uint16, uint32, etc.. But in general you want to simply use uint except in specific cases, which we'll talk about in later lessons.

MATH OPERATIONS

Math in Solidity is pretty straightforward. The following operations are the same as in most programming languages:

Addition: x + y

Subtraction: x - y,

Multiplication: x * y

MEHMOOD SHEIKH

Division: x / y

Modulus / remainder: x % y (for example, 13 % 5 is 3, because if you divide 5 into 13, 3 is the remainder)

Exponent: x** y

STRUCTS

Sometimes you need a more complex data type. For this, Solidity provides structs:

```
struct Person {
  uint age;
  string name;
}
```

Structs allow you to create more complicated data types that have multiple properties.

Note that we just introduced a new type, string. Strings are used for arbitrary-length UTF-8 data. Ex. string greeting = "Hello world!"

ARRAYS

When you want a collection of something, you can use an array. There are two types of arrays in Solidity: fixed arrays and dynamic arrays:

```
// Array with a fixed length of 2 elements:
uint[2] fixedArray;
// another fixed Array, can contain 5 strings:
string[5] stringArray;
// a dynamic Array - has no fixed size, can keep growing:
uint[] dynamicArray;
```

You can also create an array of structs. Using the previous chapter's Person struct:

```
Person[] people; // dynamic Array, we can keep adding to it
```

MEHMOOD SHEIKH

Remember that state variables are stored permanently in the blockchain? So creating a dynamic array of structs like this can be useful for storing structured data in your contract, kind of like a database.

PUBLIC ARRAYS

You can declare an array as public, and Solidity will automatically create a getter method for it. The syntax looks like:

```
Person[] public people;
```

Other contracts would then be able to read from, but not write to, this array. So this is a useful pattern for storing public data in your contract.

FUNCTION DECLARATIONS

A function declaration in solidity looks like the following:

```
function eatHamburgers(string memory _name, uint _amount) public {

}
```

This is a function named eatHamburgers that takes 2 parameters: a string and a uint. For now the body of the function is empty. Note that we're specifying the function visibility as public. We're also providing instructions about where the _name variable should be stored- in memory. This is required for all reference types such as arrays, structs, mappings, and strings.

What is a reference type you ask?

Well, there are two ways in which you can pass an argument to a Solidity function:

By **value**, which means that the Solidity compiler creates a new copy of the parameter's value and passes it to your function. This allows your function to modify the value without worrying that the value of the initial parameter gets changed.

By **reference**, which means that your function is called with a... reference to the original variable. Thus, if your function changes the value of the variable it receives, the value of the original variable gets changed.

Note: It's convention (but not required) to start function parameter variable names with an underscore (_) in order to differentiate them from global variables. We'll use that convention throughout our tutorial.

MEHMOOD SHEIKH

You would call this function like so:

```
eatHamburgers("vitalik", 100);
```

WORKING WITH STRUCTS AND ARRAYS

Creating New Structs

Remember our Person struct in the previous example?

```
struct Person {
  uint age;
  string name;
}
Person[] public people;
```

Now we're going to learn how to create new Persons and add them to our people array.

```
// create a New Person:
Person satoshi = Person(172, "Satoshi");
// Add that person to the Array:
people.push(satoshi);
```

We can also combine these together and do them in one line of code to keep things clean:

```
people.push(Person(16, "Vitalik"));
```

Note: that array.push() adds something to the end of the array, so the elements are in the order we added them. See the following example:

```
uint[] numbers;
numbers.push(5);
numbers.push(10);
numbers.push(15);
// numbers is now equal to [5, 10, 15]
```

MEHMOOD SHEIKH

PRIVATE / PUBLIC FUNCTIONS

In Solidity, functions are public by default. This means anyone (or any other contract) can call your contract's function and execute its code.

Obviously this isn't always desirable, and can make your contract vulnerable to attacks. Thus it's good practice to mark your functions as private by default, and then only make public the functions you want to expose to the world.

Let's look at how to declare a private function:

```
uint[] numbers;

function _addToArray(uint _number) private {

  numbers.push(_number);

}
```

This means only other functions within our contract will be able to call this function and add to the numbers array.

As you can see, we use the keyword private after the function name. And as with function parameters, it's convention to start private function names with an underscore (_).

MORE ON FUNCTIONS

In this chapter, we're going to learn about function return values, and function modifiers.

Return Values

To return a value from a function, the declaration looks like this:

```
string greeting = "What's up dog";

function sayHello() public returns (string memory) {

  return greeting;

}
```

In Solidity, the function declaration contains the type of the return value (in this case string).

MEHMOOD SHEIKH

Function modifiers

The above function doesn't actually change state in Solidity — e.g. it doesn't change any values or write anything.

So in this case we could declare it as a view function, meaning it's only viewing the data but not modifying it:

```
function sayHello() public view returns (string memory) {}
```

Solidity also contains pure functions, which means you're not even accessing any data in the app. Consider the following:

```
function _multiply(uint a, uint b) private pure returns (uint) {
  return a * b;
}
```

This function doesn't even read from the state of the app — its return value depends only on its function parameters. So in this case we would declare the function as pure.

Note: It may be hard to remember when to mark functions as pure/view. Luckily the Solidity compiler is good about issuing warnings to let you know when you should use one of these modifiers

KECCAK256 AND TYPECASTING

We want our _generateRandomDna function to return a (semi) random uint. How can we accomplish this?

keccak256

Ethereum has the hash function keccak256 built in, which is a version of SHA3. A hash function basically maps an input into a random 256-bit hexadecimal number. A slight change in the input will cause a large change in the hash.

It's useful for many purposes in Ethereum, but for right now we're just going to use it for pseudo-random number generation.

Also important, keccak256 expects a single parameter of type bytes. This means that we have to "pack" any parameters before calling keccak256:

Example:

MEHMOOD SHEIKH

```
//6e91ec6b618bb462a4a6ee5aa2cb0e9cf30f7a052bb467b0ba58b8748c00d2e5

keccak256(abi.encodePacked("aaaab"));

//b1f078126895a1424524de5321b339ab00408010b7cf0e6ed451514981e58aa9
```

keccak256(abi.encodePacked("aaaac"));

As you can see, the returned values are totally different despite only a 1 character change in the input.

Note: Secure random-number generation in blockchain is a very difficult problem. Our method here is insecure, but since security isn't top priority for our Zombie DNA, it will be good enough for our purposes.

Typecasting

Sometimes you need to convert between data types. Take the following example:

```
uint8 a = 5;

uint b = 6;

// throws an error because a * b returns a uint, not uint8:

uint8 c = a * b;

// we have to typecast b as a uint8 to make it work:

uint8 c = a * uint8(b);
```

In the above, a * b returns a uint, but we were trying to store it as a uint8, which could cause potential problems. By casting it as a uint8, it works and the compiler won't throw an error.

EVENTS

Our contract is almost finished! Now let's add an **event**.

Events are a way for your contract to communicate that something happened on the blockchain to your app front-end, which can be 'listening' for certain events and take action when they happen.

Example:

```
// declare the event

event IntegersAdded(uint x, uint y, uint result);
```

MEHMOOD SHEIKH

```
function add(uint _x, uint _y) public returns (uint) {
  uint result = _x + _y;
  // fire an event to let the app know the function was called:
  emit IntegersAdded(_x, _y, result);
  return result;
}
```

Your app front-end could then listen for the event. A JavaScript implementation would look something like:

```
YourContract.IntegersAdded(function(error, result) {
  // do something with result
})
```

WEB3.JS

Our Solidity contract is complete! Now we need to write a JavaScript frontend that interacts with the contract.

Ethereum has a JavaScript library called **Web3.js**.

In a later lesson, we'll go over in depth how to deploy a contract and set up Web3.js. But for now let's just look at some sample code for how Web3.js would interact with our deployed contract.

Don't worry if this doesn't all make sense yet.

```
// Here's how we would access our contract:
var abi = /* abi generated by the compiler */
var ZombieFactoryContract = web3.eth.contract(abi)
var contractAddress = /* our contract address on Ethereum after deploying */
var ZombieFactory = ZombieFactoryContract.at(contractAddress)
// `ZombieFactory` has access to our contract's public functions and events

// some sort of event listener to take the text input:
$("#ourButton").click(function(e) {
  var name = $("#nameInput").val()
  // Call our contract's `createRandomZombie` function:
  ZombieFactory.createRandomZombie(name)
```

MEHMOOD SHEIKH

```
})

// Listen for the `NewZombie` event, and update the UI
var event = ZombieFactory.NewZombie(function(error, result) {
  if (error) return
  generateZombie(result.zombieId, result.name, result.dna)
})

// take the Zombie dna, and update our image
function generateZombie(id, name, dna) {
  let dnaStr = String(dna)
  // pad DNA with leading zeroes if it's less than 16 characters
  while (dnaStr.length < 16)
    dnaStr = "0" + dnaStr

  let zombieDetails = {
    // first 2 digits make up the head. We have 7 possible heads, so % 7
    // to get a number 0 - 6, then add 1 to make it 1 - 7. Then we have 7
    // image files named "head1.png" through "head7.png" we load based on
    // this number:
    headChoice: dnaStr.substring(0, 2) % 7 + 1,
    // 2nd 2 digits make up the eyes, 11 variations:
    eyeChoice: dnaStr.substring(2, 4) % 11 + 1,
    // 6 variations of shirts:
    shirtChoice: dnaStr.substring(4, 6) % 6 + 1,
    // last 6 digits control color. Updated using CSS filter: hue-rotate
    // which has 360 degrees:
    skinColorChoice: parseInt(dnaStr.substring(6, 8) / 100 * 360),
    eyeColorChoice: parseInt(dnaStr.substring(8, 10) / 100 * 360),
    clothesColorChoice: parseInt(dnaStr.substring(10, 12) / 100 * 360),
    zombieName: name,
```

MEHMOOD SHEIKH

```
    zombieDescription: "A Level 1 CryptoZombie",
  }
  return zombieDetails
}
```

ZOMBIES ATTACK THEIR VICTIMS

MAPPINGS AND ADDRESSES

ADDRESSES

The Ethereum blockchain is made up of accounts, which you can think of like bank accounts. An account has a balance of Ether (the currency used on the Ethereum blockchain), and you can send and receive Ether payments to other accounts, just like your bank account can wire transfer money to other bank accounts.

Each account has an address, which you can think of like a bank account number. It's a unique identifier that points to that account, and it looks like this:

0x0cE446255506E92DF41614C46F1d6df9Cc969183

We'll get into the nitty gritty of addresses in a later lesson, but for now you only need to understand that an address is owned by a specific user (or a smart contract).

So we can use it as a unique ID for ownership of our zombies. When a user creates new zombies by interacting with our app, we'll set ownership of those zombies to the Ethereum address that called the function.

MAPPINGS

Defining a mapping looks like this:

```
// For a financial app, storing a uint that holds the user's account balance:
mapping (address => uint) public accountBalance;
// Or could be used to store / lookup usernames based on userId
mapping (uint => string) userIdToName;
```

MEHMOOD SHEIKH

A mapping is essentially a key-value store for storing and looking up data. In the first example, the key is an address and the value is a uint, and in the second example the key is a uint and the value a string.

MSG.SENDER

Now that we have our mappings to keep track of who owns a zombie, we'll want to update the _createZombie method to use them.

In order to do this, we need to use something called msg.sender.

msg.sender

In Solidity, there are certain global variables that are available to all functions. One of these is msg.sender, which refers to the address of the person (or smart contract) who called the current function.

Note: In Solidity, function execution always needs to start with an external caller. A contract will just sit on the blockchain doing nothing until someone calls one of its functions. So there will always be a msg.sender.

Here's an example of using msg.sender and updating a mapping:

```
mapping (address => uint) favoriteNumber;

function setMyNumber(uint _myNumber) public {
  // Update our `favoriteNumber` mapping to store `_myNumber` under `msg.sender`
  favoriteNumber[msg.sender] = _myNumber;
  // ^ The syntax for storing data in a mapping is just like with arrays
}

function whatIsMyNumber() public view returns (uint) {
  // Retrieve the value stored in the sender's address
  // Will be `0` if the sender hasn't called `setMyNumber` yet
  return favoriteNumber[msg.sender];
}
```

In this trivial example, anyone could call setMyNumber and store a uint in our contract, which would be tied to their address. Then when they called whatIsMyNumber, they would be returned the uint that they stored.

MEHMOOD SHEIKH

Using msg.sender gives you the security of the Ethereum blockchain — the only way someone can modify someone else's data would be to steal the private key associated with their Ethereum address.

REQUIRE

How can we make it so this function can only be called once per player?

For that we use require. require makes it so that the function will throw an error and stop executing if some condition is not true:

```
function sayHiToVitalik(string memory _name) public returns (string memory) {
  // Compares if _name equals "Vitalik". Throws an error and exits if not true.
  // (Side note: Solidity doesn't have native string comparison, so we
  // compare their keccak256 hashes to see if the strings are equal)
  require(keccak256(abi.encodePacked(_name)) ==
keccak256(abi.encodePacked("Vitalik")));
  // If it's true, proceed with the function:
  return "Hi!";
}
```

If you call this function with sayHiToVitalik("Vitalik"), it will return "Hi!". If you call it with any other input, it will throw an error and not execute.

Thus require is quite useful for verifying certain conditions that must be true before running a function.

INHERITANCE

One feature of Solidity that makes this more manageable is contract **inheritance**:

```
contract Doge {
  function catchphrase() public returns (string memory) {
    return "So Wow CryptoDoge";
  }
}
```

MEHMOOD SHEIKH

BabyDoge **inherits** from Doge. That means if you compile and deploy BabyDoge, it will have access to both catchphrase() and anotherCatchphrase() (and any other public functions we may define on Doge).

This can be used for logical inheritance (such as with a subclass, a Cat is an Animal). But it can also be used simply for organizing your code by grouping similar logic together into different contracts.

IMPORT

When you have multiple files and you want to import one file into another, Solidity uses the import keyword:

```
import "./someothercontract.sol";

contract newContract is SomeOtherContract {

}
```

So if we had a file named someothercontract.sol in the same directory as this contract (that's what the ./ means), it would get imported by the compiler.

STORAGE VS MEMORY (DATA LOCATION)

In Solidity, there are two locations you can store variables — in storage and in memory.

Storage refers to variables stored permanently on the blockchain. **Memory** variables are temporary, and are erased between external function calls to your contract. Think of it like your computer's hard disk vs RAM.

Most of the time you don't need to use these keywords because Solidity handles them by default. State variables (variables declared outside of functions) are by default storage and written permanently to the blockchain, while variables declared inside functions are memory and will disappear when the function call ends.

However, there are times when you do need to use these keywords, namely when dealing with **structs** and **arrays** within functions:

```
contract SandwichFactory {

  struct Sandwich {

    string name;

    string status;
```

MEHMOOD SHEIKH

```
  }

  Sandwich[] sandwiches;

  function eatSandwich(uint _index) public {
    // Sandwich mySandwich = sandwiches[_index];
    // ^ Seems pretty straightforward, but solidity will give you a warning
    // telling you that you should explicitly declare `storage` or `memory` here.
    // So instead, you should declare with the `storage` keyword, like:
    Sandwich storage mySandwich = sandwiches[_index];
    // ...in which case `mySandwich` is a pointer to `sandwiches[_index]`
    // in storage, and...
    mySandwich.status = "Eaten!";
    // ...this will permanently change `sandwiches[_index]` on the blockchain.
    // If you just want a copy, you can use `memory`:
    Sandwich memory anotherSandwich = sandwiches[_index + 1];
    // ...in which case `anotherSandwich` will simply be a copy of the
    // data in memory, and...
    anotherSandwich.status = "Eaten!";
    // ...will just modify the temporary variable and have no effect
    // on `sandwiches[_index + 1]`. But you can do this:
    sandwiches[_index + 1] = anotherSandwich;
    // ...if you want to copy the changes back into blockchain storage.
  }
}
```

MEHMOOD SHEIKH

Don't worry if you don't fully understand when to use which one yet — throughout this tutorial we'll tell you when to use storage and when to use memory, and the Solidity compiler will also give you warnings to let you know when you should be using one of these keywords.

For now, it's enough to understand that there are cases where you'll need to explicitly declare storage or memory!

INTERNAL AND EXTERNAL

In addition to public and private, Solidity has two more types of visibility for functions: internal and external.

Internal

internal is the same as private, except that it's also accessible to contracts that inherit from this contract. **(Hey, that sounds like what we want here!)**.

External

external is similar to public, except that these functions can ONLY be called outside the contract — they can't be called by other functions inside that contract. We'll talk about why you might want to use external vs public later.

For declaring internal or external functions, the syntax is the same as private and public:

```
contract Sandwich {
  uint private sandwichesEaten = 0;

  function eat() internal {
    sandwichesEaten++;
  }
}
contract BLT is Sandwich {
  uint private baconSandwichesEaten = 0;

  function eatWithBacon() public returns (string memory) {
    baconSandwichesEaten++;
    // We can call this here because it's internal
```

MEHMOOD SHEIKH

```
    eat();
  }
}
```

INTERACTING WITH OTHER CONTRACTS

For our contract to talk to another contract on the blockchain that we don't own, first we need to define an **interface**.

Let's look at a simple **example**. Say there was a contract on the blockchain that looked like this:

```
contract LuckyNumber {
  mapping(address => uint) numbers;

  function setNum(uint _num) public {
    numbers[msg.sender] = _num;
  }

  function getNum(address _myAddress) public view returns (uint) {
    return numbers[_myAddress];
  }
}
```

This would be a simple contract where anyone could store their lucky number, and it will be associated with their Ethereum address. Then anyone else could look up that person's lucky number using their address.

Now let's say we had an external contract that wanted to read the data in this contract using the getNum function.

First we'd have to define an **interface** of the LuckyNumber contract:

```
contract NumberInterface {
  function getNum(address _myAddress) public view returns (uint);
}
```

Notice that this looks like defining a contract, with a few differences. For one, we're only declaring the functions we want to interact with — in this case getNum — and we don't mention any of the other functions or state variables.

MEHMOOD SHEIKH

Secondly, we're not defining the function bodies. Instead of curly braces ({ and }), we're simply ending the function declaration with a semi-colon (;).

So it kind of looks like a contract skeleton. This is how the compiler knows it's an interface.

By including this interface in our dapp's code our contract knows what the other contract's functions look like, how to call them, and what sort of response to expect.

We'll get into actually calling the other contract's functions in the next lesson, but for now let's declare our interface for the CryptoKitties contract.

Put it to the test

We've looked up the CryptoKitties source code for you, and found a function called getKitty that returns all the kitty's data, including its "genes" (which is what our zombie game needs to form a new zombie!).

The function looks like this:

```
function getKitty(uint256 _id) external view returns (
    bool isGestating,
    bool isReady,
    uint256 cooldownIndex,
    uint256 nextActionAt,
    uint256 siringWithId,
    uint256 birthTime,
    uint256 matronId,
    uint256 sireId,
    uint256 generation,
    uint256 genes
) {
    Kitty storage kit = kitties[_id];

    // if this variable is 0 then it's not gestating
    isGestating = (kit.siringWithId != 0);
    isReady = (kit.cooldownEndBlock <= block.number);
```

MEHMOOD SHEIKH

```
    cooldownIndex = uint256(kit.cooldownIndex);
    nextActionAt = uint256(kit.cooldownEndBlock);
    siringWithId = uint256(kit.siringWithId);
    birthTime = uint256(kit.birthTime);
    matronId = uint256(kit.matronId);
    sireId = uint256(kit.sireId);
    generation = uint256(kit.generation);
    genes = kit.genes;
}
```

The function looks a bit different than we're used to. You can see it returns... a bunch of different values. If you're coming from a programming language like JavaScript, this is different — in Solidity you can return more than one value from a function.

USING AN INTERFACE

Continuing our previous example with NumberInterface, once we've defined the interface as:

```
contract NumberInterface {
  function getNum(address _myAddress) public view returns (uint);
}
```

We can use it in a contract as follows:

```
contract MyContract {
  address NumberInterfaceAddress = 0xab38...
  // ^ The address of the FavoriteNumber contract on Ethereum
  NumberInterface numberContract = NumberInterface(NumberInterfaceAddress);
  // Now `numberContract` is pointing to the other contract

  function someFunction() public {
    // Now we can call `getNum` from that contract:
    uint num = numberContract.getNum(msg.sender);
    // ...and do something with `num` here
  }
```

MEHMOOD SHEIKH

```
}
```

In this way, your contract can interact with any other contract on the Ethereum blockchain, as long they expose those functions as public or external.

HANDLING MULTIPLE RETURN VALUES

This getKitty function is the first example we've seen that returns multiple values. Let's look at how to handle them:

```
function multipleReturns() internal returns(uint a, uint b, uint c) {
  return (1, 2, 3);
}

function processMultipleReturns() external {
  uint a;
  uint b;
  uint c;
  // This is how you do multiple assignment:
  (a, b, c) = multipleReturns();
}

// Or if we only cared about one of the values:
function getLastReturnValue() external {
  uint c;
  // We can just leave the other fields blank:
  (,,c) = multipleReturns();
}
```

IF STATEMENTS

Our function logic is now complete... but let's add in one bonus feature.

Let's make it so zombies made from kitties have some unique feature that shows they're cat-zombies.

To do this, we can add some special kitty code in the zombie's DNA.

MEHMOOD SHEIKH

If you recall from lesson 1, we're currently only using the first 12 digits of our 16 digit DNA to determine the zombie's appearance. So let's use the last 2 unused digits to handle "special" characteristics.

We'll say that cat-zombies have 99 as their last two digits of DNA (since cats have 9 lives). So in our code, we'll say if a zombie comes from a cat, then set the last two digits of DNA to 99.

If statements

If statements in Solidity look just like JavaScript:

```
function eatBLT(string memory sandwich) public {
  // Remember with strings, we have to compare their keccak256 hashes
  // to check equality
  if (keccak256(abi.encodePacked(sandwich)) ==
keccak256(abi.encodePacked("BLT"))) {
    eat();
  }}
```

JAVASCRIPT IMPLEMENTATION

Once we're ready to deploy this contract to Ethereum we'll just compile and deploy ZombieFeeding — since this contract is our final contract that inherits from ZombieFactory, and has access to all the public methods in both contracts.

Let's look at an example of interacting with our deployed contract using JavaScript and web3.js:

```
var abi = /* abi generated by the compiler */
var ZombieFeedingContract = web3.eth.contract(abi)
var contractAddress = /* our contract address on Ethereum after deploying */
var ZombieFeeding = ZombieFeedingContract.at(contractAddress)

// Assuming we have our zombie's ID and the kitty ID we want to attack
let zombieId = 1;
let kittyId = 1;

// To get the CryptoKitty's image, we need to query their web API. This
// information isn't stored on the blockchain, just their webserver.
```

MEHMOOD SHEIKH

```
// If everything was stored on a blockchain, we wouldn't have to worry
// about the server going down, them changing their API, or the company
// blocking us from loading their assets if they don't like our zombie game ;)
let apiUrl = "https://api.cryptokitties.co/kitties/" + kittyId
$.get(apiUrl, function(data) {
  let imgUrl = data.image_url
  // do something to display the image
})

// When the user clicks on a kitty:
$(".kittyImage").click(function(e) {
  // Call our contract's `feedOnKitty` method
  ZombieFeeding.feedOnKitty(zombieId, kittyId)
})

// Listen for a NewZombie event from our contract so we can display it:
ZombieFactory.NewZombie(function(error, result) {
  if (error) return
  // This function will display the zombie, like in lesson 1:
  generateZombie(result.zombieId, result.name, result.dna)
})
```

ADVANCED SOLIDITY CONCEPTS

IMMUTABILITY OF CONTRACTS

Up until now, Solidity has looked quite similar to other languages like JavaScript. But there are a number of ways that Ethereum DApps are actually quite different from normal applications.

To start with, after you deploy a contract to Ethereum, it's **immutable**, which means that it can never be modified or updated again.

MEHMOOD SHEIKH

The initial code you deploy to a contract is there to stay, permanently, on the blockchain. This is one reason security is such a huge concern in Solidity. If there's a flaw in your contract code, there's no way for you to patch it later. You would have to tell your users to start using a different smart contract address that has the fix.

But this is also a feature of smart contracts. The code is law. If you read the code of a smart contract and verify it, you can be sure that every time you call a function it's going to do exactly what the code says it will do. No one can later change that function and give you unexpected results.

EXTERNAL DEPENDENCIES

In Lesson 2, we hard-coded the CryptoKitties contract address into our DApp. But what would happen if the CryptoKitties contract had a bug and someone destroyed all the kitties?

It's unlikely, but if this did happen it would render our DApp completely useless — our DApp would point to a hardcoded address that no longer returned any kitties. Our zombies would be unable to feed on kitties, and we'd be unable to modify our contract to fix it.

For this reason, it often makes sense to have functions that will allow you to update key portions of the DApp.

For example, instead of hard coding the CryptoKitties contract address into our DApp, we should probably have a setKittyContractAddress function that lets us change this address in the future in case something happens to the CryptoKitties contract.

OWNABLE CONTRACTS

Did you spot the security hole in the previous chapter?

setKittyContractAddress is external, so anyone can call it! That means anyone who called the function could change the address of the CryptoKitties contract, and break our app for all its users.

We do want the ability to update this address in our contract, but we don't want everyone to be able to update it.

To handle cases like this, one common practice that has emerged is to make contracts Ownable — meaning they have an owner (you) who has special privileges.

MEHMOOD SHEIKH

OPENZEPPELIN'S OWNABLE CONTRACT

Below is the Ownable contract taken from the **OpenZeppelin** Solidity library. OpenZeppelin is a library of secure and community-vetted smart contracts that you can use in your own DApps. After this lesson, we highly recommend you check out their site to further your learning!

Give the contract below a read-through. You're going to see a few things we haven't learned yet, but don't worry, we'll talk about them afterward.

```
/**
 * @title Ownable
 * @dev The Ownable contract has an owner address, and provides basic
authorization control
 * functions, this simplifies the implementation of "user permissions".
 */
contract Ownable {
  address private _owner;

  event OwnershipTransferred(
    address indexed previousOwner,
    address indexed newOwner
  );

  /**
   * @dev The Ownable constructor sets the original `owner` of the contract to
the sender
   * account.
   */
  constructor() internal {
    _owner = msg.sender;
    emit OwnershipTransferred(address(0), _owner);
  }

  /**
```

MEHMOOD SHEIKH

```
 * @return the address of the owner.
 */
function owner() public view returns(address) {
  return _owner;
}

/**
 * @dev Throws if called by any account other than the owner.
 */
modifier onlyOwner() {
  require(isOwner());
  _;
}

/**
 * @return true if `msg.sender` is the owner of the contract.
 */
function isOwner() public view returns(bool) {
  return msg.sender == _owner;
}

/**
 * @dev Allows the current owner to relinquish control of the contract.
 * @notice Renouncing to ownership will leave the contract without an owner.
 * It will not be possible to call the functions with the `onlyOwner`
 * modifier anymore.
 */
function renounceOwnership() public onlyOwner {
  emit OwnershipTransferred(_owner, address(0));
  _owner = address(0);
}
```

MEHMOOD SHEIKH

```
  /**
   * @dev Allows the current owner to transfer control of the contract to a
newOwner.
   * @param newOwner The address to transfer ownership to.
   */
  function transferOwnership(address newOwner) public onlyOwner {
    _transferOwnership(newOwner);
  }

  /**
   * @dev Transfers control of the contract to a newOwner.
   * @param newOwner The address to transfer ownership to.
   */
  function _transferOwnership(address newOwner) internal {
    require(newOwner != address(0));
    emit OwnershipTransferred(_owner, newOwner);
    _owner = newOwner;
  }
}
```

A few new things here we haven't seen before:

CONSTRUCTORS: constructor() is a **constructor**, which is an optional special function. It will get executed only one time, when the contract is first created.

FUNCTION MODIFIERS: modifier onlyOwner(). Modifiers are kind of half-functions that are used to modify other functions, usually to check some requirements prior to execution. In this case, onlyOwner can be used to limit access so **only** the **owner** of the contract can run this function. We'll talk more about function modifiers in the next chapter, and what that weird _; does.

INDEXED KEYWORD: don't worry about this one, we don't need it yet.

So the Ownable contract basically does the following:

MEHMOOD SHEIKH

1. When a contract is created, its constructor sets the owner to msg.sender (the person who deployed it)
2. It adds an onlyOwner modifier, which can restrict access to certain functions to only the owner
3. It allows you to transfer the contract to a new owner

onlyOwner is such a common requirement for contracts that most Solidity DApps start with a copy/paste of this Ownable contract, and then their first contract inherits from it.

Since we want to limit setKittyContractAddress to onlyOwner, we're going to do the same for our contract.

ONLYOWNER FUNCTION MODIFIER

Now that our base contract ZombieFactory inherits from Ownable, we can use the onlyOwner function modifier in ZombieFeeding as well.

This is because of how contract inheritance works. Remember:

ZombieFeeding is ZombieFactory

ZombieFactory is Ownable

Thus ZombieFeeding is also Ownable, and can access the functions / events / modifiers from the Ownable contract. This applies to any contracts that inherit from ZombieFeeding in the future as well.

FUNCTION MODIFIERS

A function modifier looks just like a function, but uses the keyword modifier instead of the keyword function. And it can't be called directly like a function can — instead we can attach the modifier's name at the end of a function definition to change that function's behavior.

Notice the onlyOwner modifier on the renounceOwnership function. When you call renounceOwnership, the code inside onlyOwner executes **first**. Then when it hits the _; statement in onlyOwner, it goes back and executes the code inside renounceOwnership.

So while there are other ways you can use modifiers, one of the most common use-cases is to add a quick require check before a function executes.

In the case of onlyOwner, adding this modifier to a function makes it so **only** the **owner** of the contract (you, if you deployed it) can call that function.

MEHMOOD SHEIKH

Note: *Giving the owner special powers over the contract like this is often necessary, but it could also be used maliciously. For example, the owner could add a backdoor function that would allow him to transfer anyone's zombies to himself!*

So it's important to remember that just because a DApp is on Ethereum does not automatically mean it's decentralized — you have to actually read the full source code to make sure it's free of special controls by the owner that you need to potentially worry about. There's a careful balance as a developer between maintaining control over a DApp such that you can fix potential bugs, and building an owner-less platform that your users can trust to secure their data.

GAS

Great! Now we know how to update key portions of the DApp while preventing other users from messing with our contracts.

Let's look at another way Solidity is quite different from other programming languages:

GAS — THE FUEL ETHEREUM DAPPS RUN ON

In Solidity, your users have to pay every time they execute a function on your DApp using a currency called **gas**. Users buy gas with Ether (the currency on Ethereum), so your users have to spend ETH in order to execute functions on your DApp.

How much gas is required to execute a function depends on how complex that function's logic is. Each individual operation has a **gas cost** based roughly on how much computing resources will be required to perform that operation (e.g. writing to storage is much more expensive than adding two integers). The total **gas cost** of your function is the sum of the gas costs of all its individual operations.

Because running functions costs real money for your users, code optimization is much more important in Ethereum than in other programming languages. If your code is sloppy, your users are going to have to pay a premium to execute your functions — and this could add up to millions of dollars in unnecessary fees across thousands of users.

WHY IS GAS NECESSARY?

Ethereum is like a big, slow, but extremely secure computer. When you execute a function, every single node on the network needs to run that same function to verify its output — thousands of nodes verifying every function execution is what makes Ethereum decentralized, and its data immutable and censorship-resistant.

MEHMOOD SHEIKH

The creators of Ethereum wanted to make sure someone couldn't clog up the network with an infinite loop, or hog all the network resources with really intensive computations. So they made it so transactions aren't free, and users have to pay for computation time as well as storage.

Note: This isn't necessarily true for other blockchain, like the ones the CryptoZombies authors are building at Loom Network. It probably won't ever make sense to run a game like World of Warcraft directly on the Ethereum mainnet — the gas costs would be prohibitively expensive. But it could run on a blockchain with a different consensus algorithm. We'll talk more about what types of DApps you would want to deploy on Loom vs the Ethereum mainnet in a future lesson.

STRUCT PACKING TO SAVE GAS

In Lesson 1, we mentioned that there are other types of uints: uint8, uint16, uint32, etc.

Normally there's no benefit to using these sub-types because Solidity reserves 256 bits of storage regardless of the uint size. For example, using uint8 instead of uint (uint256) won't save you any gas.

But there's an exception to this: inside structs.

If you have multiple uints inside a struct, using a smaller-sized uint when possible will allow Solidity to pack these variables together to take up less storage. For example:

```
struct NormalStruct {
  uint a;
  uint b;
  uint c;
}

struct MiniMe {
  uint32 a;
  uint32 b;
  uint c;
}

// `mini` will cost less gas than `normal` because of struct packing
```

MEHMOOD SHEIKH

```
NormalStruct normal = NormalStruct(10, 20, 30);
MiniMe mini = MiniMe(10, 20, 30);
```

For this reason, inside a struct you'll want to use the smallest integer sub-types you can get away with.

You'll also want to cluster identical data types together (i.e. put them next to each other in the struct) so that Solidity can minimize the required storage space. For example, a struct with fields uint c; uint32 a; uint32 b; will cost less gas than a struct with fields uint32 a; uint c; uint32 b; because the uint32 fields are clustered together.

TIME UNITS

The level property is pretty self-explanatory. Later on, when we create a battle system, zombies who win more battles will level up over time and get access to more abilities.

The readyTime property requires a bit more explanation. The goal is to add a "cooldown period", an amount of time a zombie has to wait after feeding or attacking before it's allowed to feed / attack again. Without this, the zombie could attack and multiply 1,000 times per day, which would make the game way too easy.

In order to keep track of how much time a zombie has to wait until it can attack again, we can use Solidity's time units.

Time units

Solidity provides some native units for dealing with time.

The variable now will return the current unix timestamp of the latest block (the number of seconds that have passed since January 1st 1970). The unix time as I write this is 1515527488.

Note: Unix time is traditionally stored in a 32-bit number. This will lead to the "Year 2038" problem, when 32-bit unix timestamps will overflow and break a lot of legacy systems. So if we wanted our DApp to keep running 20 years from now, we could use a 64-bit number instead — but our users would have to spend more gas to use our DApp in the meantime. Design decisions!

Solidity also contains the time units seconds, minutes, hours, days, weeks and years. These will convert to a uint of the number of seconds in that length of time. So 1 minutes is 60, 1 hours is 3600 (60 seconds x 60 minutes), 1 days is 86400 (24 hours x 60 minutes x 60 seconds), etc.

Here's an **example** of how these time units can be useful:

MEHMOOD SHEIKH

```
uint lastUpdated;

// Set `lastUpdated` to `now`
function updateTimestamp() public {
  lastUpdated = now;
}

// Will return `true` if 5 minutes have passed since `updateTimestamp` was
// called, `false` if 5 minutes have not passed
function fiveMinutesHavePassed() public view returns (bool) {
  return (now >= (lastUpdated + 5 minutes));
}
```

We can use these time units for our Zombie cooldown feature.

PASSING STRUCTS AS ARGUMENTS

You can pass a storage pointer to a struct as an argument to a private or internal function. This is useful, for example, for passing around our Zombie structs between functions.

The syntax looks like this:

```
function _doStuff(Zombie storage _zombie) internal {
  // do stuff with _zombie
}
```

This way we can pass a reference to our zombie into a function instead of passing in a zombie ID and looking it up.

PUBLIC FUNCTIONS & SECURITY

Now let's modify feedAndMultiply to take our cooldown timer into account.

Looking back at this function, you can see we made it public in the previous lesson. An important security practice is to examine all your public and external functions, and try to think of ways users might abuse them. Remember — unless these functions have a modifier like onlyOwner, any user can call them and pass them any data they want to.

MEHMOOD SHEIKH

Re-examining this particular function, the user could call the function directly and pass in any _targetDna or _species they want to. This doesn't seem very game-like — we want them to follow our rules!

On closer inspection, this function only needs to be called by feedOnKitty(), so the easiest way to prevent these exploits is to make it internal.

MORE ON FUNCTION MODIFIERS

Great! Our zombie now has a functional cooldown timer.

Next, we're going to add some additional helper methods. We've created a new file for you called zombiehelper.sol, which imports zombiefeeding.sol. This will help to keep our code organized.

Let's make it so zombies gain special abilities after reaching a certain level. But in order to do that, first we'll need to learn a little bit more about function modifiers.

FUNCTION MODIFIERS WITH ARGUMENTS

Previously we looked at the simple example of onlyOwner. But function modifiers can also take arguments.

For example:

```
// A mapping to store a user's age:
mapping (uint => uint) public age;

// Modifier that requires this user to be older than a certain age:
modifier olderThan(uint _age, uint _userId) {
  require(age[_userId] >= _age);
  _;
}

// Must be older than 16 to drive a car (in the US, at least).
// We can call the `olderThan` modifier with arguments like so:
function driveCar(uint _userId) public olderThan(16, _userId) {
  // Some function logic
}
```

MEHMOOD SHEIKH

You can see here that the olderThan modifier takes arguments just like a function does. And that the driveCar function passes its arguments to the modifier.

Let's try making our own modifier that uses the zombie level property to restrict access to special abilities.

USING MODIFIERS

Now let's use our aboveLevel modifier to create some functions.

Our game will have some incentives for people to level up their zombies:

- For zombies level 2 and higher, users will be able to change their name.
- For zombies level 20 and higher, users will be able to give them custom DNA.

We'll implement these functions below. Here's the example code from the previous lesson for reference:

```
// A mapping to store a user's age:
mapping (uint => uint) public age;

// Require that this user be older than a certain age:
modifier olderThan(uint _age, uint _userId) {
  require (age[_userId] >= _age);
  _;
}

// Must be older than 16 to drive a car (in the US, at least)
function driveCar(uint _userId) public olderThan(16, _userId) {
  // Some function logic
}
```

SAVING GAS WITH 'VIEW' FUNCTIONS

Awesome! Now we have some special abilities for higher-level zombies, to give our owners an incentive to level them up. We can add more of these later if we want to.

Let's add one more function: our DApp needs a method to view a user's entire zombie army — let's call it getZombiesByOwner.

MEHMOOD SHEIKH

This function will only need to read data from the blockchain, so we can make it a view function. Which brings us to an important topic when talking about gas optimization:

VIEW FUNCTIONS DON'T COST GAS

view functions don't cost any gas when they're called externally by a user.

This is because view functions don't actually change anything on the blockchain – they only read the data. So marking a function with view tells web3.js that it only needs to query your local Ethereum node to run the function, and it doesn't actually have to create a transaction on the blockchain (which would need to be run on every single node, and cost gas).

We'll cover setting up web3.js with your own node later. But for now the big takeaway is that you can optimize your DApp's gas usage for your users by using read-only external view functions wherever possible.

***Note:** If a view function is called internally from another function in the same contract that is **not** a view function, it will still cost gas. This is because the other function creates a transaction on Ethereum, and will still need to be verified from every node. So view functions are only free when they're called externally.*

STORAGE IS EXPENSIVE

One of the more expensive operations in Solidity is using storage — particularly writes.

This is because every time you write or change a piece of data, it's written permanently to the blockchain. Forever! Thousands of nodes across the world need to store that data on their hard drives, and this amount of data keeps growing over time as the blockchain grows. So there's a cost to doing that.

In order to keep costs down, you want to avoid writing data to storage except when absolutely necessary. Sometimes this involves seemingly inefficient programming logic — like rebuilding an array in memory every time a function is called instead of simply saving that array in a variable for quick lookups.

In most programming languages, looping over large data sets is expensive. But in Solidity, this is way cheaper than using storage if it's in an external view function, since view functions don't cost your users any gas. (And gas costs your users real money!).

We'll go over for loops in the next chapter, but first, let's go over how to declare arrays in memory.

MEHMOOD SHEIKH

DECLARING ARRAYS IN MEMORY

You can use the memory keyword with arrays to create a new array inside a function without needing to write anything to storage. The array will only exist until the end of the function call, and this is a lot cheaper gas-wise than updating an array in storage — free if it's a view function called externally.

Here's how to declare an array in memory:

function getArray() external pure returns(uint[] memory) {

```
  // Instantiate a new array in memory with a length of 3
  uint[] memory values = new uint[](3);

  // Put some values to it
  values[0] = 1;
  values[1] = 2;
  values[2] = 3;

  return values;
}
```

This is a trivial example just to show you the syntax, but in the next chapter we'll look at combining this with for loops for real use-cases.

*Note: memory arrays **must** be created with a length argument (in this example, 3). They currently cannot be resized like storage arrays can with array.push(), although this may be changed in a future version of Solidity.*

FOR LOOPS

In the previous chapter, we mentioned that sometimes you'll want to use a for loop to build the contents of an array in a function rather than simply saving that array to storage.

Let's look at why.

For our getZombiesByOwner function, a naive implementation would be to store a mapping of owners to zombie armies in the ZombieFactory contract:

mapping (address => uint[]) public ownerToZombies

MEHMOOD SHEIKH

Then every time we create a new zombie, we would simply use ownerToZombies[owner].push(zombieId) to add it to that owner's zombies array. And getZombiesByOwner would be a very straightforward function:

```
function getZombiesByOwner(address _owner) external view returns (uint[] memory)
{
  return ownerToZombies[_owner];
}
```

The problem with this approach

This approach is tempting for its simplicity. But let's look at what happens if we later add a function to transfer a zombie from one owner to another (which we'll definitely want to add in a later lesson!).

That transfer function would need to:

1. Push the zombie to the new owner's ownerToZombies array,
2. Remove the zombie from the old owner's ownerToZombies array,
3. Shift every zombie in the older owner's array up one place to fill the hole, and then
4. Reduce the array length by 1.

Step 3 would be extremely expensive gas-wise, since we'd have to do a write for every zombie whose position we shifted. If an owner has 20 zombies and trades away the first one, we would have to do 19 writes to maintain the order of the array.

Since writing to storage is one of the most expensive operations in Solidity, every call to this transfer function would be extremely expensive gas-wise. And worse, it would cost a different amount of gas each time it's called, depending on how many zombies the user has in their army and the index of the zombie being traded. So the user wouldn't know how much gas to send.

Note: *Of course, we could just move the last zombie in the array to fill the missing slot and reduce the array length by one. But then we would change the ordering of our zombie army every time we made a trade.*

Since view functions don't cost gas when called externally, we can simply use a for-loop in getZombiesByOwner to iterate the entire zombies array and build an array of the zombies that belong to this specific owner. Then our transfer function will be much cheaper, since we don't need to reorder any arrays in storage, and somewhat counter-intuitively this approach is cheaper overall.

MEHMOOD SHEIKH

USING FOR LOOPS

The syntax of for loops in Solidity is similar to JavaScript.

Let's look at an example where we want to make an array of even numbers:

```
function getEvens() pure external returns(uint[] memory) {
  uint[] memory evens = new uint[](5);
  // Keep track of the index in the new array:
  uint counter = 0;
  // Iterate 1 through 10 with a for loop:
  for (uint i = 1; i <= 10; i++) {
    // If `i` is even...
    if (i % 2 == 0) {
      // Add it to our array
      evens[counter] = i;
      // Increment counter to the next empty index in `evens`:
      counter++;
    }
  }
  return evens;
}
```

This function will return an array with the contents [2, 4, 6, 8, 10].

ZOMBIE BATTLE SYSTEM

PRVIOUSLY COVERED MODIFIERS

Up until now, we've covered quite a few **function modifiers**. It can be difficult to try to remember everything, so let's run through a quick review:

1. We have visibility modifiers that control when and where the function can be called from: private means it's only callable from other functions inside the contract; internal is like private but can also be called by contracts that inherit from this one; external can only be called outside the contract; and finally public can be called anywhere, both internally and externally.
2. We also have state modifiers, which tell us how the function interacts with the BlockChain: view tells us that by running the function, no data will be saved/changed. pure tells us that not only does the

MEHMOOD SHEIKH

function not save any data to the blockchain, but it also doesn't read any data from the blockchain. Both of these don't cost any gas to call if they're called externally from outside the contract (but they do cost gas if called internally by another function).

3. Then we have custom modifiers, which we learned about in Lesson 3: onlyOwner and aboveLevel, for example. For these we can define custom logic to determine how they affect a function.

These modifiers can all be stacked together on a function definition as follows:

function test() external view onlyOwner anotherModifier { */* ... */* }

In this chapter, we're going to introduce one more function modifier: payable.

THE PAYABLE MODIFIER

payable functions are part of what makes Solidity and Ethereum so cool — they are a special type of function that can receive Ether.

Let that sink in for a minute. When you call an API function on a normal web server, you can't send US dollars along with your function call — nor can you send Bitcoin.

But in Ethereum, because the money (*Ether*), the data (*transaction payload*), and the contract code itself all live on Ethereum, it's possible for you to call a function **and** pay money to the contract at the same time.

This allows for some really interesting logic, like requiring a certain payment to the contract in order to execute a function.

Let's look at an example

```
contract OnlineStore {
  function buySomething() external payable {
    // Check to make sure 0.001 ether was sent to the function call:
    require(msg.value == 0.001 ether);
    // If so, some logic to transfer the digital item to the caller of the function:
    transferThing(msg.sender);
  }
}
```

Here, msg.value is a way to see how much Ether was sent to the contract, and ether is a built-in unit.

What happens here is that someone would call the function from web3.js (from the DApp's JavaScript front-end) as follows:

```
// Assuming `OnlineStore` points to your contract on Ethereum:
```

MEHMOOD SHEIKH

```
OnlineStore.buySomething({from: web3.eth.defaultAccount, value:
web3.utils.toWei(0.001)})
```

Notice the value field, where the javascript function call specifies how much ether to send (0.001). If you think of the transaction like an envelope, and the parameters you send to the function call are the contents of the letter you put inside, then adding a value is like putting cash inside the envelope — the letter and the money get delivered together to the recipient.

Note: *If a function is not marked payable and you try to send Ether to it as above, the function will reject your transaction.*

WITHDRAWS

In the previous chapter, we learned how to send Ether to a contract. So what happens after you send it?

After you send Ether to a contract, it gets stored in the contract's Ethereum account, and it will be trapped there — unless you add a function to withdraw the Ether from the contract.

You can write a function to withdraw Ether from the contract as follows:

```
contract GetPaid is Ownable {

  function withdraw() external onlyOwner {

    address payable _owner = address(uint160(owner()));

    _owner.transfer(address(this).balance);

  }

}
```

Note that we're using owner() and onlyOwner from the Ownable contract, assuming that was imported.

It is important to note that you cannot transfer Ether to an address unless that address is of type address payable. But the _owner variable is of type uint160, meaning that we must explicitly cast it to address payable.

Once you cast the address from uint160 to address payable, you can transfer Ether to that address using the transfer function, and address(this).balance will return the total balance stored on the contract. So if 100 users had paid 1 Ether to our contract, address(this).balance would equal 100 Ether.

You can use transfer to send funds to any Ethereum address. For example, you could have a function that transfers Ether back to the msg.sender if they overpaid for an item:

```
uint itemFee = 0.001 ether;

msg.sender.transfer(msg.value - itemFee);
```

Or in a contract with a buyer and a seller, you could save the seller's address in storage, then when someone purchases his item, transfer him the fee paid by the buyer: seller.transfer(msg.value).

These are some examples of what makes Ethereum programming really cool — you can have decentralized marketplaces like this that aren't controlled by anyone.

MEHMOOD SHEIKH

RANDOM NUMBERS

Great! Now let's figure out the battle logic.

All good games require some level of randomness. So how do we generate random numbers in Solidity?

The real answer here is, you can't. Well, at least you can't do it safely.

Let's look at why.

Random number generation via keccak256

The best source of randomness we have in Solidity is the keccak256 hash function.

We could do something like the following to generate a random number:

```
// Generate a random number between 1 and 100:

uint randNonce = 0;

uint random = uint(keccak256(abi.encodePacked(now, msg.sender, randNonce))) %
100;

randNonce++;

uint random2 = uint(keccak256(abi.encodePacked(now, msg.sender, randNonce))) %
100;
```

What this would do is take the timestamp of now, the msg.sender, and an incrementing nonce (a number that is only ever used once, so we don't run the same hash function with the same input parameters twice).

It would then "pack" the inputs and use keccak to convert them to a random hash. Next, it would convert that hash to a uint, and then use % 100 to take only the last 2 digits. This will give us a totally random number between 0 and 99.

This method is vulnerable to attack by a dishonest node.

In Ethereum, when you call a function on a contract, you broadcast it to a node or nodes on the network as a **transaction**. The nodes on the network then collect a bunch of transactions, try to be the first to solve a computationally-intensive mathematical problem as a "Proof of Work", and then publish that group of transactions along with their Proof of Work (PoW) as a **block** to the rest of the network.

Once a node has solved the PoW, the other nodes stop trying to solve the PoW, verify that the other node's list of transactions are valid, and then accept the block and move on to trying to solve the next block.

This makes our random number function exploitable.

Let's say we had a coin flip contract — heads you double your money, tails you lose everything. Let's say it used the above random function to determine heads or tails. (random >= 50 is heads, random < 50 is tails).

MEHMOOD SHEIKH

If I were running a node, I could publish a transaction **only to my own node** and not share it. I could then run the coin flip function to see if I won — and if I lost, choose not to include that transaction in the next block I'm solving. I could keep doing this indefinitely until I finally won the coin flip and solved the next block, and profit.

So how do we generate random numbers safely in Ethereum?

Because the entire contents of the blockchain are visible to all participants, this is a hard problem, and its solution is beyond the scope of this tutorial. You can read this StackOverflow thread for some ideas. One idea would be to use an **oracle** to access a random number function from outside of the Ethereum blockchain.

Of course, since tens of thousands of Ethereum nodes on the network are competing to solve the next block, my odds of solving the next block are extremely low. It would take me a lot of time or computing resources to exploit this profitably — but if the reward were high enough (like if I could bet $100,000,000 on the coin flip function), it would be worth it for me to attack.

So while this random number generation is NOT secure on Ethereum, in practice unless our random function has a lot of money on the line, the users of your game likely won't have enough resources to attack it.

Because we're just building a simple game for demo purposes in this tutorial and there's no real money on the line, we're going to accept the tradeoffs of using a random number generator that is simple to implement, knowing that it isn't totally secure.

In a future lesson, we may cover using **oracles** (a secure way to pull data in from outside of Ethereum) to generate secure random numbers from outside the blockchain.

BUILDING ZOMBIE FIGHTIN' LOGIC

Now that we have a source of some randomness in our contract, we can use it in our zombie battles to calculate the outcome.

Our zombie battles will work as follows:

- You choose one of your zombies, and choose an opponent's zombie to attack.
- If you're the attacking zombie, you will have a 70% chance of winning. The defending zombie will have a 30% chance of winning.
- All zombies (attacking and defending) will have a winCount and a lossCount that will increment depending on the outcome of the battle.
- If the attacking zombie wins, it levels up and spawns a new zombie.
- If it loses, nothing happens (except its lossCount incrementing).
- Whether it wins or loses, the attacking zombie's cooldown time will be triggered.

This is a lot of logic to implement, so we'll do it in pieces over the coming chapters.

MEHMOOD SHEIKH

REFACTORING COMMON LOGIC

Whoever calls our attack function — we want to make sure the user actually owns the zombie they're attacking with. It would be a security concern if you could attack with someone else's zombie!

Can you think of how we would add a check to see if the person calling this function is the owner of the _zombieId they're passing in?

Give it some thought, and see if you can come up with the answer on your own.

Take a moment... Refer to some of our previous lessons' code for ideas...

Answer below, don't continue until you've given it some thought.

The answer

We've done this check multiple times now in previous lessons. In changeName(), changeDna(), and feedAndMultiply(), we used the following check:

require(msg.sender == zombieToOwner[_zombieId]);

This is the same logic we'll need for our attack function. Since we're using the same logic multiple times, let's move this into its own modifier to clean up our code and avoid repeating ourselves.

ZOMBIE WINS AND LOSSES

For our zombie game, we're going to want to keep track of how many battles our zombies have won and lost. That way we can maintain a "zombie leaderboard" in our game state.

We could store this data in a number of ways in our DApp — as individual mappings, as leaderboard Struct, or in the Zombie struct itself.

Each has its own benefits and tradeoffs depending on how we intend on interacting with the data. In this tutorial, we're going to store the stats on our Zombie struct for simplicity, and call them winCount and lossCount.

So let's jump back to zombiefactory.sol, and add these properties to our Zombie struct.

ZOMBIE LOSS

Now that we've coded what happens when your zombie wins, let's figure out what happens when it **loses**.

In our game, when zombies lose, they don't level down — they simply add a loss to their lossCount, and their cooldown is triggered so they have to wait a day before attacking again.

To implement this logic, we'll need an else statement.

else statements are written just like in JavaScript and many other languages:

```
if (zombieCoins[msg.sender] > 100000000) {
  // You rich!!!
} else {
```

MEHMOOD SHEIKH

```
  // We require more ZombieCoins...
}
```

ERC721 & CRYPTO-COLLECTIBLES

TOKENS ON ETHEREUM

Let's talk about **tokens**.

If you've been in the Ethereum space for any amount of time, you've probably heard people talking about tokens — specifically **ERC20 tokens**.

A **token** on Ethereum is basically just a smart contract that follows some common rules — namely it implements a standard set of functions that all other token contracts share, such as `transferFrom(address _from, address _to, uint256 _amount)` and `balanceOf(address _owner)`.

Internally the smart contract usually has a mapping, `mapping(address => uint256) _`balances, that keeps track of how much balance each address has.

So basically a token is just a contract that keeps track of who owns how much of that token, and some functions so those users can transfer their tokens to other addresses.

Why does it matter?

Since all ERC20 tokens share the same set of functions with the same names, they can all be interacted with in the same ways.

This means if you build an application that is capable of interacting with one ERC20 token, it's also capable of interacting with any ERC20 token. That way more tokens can easily be added to your app in the future without needing to be custom coded. You could simply plug in the new token contract address, and boom, your app has another token it can use.

One example of this would be an exchange. When an exchange adds a new ERC20 token, really it just needs to add another smart contract it talks to. Users can tell that contract to send tokens to the exchange's wallet address, and the exchange can tell the contract to send the tokens back out to users when they request a withdraw.

The exchange only needs to implement this transfer logic once, then when it wants to add a new ERC20 token, it's simply a matter of adding the new contract address to its database.

Other token standards

ERC20 tokens are really cool for tokens that act like currencies. But they're not particularly useful for representing zombies in our zombie game.

MEHMOOD SHEIKH

For one, zombies aren't divisible like currencies — I can send you 0.237 ETH, but transfering you 0.237 of a zombie doesn't really make sense.

Secondly, all zombies are not created equal. Your Level 2 zombie "**Steve**" is totally not equal to my Level 732 zombie "**H4XF13LD MORRIS** 💯 💯 😎 💯 💯". (Not even close, *Steve*).

There's another token standard that's a much better fit for crypto-collectibles like CryptoZombies — and they're called **ERC721 tokens.**

ERC721 tokens

ERC721 tokens are **not** interchangeable since each one is assumed to be unique, and are not divisible. You can only trade them in whole units, and each one has a unique ID. So these are a perfect fit for making our zombies tradeable.

***Note**: that using a standard like ERC721 has the benefit that we don't have to implement the auction or escrow logic within our contract that determines how players can trade / sell our zombies. If we conform to the spec, someone else could build an exchange platform for crypto-tradable ERC721 assets, and our ERC721 zombies would be usable on that platform. So there are clear benefits to using a token standard instead of rolling your own trading logic.*

ERC721 STANDARD, MULTIPLE INHERITANCE

Let's take a look at the ERC721 standard:

```
contract ERC721 {

  event Transfer(address indexed _from, address indexed _to, uint256 indexed
_tokenId);

  event Approval(address indexed _owner, address indexed _approved, uint256
indexed _tokenId);


  function balanceOf(address _owner) external view returns (uint256);

  function ownerOf(uint256 _tokenId) external view returns (address);

  function transferFrom(address _from, address _to, uint256 _tokenId) external
payable;

  function approve(address _approved, uint256 _tokenId) external payable;

}
```

This is the list of methods we'll need to implement, which we'll be doing over the coming chapters in pieces.

It looks like a lot, but don't get overwhelmed! We're here to walk you through it.

Implementing a token contract

MEHMOOD SHEIKH

When implementing a token contract, the first thing we do is copy the interface to its own Solidity file and import it, import "./erc721.sol";. Then we have our contract inherit from it, and we override each method with a function definition.

But wait — ZombieOwnership is already inheriting from ZombieAttack — how can it also inherit from ERC721?

Luckily in Solidity, your contract can inherit from multiple contracts as follows:

```
contract SatoshiNakamoto is NickSzabo, HalFinney {

  // Omg, the secrets of the universe revealed!

}
```

This is the list of methods we'll need to implement, which we'll be doing over the coming chapters in pieces.

It looks like a lot, but don't get overwhelmed! We're here to walk you through it.

IMPLEMENTING A TOKEN CONTRACT

When implementing a token contract, the first thing we do is copy the interface to its own Solidity file and import it, import "./erc721.sol";. Then we have our contract inherit from it, and we override each method with a function definition.

But wait — ZombieOwnership is already inheriting from ZombieAttack — how can it also inherit from ERC721?

Luckily in Solidity, your contract can inherit from multiple contracts as follows:

```
contract SatoshiNakamoto is NickSzabo, HalFinney {

  // Omg, the secrets of the universe revealed!

}
```

As you can see, when using multiple inheritance, you just separate the multiple contracts you're inheriting from with a comma, ,. In this case, our contract is inheriting from NickSzabo and HalFinney.

BALANCEOF & OWNEROF

Great, let's dive into the ERC721 implementation!

We've gone ahead and copied the empty shell of all the functions you'll be implementing in this lesson.

In this chapter, we're going to implement the first two methods: balanceOf and ownerOf.

balanceOf

```
  function balanceOf(address _owner) external view returns (uint256 _balance);
```

This function simply takes an address, and returns how many tokens that address owns.

MEHMOOD SHEIKH

In our case, our "tokens" are Zombies. Do you remember where in our DApp we stored how many zombies an owner has?

ownerOf

```
function ownerOf(uint256 _tokenId) external view returns (address _owner);
```

This function takes a token ID (in our case, a Zombie ID), and returns the address of the person who owns it.

Again, this is very straightforward for us to implement, since we already have a mapping in our DApp that stores this information. We can implement this function in one line, just a return statement.

Note: Remember, uint256 is equivalent to uint. We've been using uint in our code up until now, but we're using uint256 here because we copy/pasted from the spec.

REFACTORING

Uh oh! We've just introduced an error in our code that will make it not compile. Did you notice it?

In the previous chapter we defined a function called ownerOf. But if you recall from Lesson 4, we also created a modifier with the same name, ownerOf, in zombiefeeding.sol.

If you tried compiling this code, the compiler would give you an error saying you can't have a modifier and a function with the same name.

So should we just change the function name in ZombieOwnership to something else?

No, we can't do that!!! Remember, we're using the ERC721 token standard, which means other contracts will expect our contract to have functions with these exact names defined. That's what makes these standards useful — if another contract knows our contract is ERC721-compliant, it can simply talk to us without needing to know anything about our internal implementation decisions.

So that means we'll have to refactor our code from Lesson 4 to change the name of the modifier to something else.

ERC721: TRANSFER LOGIC

Great, we've fixed the conflict!

Now we're going to continue our ERC721 implementation by looking at transfering ownership from one person to another.

Note that the ERC721 spec has 2 different ways to transfer tokens:

function transferFrom(address _from, address _to, uint256 _tokenId) external payable;

and

function approve(address _approved, uint256 _tokenId) external payable;

function transferFrom(address _from, address _to, uint256 _tokenId) external payable;

MEHMOOD SHEIKH

1. The first way is the token's owner calls transferFrom with his address as the _from parameter, the address he wants to transfer to as the _to parameter, and the _tokenId of the token he wants to transfer.
2. The second way is the token's owner first calls approve with the address he wants to transfer to, and the _tokenID . The contract then stores who is approved to take a token, usually in a mapping (uint256 => address). Then, when the owner or the approved address calls transferFrom, the contract checks if that msg.sender is the owner or is approved by the owner to take the token, and if so it transfers the token to him.

Notice that both methods contain the same transfer logic. In one case the sender of the token calls the transferFrom function; in the other the owner or the approved receiver of the token calls it.

So it makes sense for us to abstract this logic into its own private function, _transfer, which is then called by transferFrom.

ERC721: APPROVE

Now, let's implement approve.

Remember, with approve the transfer happens in 2 steps:

1. You, the owner, call approve and give it the _approved address of the new owner, and the _tokenId you want them to take.
2. The new owner calls transferFrom with the _tokenId. Next, the contract checks to make sure the new owner has been already approved, and then transfers them the token.

Because this happens in 2 function calls, we need to use the zombieApprovals data structure to store who's been approved for what in between function calls.

Great, we are almost done!

There is one more thing to do- there's an Approval event in the ERC721 spec. So we should fire this event at the end of the approve function.

PREVENTING OVERFLOWS

Congratulations, that completes our **ERC721** and **ERC721x** implementation!

That wasn't so tough, was it? A lot of this Ethereum stuff sounds really complicated when you hear people talking about it, so the best way to understand it is to actually go through an implementation of it yourself.

Keep in mind that this is only a minimal implementation. There are extra features we may want to add to our implementation, such as some extra checks to make sure users don't accidentally transfer their zombies to address 0 (which is called "burning" a token — basically it's sent to an address that no one has the private key

MEHMOOD SHEIKH

of, essentially making it unrecoverable). Or to put some basic auction logic in the DApp itself. (Can you think of some ways we could implement that?)

But we wanted to keep this lesson manageable, so we went with the most basic implementation. If you want to see an example of a more in-depth implementation, you can take a look at the OpenZeppelin ERC721 contract after this tutorial.

Contract security enhancements: Overflows and Underflows

We're going to look at one major security feature you should be aware of when writing smart contracts: Preventing overflows and underflows.

WHAT'S AN OVERFLOW?

Let's say we have a uint8, which can only have 8 bits. That means the largest number we can store is binary 11111111 (or in decimal, 2^8 - 1 = 255).

Take a look at the following code. What is number equal to at the end?

```
uint8 number = 255;

number++;
```

In this case, we've caused it to overflow — so number is counterintuitively now equal to 0 even though we increased it. (If you add 1 to binary 11111111, it resets back to 00000000, like a clock going from 23:59 to 00:00).

WHAT'S AN UNDERFLOW?

An underflow is similar, where if you subtract 1 from a uint8 that equals 0, it will now equal 255 (because uints are unsigned, and cannot be negative).

While we're not using uint8 here, and it seems unlikely that a uint256 will overflow when incrementing by 1 each time (2^256 is a really big number), it's still good to put protections in our contract so that our DApp never has unexpected behavior in the future.

Using SafeMath

To prevent this, **OpenZeppelin** has created a **library** called SafeMath that prevents these issues by default.

But before we get into that...

What's a library?

A **library** is a special type of contract in Solidity. One of the things it is useful for is to attach functions to native data types.

For example, with the SafeMath library, we'll use the syntax using SafeMath for uint256. The SafeMath library has 4 functions — add, sub, mul, and div. And now we can access these functions from uint256 as follows:

```
using SafeMath for uint256;
```

MEHMOOD SHEIKH

```
uint256 a = 5;
uint256 b = a.add(3); // 5 + 3 = 8
uint256 c = a.mul(2); // 5 * 2 = 10
```

We'll look at what these functions do in the next chapter, but for now let's add the SafeMath library to our contract.

SAFEMATH PART 2

Let's take a look at the code behind SafeMath:

```
library SafeMath {

  function mul(uint256 a, uint256 b) internal pure returns (uint256) {
    if (a == 0) {
      return 0;
    }
    uint256 c = a * b;
    assert(c / a == b);
    return c;
  }

  function div(uint256 a, uint256 b) internal pure returns (uint256) {
    // assert(b > 0); // Solidity automatically throws when dividing by 0
    uint256 c = a / b;
    // assert(a == b * c + a % b); // There is no case in which this doesn't hold
    return c;
  }

  function sub(uint256 a, uint256 b) internal pure returns (uint256) {
    assert(b <= a);
    return a - b;
  }
```

MEHMOOD SHEIKH

```
  function add(uint256 a, uint256 b) internal pure returns (uint256) {
    uint256 c = a + b;
    assert(c >= a);
    return c;
  }
}
```

First we have the **library** keyword — libraries are similar to contracts but with a few differences. For our purposes, libraries allow us to use the using keyword, which automatically tacks on all of the library's methods to another data type:

```
using SafeMath for uint;
// now we can use these methods on any uint
uint test = 2;
test = test.mul(3); // test now equals 6
test = test.add(5); // test now equals 11
```

Note that the mul and add functions each require 2 arguments, but when we declare using SafeMath for uint, the uint we call the function on (test) is automatically passed in as the first argument.

Let's look at the code behind add to see what SafeMath does:

```
function add(uint256 a, uint256 b) internal pure returns (uint256) {
  uint256 c = a + b;
  assert(c >= a);
  return c;
}
```

Basically **add** just adds 2 uints like +, but it also contains an assert statement to make sure the sum is greater than a. This protects us from overflows.

ASSERT

assert is similar to require, where it will throw an error if false. The difference between assert and require is that require will refund the user the rest of their gas when a function fails, whereas assert will not. So most of the time you want to use require in your code; assert is typically used when something has gone horribly wrong with the code (like a uint overflow).

So, simply put, SafeMath's add, sub, mul, and div are functions that do the basic 4 math operations, but throw an error if an overflow or underflow occurs.

Using SafeMath in our code.

MEHMOOD SHEIKH

To prevent overflows and underflows, we can look for places in our code where we use +, -, *, or /, and replace them with add, sub, mul, div.

Ex. Instead of doing:

```
myUint++;
```

We would do:

```
myUint = myUint.add(1);
```

SAFEMATH PART 3

Great, now our ERC721 implementation is safe from overflows & underflows!

Going back through the code we wrote in previous lessons, there's a few other places in our code that could be vulnerable to overflows or underflows.

For example, in ZombieAttack we have:

```
myZombie.winCount++;

myZombie.level++;

enemyZombie.lossCount++;
```

We should prevent overflows here as well just to be safe. (It's a good idea in general to just use SafeMath instead of the basic math operations. Maybe in a future version of Solidity these will be implemented by default, but for now we have to take extra security precautions in our code).

However we have a slight problem — winCount and lossCount are uint16s, and level is a uint32. So if we use SafeMath's add method with these as arguments, it won't actually protect us from overflow since it will convert these types to uint256:

```
function add(uint256 a, uint256 b) internal pure returns (uint256) {
  uint256 c = a + b;
  assert(c >= a);
  return c;
}


// If we call `.add` on a `uint8`, it gets converted to a `uint256`.
// So then it won't overflow at 2^8, since 256 is a valid `uint256`.
```

This means we're going to need to implement 2 more libraries to prevent overflow/underflows with our uint16s and uint32s. We can call them SafeMath16 and SafeMath32.

The code will be exactly the same as SafeMath, except all instances of uint256 will be replaced with uint32 or uint16.

MEHMOOD SHEIKH

We've gone ahead and implemented that code for you — go ahead and look at safemath.sol to see the code.

Now we need to implement it in ZombieFactory.

COMMENTS

The Solidity code for our zombie game is finally finished!

In the next lessons, we'll look at how to deploy the code to Ethereum, and how to interact with it with Web3.js.

But one final thing before we let you go in Lesson 5: Let's talk about **commenting your code**.

Syntax for comments

Commenting in Solidity is just like JavaScript. You've already seen some examples of single line comments throughout the CryptoZombies lessons:

```
// This is a single-line comment. It's kind of like a note to self (or to others)
```

Just add double // anywhere and you're commenting. It's so easy that you should do it all the time.

But I hear you — sometimes a single line is not enough. You are born a writer, after all!

Thus we also have multi-line comments:

```
contract CryptoZombies {
  /* This is a multi-lined comment. I'd like to thank all of you
     who have taken your time to try this programming course.
     I know it's free to all of you, and it will stay free
     forever, but we still put our heart and soul into making
     this as good as it can be.

     Know that this is still the beginning of Blockchain development.
     We've come very far but there are so many ways to make this
     community better. If we made a mistake somewhere, you can
     help us out and open a pull request here:
     https://github.com/loomnetwork/cryptozombie-lessons

     Or if you have some ideas, comments, or just want to say
     hi - drop by our Telegram community at https://t.me/loomnetworkdev
  */
```

MEHMOOD SHEIKH

```
}
```

In particular, it's good practice to comment your code to explain the expected behavior of every function in your contract. This way another developer (or you, after a 6 month hiatus from a project!) can quickly skim and understand at a high level what your code does without having to read the code itself.

NATSPEC

The standard in the Solidity community is to use a format called **natspec**, which looks like this:

```
/// @title A contract for basic math operations
/// @author H4XF13LD MORRIS 💯💯😎💯💯
/// @notice For now, this contract just adds a multiply function
contract Math {
  /// @notice Multiplies 2 numbers together
  /// @param x the first uint.
  /// @param y the second uint.
  /// @return z the product of (x * y)
  /// @dev This function does not currently check for overflows
  function multiply(uint x, uint y) returns (uint z) {
    // This is just a normal comment, and won't get picked up by natspec
    z = x * y;
  }
}
```

@title and **@author** are straightforward.

@notice explains to a **user** what the contract / function does. @dev is for explaining extra details to developers.

@param and **@return** are for describing what each parameter and return value of a function are for.

Note that you don't always have to use all of these tags for every function — all tags are optional. But at the very least, leave a @dev note explaining what each function does.

MEHMOOD SHEIKH

APP FRONT-ENDS & WEB3.JS

INTRO TO WEB3.JS

Now we're going to create a basic web page where your users can interact with it.

To do this, we're going to use a JavaScript library from the Ethereum Foundation called **Web3.js**.

What is Web3.js?

Remember, the Ethereum network is made up of nodes, with each containing a copy of the blockchain. When you want to call a function on a smart contract, you need to query one of these nodes and tell it:

1. The address of the smart contract
2. The function you want to call, and
3. The variables you want to pass to that function.

Ethereum nodes only speak a language called **JSON-RPC**, which isn't very human-readable. A query to tell the node you want to call a function on a contract looks something like this:

```
// Yeah... Good luck writing all your function calls this way!

// Scroll right ==>

{"jsonrpc":"2.0",

"method":"eth_sendTransaction",

"params":[{"from":"0xb60e8dd61c5d32be8058bb8eb970870f07233155",

          "to":"0xd46e8dd67c5d32be8058bb8eb970870f07244567",

          "gas":"0x76c0",

          "gasPrice":"0x9184e72a000",

          "value":"0x9184e72a",

          "data":"0xd46e8dd67c5d32be8d46e8dd67c5d32be8058bb8eb970870f072445675058bb8eb970870f072445675"}],

          "id":1}
```

Luckily, Web3.js hides these nasty queries below the surface, so you only need to interact with a convenient and easily readable JavaScript interface.

Instead of needing to construct the above query, calling a function in your code will look something like this:

```
CryptoZombies.methods.createRandomZombie("Vitalik Nakamoto 🤔")
  .send({ from: "0xb60e8dd61c5d32be8058bb8eb970870f07233155", gas: "3000000" })
```

MEHMOOD SHEIKH

We'll explain the syntax in detail over the next few chapters, but first let's get your project set up with Web3.js.

Getting started

Depending on your project's workflow, you can add Web3.js to your project using most package tools:

```
// Using NPM
npm install web3

// Using Yarn
yarn add web3

// Using Bower
bower install web3

// ...etc.
```

Or you can simply download the minified .js file from github and include it in your project:

```
<script language="javascript" type="text/javascript" src="web3.min.js"></script>
```

Since we don't want to make too many assumptions about your development environment and what package manager you use, for this tutorial we're going to simply include Web3 in our project using a script tag as above.

WEB3 PROVIDERS

he first thing we need is a **Web3 Provider**.

Remember, Ethereum is made up of **nodes** that all share a copy of the same data. Setting a Web3 Provider in Web3.js tells our code **which node** we should be talking to handle our reads and writes. It's kind of like setting the URL of the remote web server for your API calls in a traditional web app.

You could host your own Ethereum node as a provider. However, there's a third-party service that makes your life easier so you don't need to maintain your own Ethereum node in order to provide a DApp for your users — **Infura**.

INFURA

Infura is a service that maintains a set of Ethereum nodes with a caching layer for fast reads, which you can access for free through their API. Using Infura as a provider, you can reliably send and receive messages to/from the Ethereum blockchain without needing to set up and maintain your own node.

You can set up Web3 to use Infura as your web3 provider as follows:

MEHMOOD SHEIKH

```
var web3 = new Web3(new
Web3.providers.WebsocketProvider("wss://mainnet.infura.io/ws"));
```

However, since our DApp is going to be used by many users — and these users are going to WRITE to the blockchain and not just read from it — we'll need a way for these users to sign transactions with their private key.

Note: *Ethereum (and blockchains in general) use a public / private key pair to digitally sign transactions. Think of it like an extremely secure password for a digital signature. That way if I change some data on the blockchain, I can* ***prove*** *via my public key that I was the one who signed it — but since no one knows my private key, no one can forge a transaction for me.*

Cryptography is complicated, so unless you're a security expert and you really know what you're doing, it's probably not a good idea to try to manage users' private keys yourself in our app's front-end.

But luckily you don't need to — there are already services that handle this for you. The most popular of these is **Metamask**.

METAMASK

Metamask is a browser extension for Chrome and Firefox that lets users securely manage their Ethereum accounts and private keys, and use these accounts to interact with websites that are using Web3.js. (If you haven't used it before, you'll definitely want to go and install it — then your browser is Web3 enabled, and you can now interact with any website that communicates with the Ethereum blockchain!).

And as a developer, if you want users to interact with your DApp through a website in their web browser (like we're doing with our CryptoZombies game), you'll definitely want to make it Metamask-compatible.

Note*: Metamask uses Infura's servers under the hood as a web3 provider, just like we did above — but it also gives the user the option to choose their own web3 provider. So by using Metamask's web3 provider, you're giving the user a choice, and it's one less thing you have to worry about in your app.*

Using Metamask's web3 provider

Metamask injects their web3 provider into the browser in the global JavaScript object web3. So your app can check to see if web3 exists, and if it does use web3.currentProvider as its provider.

Here's some template code provided by Metamask for how we can detect to see if the user has Metamask installed, and if not tell them they'll need to install it to use our app:

```
window.addEventListener('load', function() {

  // Checking if Web3 has been injected by the browser (Mist/MetaMask)

  if (typeof web3 !== 'undefined') {

    // Use Mist/MetaMask's provider

    web3js = new Web3(web3.currentProvider);
```

MEHMOOD SHEIKH

```
  } else {
    // Handle the case where the user doesn't have web3. Probably
    // show them a message telling them to install Metamask in
    // order to use our app.
  }
  // Now you can start your app & access web3js freely:
  startApp()
})
```

You can use this boilerplate code in all the apps you create in order to require users to have Metamask to use your DApp.

Note: *There are other private key management programs your users might be using besides MetaMask, such as the web browser* ***Mist****. However, they all implement a common pattern of injecting the variable web3, so the method we describe here for detecting the user's web3 provider will work for these as well.*

TALKING TO CONTRACTS

Web3.js will need 2 things to talk to your contract: its **address** and its **ABI**.

CONTRACT ADDRESS

After you finish writing your smart contract, you will compile it and deploy it to Ethereum. We're going to cover **deployment** in the **next lesson**, but since that's quite a different process from writing code, we've decided to go out of order and cover Web3.js first.

After you deploy your contract, it gets a fixed address on Ethereum where it will live forever. If you recall from Lesson 2, the address of the CryptoKitties contract on Ethereum mainnet is 0x06012c8cf97BEaD5deAe237070F9587f8E7A266d.

You'll need to copy this address after deploying in order to talk to your smart contract.

CONTRACT ABI

The other thing Web3.js will need to talk to your contract is its **ABI**.

ABI stands for Application Binary Interface. Basically it's a representation of your contracts' methods in JSON format that tells Web3.js how to format function calls in a way your contract will understand.

When you compile your contract to deploy to Ethereum (which we'll cover in Lesson 7), the Solidity compiler will give you the ABI, so you'll need to copy and save this in addition to the contract address.

Since we haven't covered deployment yet, for this lesson we've compiled the ABI for you and put it in a file named cryptozombies_abi.js, stored in variable called cryptoZombiesABI.

If we include cryptozombies_abi.js in our project, we'll be able to access the CryptoZombies ABI using that variable.

MEHMOOD SHEIKH

Instantiating a Web3.js Contract

Once you have your contract's address and ABI, you can instantiate it in Web3 as follows:

```
// Instantiate myContract
var myContract = new web3js.eth.Contract(myABI, myContractAddress);
```

CALLING CONTRACT FUNCTIONS

Web3.js has two methods we will use to call functions on our contract: call and send.

Call

call is used for view and pure functions. It only runs on the local node, and won't create a transaction on the blockchain.

Review: *view and pure functions are read-only and don't change state on the blockchain. They also don't cost any gas, and the user won't be prompted to sign a transaction with MetaMask.*

Using **Web3.js**, you would call a function named **myMethod** with the parameter 123 as follows:

```
myContract.methods.myMethod(123).call()
```

Send

send will create a transaction and change data on the blockchain. You'll need to use send for any functions that aren't view or pure.

Note: *sending a transaction will require the user to pay gas, and will pop up their Metamask to prompt them to sign a transaction. When we use Metamask as our web3 provider, this all happens automatically when we call send(), and we don't need to do anything special in our code. Pretty cool!*

Using **Web3.js**, you would send a transaction calling a function named **myMethod** with the parameter 123 as follows:

```
myContract.methods.myMethod(123).send()
```

The syntax is almost identical to **call()**.

Getting Zombie Data

Now let's look at a real example of using call to access data on our contract.

Recall that we made our array of zombies public:

```
Zombie[] public zombies;
```

In Solidity, when you declare a variable public, it automatically creates a public "getter" function with the same name. So if you wanted to look up the zombie with id 15, you would call it as if it were a function: zombies(15).

MEHMOOD SHEIKH

Here's how we would write a JavaScript function in our front-end that would take a zombie id, query our contract for that zombie, and return the result:

***Note**: All the code examples we're using in this lesson are using **version 1.0** of Web3.js, which uses promises instead of callbacks. Many other tutorials you'll see online are using an older version of Web3.js. The syntax changed a lot with version 1.0, so if you're copying code from other tutorials, make sure they're using the same version as you!*

```
function getZombieDetails(id) {
  return cryptoZombies.methods.zombies(id).call()
}

// Call the function and do something with the result:
getZombieDetails(15)
.then(function(result) {
  console.log("Zombie 15: " + JSON.stringify(result));
});
```

Let's walk through what's happening here.

cryptoZombies.methods.zombies(id).call() will communicate with the Web3 provider node and tell it to return the zombie with index id from **Zombie[] public zombies** on our contract.

Note that this is **asynchronous**, like an API call to an external server. So Web3 returns a promise here. (If you're not familiar with JavaScript promises... Time to do some additional homework before continuing!)

Once the promise resolves (which means we got an answer back from the web3 provider), our example code continues with the then statement, which logs result to the console.

result will be a JavaScript object that looks like this:

```
{
  "name": "H4XF13LD MORRIS'S COOLER OLDER BROTHER",
  "dna": "1337133713371337",
  "level": "9999",
  "readyTime": "1522498671",
  "winCount": "999999999",
  "lossCount": "0" // Obviously.
}
```

MEHMOOD SHEIKH

We could then have some front-end logic to parse this object and display it in a meaningful way on the front-end.

METAMASK & ACCOUNTS

Now let's put some pieces together — let's say we want our app's homepage to display a user's entire zombie army.

Obviously we'd first need to use our function **getZombiesByOwner(owner)** to look up all the IDs of zombies the current user owns.

But our Solidity contract is expecting owner to be a Solidity address. How can we know the address of the user using our app?

Getting the user's account in MetaMask

MetaMask allows the user to manage multiple accounts in their extension.

We can see which account is currently active on the injected web3 variable via:

```
var userAccount = web3.eth.accounts[0]
```

Because the user can switch the active account at any time in MetaMask, our app needs to monitor this variable to see if it has changed and update the UI accordingly. For example, if the user's homepage displays their zombie army, when they change their account in MetaMask, we'll want to update the page to show the zombie army for the new account they've selected.

We can do that with a **setInterval** loop as follows:

```
var accountInterval = setInterval(function() {
  // Check if account has changed
  if (web3.eth.accounts[0] !== userAccount) {
    userAccount = web3.eth.accounts[0];
    // Call some function to update the UI with the new account
    updateInterface();
  }
}, 100);
```

What this does is check every 100 milliseconds to see if userAccount is still equal web3.eth.accounts[0] (i.e. does the user still have that account active). If not, it reassigns userAccount to the currently active account, and calls a function to update the display.

MEHMOOD SHEIKH