# Lesson 2

## So, you've made it to Lesson 2!

Impressive, human! You're a better coder than I thought.

Lesson 2 will teach you how to **multiply your zombie army by feeding on other lifeforms**.

In this lesson we will cover some more advanced Solidity concepts, so it's highly recommended that you complete Lesson 1 before starting.

Are you ready to get started?

## Chapter 1: Lesson 2 Overview

In lesson 1, we created a function that takes a name, uses it to generate a random zombie, and adds that zombie to our app's zombie database on the blockchain.

In lesson 2, we're going to make our app more game-like: We're going to make it multi-player, and we'll also be adding a more fun way to create zombies instead of just generating them randomly.

How will we create new zombies? By having our zombies "feed" on other lifeforms!

### Zombie Feeding

When a zombie feeds, it infects the host with a virus. The virus then turns the host into a new zombie that joins your army. The new zombie's DNA will be calculated from the previous zombie's DNA and the host's DNA.

And what do our zombies like to feed on most?

To find that out... You'll have to complete lesson 2!

## Put it to the test

There's a simple demo of feeding to the right. Click on a human to see what happens when your zombie feeds!

You can see that the new zombie's DNA is determined by your original zombie's DNA, as well as the host's DNA.

When you're ready, click "Next chapter" to move on, and let's get started by making our game multi-player.

Check Answer

## Chapter 2: Mappings and Addresses

Let's make our game multi-player by giving the zombies in our database an owner.

To do this, we'll need 2 new data types: `mapping` and `address`.

### 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`

(This address belongs to the CryptoZombies team. If you're enjoying CryptoZombies, you can send us some Ether! 😉 )

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

In Lesson 1 we looked at **structs** and **arrays**. **Mappings** are another way of storing organized data in Solidity.

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;
```

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`.

## Put it to the test

To store zombie ownership, we're going to use two mappings: one that keeps track of the address that owns a zombie, and another that keeps track of how many zombies an owner has.

1. Create a mapping called `zombieToOwner`. The key will be a `uint` (we'll store and look up the zombie based on its id) and the value an `address`. Let's make this mapping `public`.

2. Create a mapping called `ownerZombieCount`, where the key is an `address` and the value a `uint`.

```
pragma solidity ^0.4.19;

contract ZombieFactory {

    event NewZombie(uint zombieId, string name, uint dna);

    uint dnaDigits = 16;
    uint dnaModulus = 10 ** dnaDigits;

    struct Zombie {
        string name;
        uint dna;
    }

    Zombie[] public zombies;

    mapping (uint => address) public zombieToOwner;
    mapping (address => uint) ownerZombieCount;

    function _createZombie(string _name, uint _dna) private {
        uint id = zombies.push(Zombie(_name, _dna)) - 1;
        // start here
        NewZombie(id, _name, _dna);
    }

    function _generateRandomDna(string _str) private view returns (uint) {
        uint rand = uint(keccak256(_str));
        return rand % dnaModulus;
    }

    function createRandomZombie(string _name) public {
        uint randDna = _generateRandomDna(_name);
        _createZombie(_name, randDna);
    }

}
```

# Chapter 3: 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.

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.

## Put it to the test

Let's update our `_createZombie` method from lesson 1 to assign ownership of the zombie to whoever called the function.

1. First, after we get back the new zombie's `id`, let's update our `zombieToOwner` mapping to store `msg.sender` under that `id`.
2. Second, let's increase `ownerZombieCount` for this `msg.sender`.

In Solidity, you can increase a `uint` with `++`, just like in javascript:

```
uint number = 0;
number++;
// `number` is now `1`
```

Your final answer for this chapter should be 2 lines of code.

## Chapter 4: Require

In lesson 1, we made it so users can create new zombies by calling `createRandomZombie` and entering a name. However, if users could keep calling this function to create unlimited zombies in their army, the game wouldn't be very fun.

Let's make it so each player can only call this function once. That way new players will call it when they first start the game in order to create the initial zombie in their army.

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 _name) public returns (string) {
  // 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(_name) == keccak256("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.

## Put it to the test

In our zombie game, we don't want the user to be able to create unlimited zombies in their army by repeatedly calling `createRandomZombie` — it would make the game not very fun.

Let's use `require` to make sure this function only gets executed one time per user, when they create their first zombie.

1. Put a `require` statement at the beginning of `createRandomZombie`. The function should check to make sure `ownerZombieCount[msg.sender]` is equal to `0`, and throw an error otherwise.

*Note: In Solidity, it doesn't matter which term you put first — both orders are equivalent. However, since our answer checker is really basic, it will only accept one answer as correct — it's expecting `ownerZombieCount[msg.sender]` to come first.*

## Chapter 5: Inheritance

Our game code is getting quite long. Rather than making one extremely long contract, sometimes it makes sense to split your code logic across multiple contracts to organize the code.

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

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

contract BabyDoge is Doge {
  function anotherCatchphrase() public returns (string) {
    return "Such Moon BabyDoge";
  }
}
```

`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.

## Put it to the test

In the next chapters, we're going to be implementing the functionality for our zombies to feed and multiply. Let's put this logic into its own contract that inherits all the methods from `ZombieFactory`.

1. Make a contract called `ZombieFeeding` below `ZombieFactory`. This contract should inherit from our `ZombieFactory` contract.

```
2. pragma solidity ^0.4.19;
3.
4. contract ZombieFactory {
5.
6.     event NewZombie(uint zombieId, string name, uint dna);
7.
8.     uint dnaDigits = 16;
9.     uint dnaModulus = 10 ** dnaDigits;
10.
11.    struct Zombie {
12.        string name;
13.        uint dna;
14.    }
15.
16.    Zombie[] public zombies;
17.
18.    mapping (uint => address) public zombieToOwner;
19.    mapping (address => uint) ownerZombieCount;
20.
21.    function _createZombie(string _name, uint _dna) private {
22.        uint id = zombies.push(Zombie(_name, _dna)) - 1;
23.        zombieToOwner[id] = msg.sender;
24.        ownerZombieCount[msg.sender]++;
25.        NewZombie(id, _name, _dna);
26.    }
27.
28.    function _generateRandomDna(string _str) private view returns (uint) {
29.        uint rand = uint(keccak256(_str));
30.        return rand % dnaModulus;
31.    }
32.
33.    function createRandomZombie(string _name) public {
34.        require(ownerZombieCount[msg.sender] == 0);
35.        uint randDna = _generateRandomDna(_name);
36.        _createZombie(_name, randDna);
37.    }
38.
39.}
40.
```

# Chapter 6: Import

Whoa! You'll notice we just cleaned up the code to the right, and you now have tabs at the top of your editor. Go ahead, click between the tabs to try it out.

Our code was getting pretty long, so we split it up into multiple files to make it more manageable. This is normally how you will handle long codebases in your Solidity projects.

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.

## Put it to the test

Now that we've set up a multi-file structure, we need to use `import` to read the contents of the other file:

1. Import `zombiefactory.sol` into our new file, `zombiefeeding.sol`.

```
pragma solidity ^0.4.19;

// put import statement here

contract ZombieFeeding is ZombieFactory {

}
```

# Chapter 7: Storage vs Memory

In Solidity, there are two places 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;
  }

  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.
  }
}
```

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`!

## Put it to the test

It's time to give our zombies the ability to feed and multiply!

When a zombie feeds on some other lifeform, its DNA will combine with the other lifeform's DNA to create a new zombie.

1. Create a function called `feedAndMultiply`. It will take two parameters: `_zombieId` (a `uint`) and `_targetDna` (also a `uint`). This function should be `public`.
2. We don't want to let someone else feed using our zombie! So first, let's make sure we own this zombie. Add a `require` statement to make sure `msg.sender` is equal to this zombie's owner (similar to how we did in the `createRandomZombie` function).

    *Note: Again, because our answer-checker is primitive, it's expecting `msg.sender` to come first and will mark it wrong if you switch the order. But normally when you're coding, you can use whichever order you prefer — both are correct.*

3. We're going to need to get this zombie's DNA. So the next thing our function should do is declare a local `Zombie` named `myZombie` (which will be a `storage` pointer). Set this variable to be equal to index `_zombieId` in our `zombies` array.

You should have 4 lines of code so far, including the line with the closing `}`.

We'll continue fleshing out this function in the next chapter!

# Chapter 7: Storage vs Memory

In Solidity, there are two places 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;
  }

  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.
  }
}
```

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`!

## Put it to the test

It's time to give our zombies the ability to feed and multiply!

When a zombie feeds on some other lifeform, its DNA will combine with the other lifeform's DNA to create a new zombie.

1. Create a function called `feedAndMultiply`. It will take two parameters: `_zombieId` (a `uint`) and `_targetDna` (also a `uint`). This function should be `public`.
2. We don't want to let someone else feed using our zombie! So first, let's make sure we own this zombie. Add a `require` statement to make sure `msg.sender` is equal to this zombie's owner (similar to how we did in the `createRandomZombie` function).

    *Note: Again, because our answer-checker is primitive, it's expecting `msg.sender` to come first and will mark it wrong if you switch the order. But normally when you're coding, you can use whichever order you prefer — both are correct.*

3. We're going to need to get this zombie's DNA. So the next thing our function should do is declare a local `Zombie` named `myZombie` (which will be a `storage` pointer). Set this variable to be equal to index `_zombieId` in our `zombies` array.

You should have 4 lines of code so far, including the line with the closing `}`.

We'll continue fleshing out this function in the next chapter!

# Chapter 8: Zombie DNA

Let's finish writing the `feedAndMultiply` function.

The formula for calculating a new zombie's DNA is simple: It's simply that average between the feeding zombie's DNA and the target's DNA.

For example:

```
function testDnaSplicing() public {
  uint zombieDna = 2222222222222222;
  uint targetDna = 4444444444444444;
  uint newZombieDna = (zombieDna + targetDna) / 2;
  // ^ will be equal to 3333333333333333
}
```

Later we can make our formula more complicated if we want to, like adding some randomness to the new zombie's DNA. But for now we'll keep it simple — we can always come back to it later.

## Put it to the test

1. First we need to make sure that `_targetDna` isn't longer than 16 digits. To do this, we can set `_targetDna` equal to `_targetDna % dnaModulus` to only take the last 16 digits.
2. Next our function should declare a `uint` named `newDna`, and set it equal to the average of `myZombie`'s DNA and `_targetDna` (as in the example above).

> *Note: You can access the properties of `myZombie` using `myZombie.name` and `myZombie.dna`*

3. Once we have the new DNA, let's call `_createZombie`. You can look at the `zombiefactory.sol` tab if you forget which parameters this function needs to call it. Note that it requires a name, so let's set our new zombie's name to `"NoName"` for now — we can write a function to change zombies' names later.

*Note: For you Solidity whizzes, you may notice a problem with our code here! Don't worry, we'll fix this in the next chapter ;)*

# Chapter 9: More on Function Visibility

**The code in our previous lesson has a mistake!**

If you try compiling it, the compiler will throw an error.

The issue is we tried calling the `_createZombie` function from within `ZombieFeeding`, but `_createZombie` is a `private` function inside `ZombieFactory`. This means none of the contracts that inherit from `ZombieFactory` can access it.

## Internal and External

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

`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` 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) {
    baconSandwichesEaten++;
    // We can call this here because it's internal
    eat();
  }
}
```

## Put it to the test

1. Change `_createZombie()` from `private` to `internal` so our other contract can access it.

We've already focused you back to the proper tab, `zombiefactory.sol`.

```
pragma solidity ^0.4.19;

contract ZombieFactory {

    event NewZombie(uint zombieId, string name, uint dna);

    uint dnaDigits = 16;
    uint dnaModulus = 10 ** dnaDigits;

    struct Zombie {
        string name;
        uint dna;
    }

    Zombie[] public zombies;

    mapping (uint => address) public zombieToOwner;
    mapping (address => uint) ownerZombieCount;

    // edit function definition below
    function _createZombie(string _name, uint _dna) private {
        uint id = zombies.push(Zombie(_name, _dna)) - 1;
        zombieToOwner[id] = msg.sender;
        ownerZombieCount[msg.sender]++;
        NewZombie(id, _name, _dna);
    }

    function _generateRandomDna(string _str) private view returns (uint) {
        uint rand = uint(keccak256(_str));
        return rand % dnaModulus;
    }

    function createRandomZombie(string _name) public {
        require(ownerZombieCount[msg.sender] == 0);
        uint randDna = _generateRandomDna(_name);
        _createZombie(_name, randDna);
    }

}
```

# Chapter 10: What Do Zombies Eat?

It's time to feed our zombies! And what do zombies like to eat most?

Well it just so happens that CryptoZombies love to eat...

**CryptoKitties!** 😱😱😱

(Yes, I'm serious 😆 )

In order to do this we'll need to read the kittyDna from the CryptoKitties smart contract. We can do that because the CryptoKitties data is stored openly on the blockchain. Isn't the blockchain cool?!

Don't worry — our game isn't actually going to hurt anyone's CryptoKitty. We're only *reading* the CryptoKitties data, we're not able to actually delete it 😉

## 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.

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);
    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.

Now that we know what this function looks like, we can use it to create an interface:

1. Define an interface called `KittyInterface`. Remember, this looks just like creating a new contract — we use the `contract` keyword.
2. Inside the interface, define the function `getKitty` (which should be a copy/paste of the function above, but with a semi-colon after the `returns` statement, instead of everything inside the curly braces.

```
pragma solidity ^0.4.19;

import "./zombiefactory.sol";

// Create KittyInterface here

contract ZombieFeeding is ZombieFactory {

  function feedAndMultiply(uint _zombieId, uint _targetDna) public {
    require(msg.sender == zombieToOwner[_zombieId]);
    Zombie storage myZombie = zombies[_zombieId];
    _targetDna = _targetDna % dnaModulus;
    uint newDna = (myZombie.dna + _targetDna) / 2;
    _createZombie("NoName", newDna);
  }

}
```

## Chapter 11: 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
  }
}
```

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`.

## Put it to the test

Let's set up our contract to read from the CryptoKitties smart contract!

1. I've saved the address of the CryptoKitties contract in the code for you, under a variable named `ckAddress`. In the next line, create a `KittyInterface` named `kittyContract`, and initialize it with `ckAddress` — just like we did with `numberContract` above.

```
pragma solidity ^0.4.19;

import "./zombiefactory.sol";

contract KittyInterface {
  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
  );
}

contract ZombieFeeding is ZombieFactory {

  address ckAddress = 0x06012c8cf97BEaD5deAe237070F9587f8E7A266d;
  // Initialize kittyContract here using `ckAddress` from above

  function feedAndMultiply(uint _zombieId, uint _targetDna) public {
    require(msg.sender == zombieToOwner[_zombieId]);
    Zombie storage myZombie = zombies[_zombieId];
    _targetDna = _targetDna % dnaModulus;
    uint newDna = (myZombie.dna + _targetDna) / 2;
    _createZombie("NoName", newDna);
  }

}
```

## Chapter 12: 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();
}
```

## Put it to the test

Time to interact with the CryptoKitties contract!

Let's make a function that gets the kitty genes from the contract:

1. Make a function called `feedOnKitty`. It will take 2 `uint` parameters, `_zombieId` and `_kittyId`, and should be a `public` function.
2. The function should first declare a `uint` named `kittyDna`.

> *Note: In our `KittyInterface`, `genes` is a `uint256` — but if you remember back to lesson 1, `uint` is an alias for `uint256` — they're the same thing.*

3. The function should then call the `kittyContract.getKitty` function with `_kittyId` and store `genes` in `kittyDna`. Remember — `getKitty`returns a ton of variables. (10 to be exact — I'm nice, I counted them for you!). But all we care about is the last one, `genes`. Count your commas carefully!
4. Finally, the function should call `feedAndMultiply`, and pass it both `_zombieId` and `kittyDna`.

## Chapter 13: Bonus: Kitty Genes

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.

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 sandwich) public {
  // Remember with strings, we have to compare their keccak256 hashes
```

```
  // to check equality
  if (keccak256(sandwich) == keccak256("BLT")) {
    eat();
  }
}
```

## Put it to the test

Let's implement cat genes in our zombie code.

1. First, let's change the function definition for `feedAndMultiply` so it takes a 3rd argument: a `string` named `_species`
2. Next, after we calculate the new zombie's DNA, let's add an `if`statement comparing the `keccak256` hashes of `_species` and the string `"kitty"`
3. Inside the `if` statement, we want to replace the last 2 digits of DNA with `99`. One way to do this is using the logic: `newDna = newDna - newDna % 100 + 99;`.

    *Explanation: Assume `newDna` is `334455`. Then `newDna % 100` is `55`, so `newDna - newDna % 100` is `334400`. Finally add `99` to get `334499`.*

4. Lastly, we need to change the function call inside `feedOnKitty`. When it calls `feedAndMultiply`, add the parameter `"kitty"` to the end.

```
pragma solidity ^0.4.19;

import "./zombiefactory.sol";

contract KittyInterface {
  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
  );
}

contract ZombieFeeding is ZombieFactory {

  address ckAddress = 0x06012c8cf97BEaD5deAe237070F9587f8E7A266d;
```

```
  KittyInterface kittyContract = KittyInterface(ckAddress);

  // Modify function definition here:
  function feedAndMultiply(uint _zombieId, uint _targetDna) public {
    require(msg.sender == zombieToOwner[_zombieId]);
    Zombie storage myZombie = zombies[_zombieId];
    _targetDna = _targetDna % dnaModulus;
    uint newDna = (myZombie.dna + _targetDna) / 2;
    // Add an if statement here
    _createZombie("NoName", newDna);
  }

  function feedOnKitty(uint _zombieId, uint _kittyId) public {
    uint kittyDna;
    (,,,,,,,,,kittyDna) = kittyContract.getKitty(_kittyId);
    // And modify function call here:
    feedAndMultiply(_zombieId, kittyDna);
  }

}
```

## Chapter 14: Wrapping It Up

That's it, you've completed lesson 2!

You can check out the demo to the right to see it in action. Go ahead, I know you can't wait until the bottom of this page 😉. Click a kitty to attack, and see the new kitty zombie you get!

# 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.
// 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)
})
```

## Give it a try!

Select the kitty you want to feed on. Your zombie's DNA and the kitty's DNA will combine, and you'll get a new zombie in your army!

Notice those cute cat legs on your new zombie? That's our final `99` digits of DNA at work 😉

You can start over and try again if you want. When you get a kitty zombie you're happy with (you only get to keep one), go ahead and proceed to the next chapter to complete lesson 2!

```
pragma solidity ^0.4.19;

import "./zombiefactory.sol";

contract ZombieFeeding is ZombieFactory {

  function feedAndMultiply(uint _zombieId, uint _targetDna) public {
    require(msg.sender == zombieToOwner[_zombieId]);
    Zombie storage myZombie = zombies[_zombieId];
    // start here
  }

}
```


Your Turn
+7
+5
+5
OPEN