2. Functions are Objects

* These two pillars are unique to JS no language has it, that’s why JS is so powerful.
* functions() and arrays are objects.

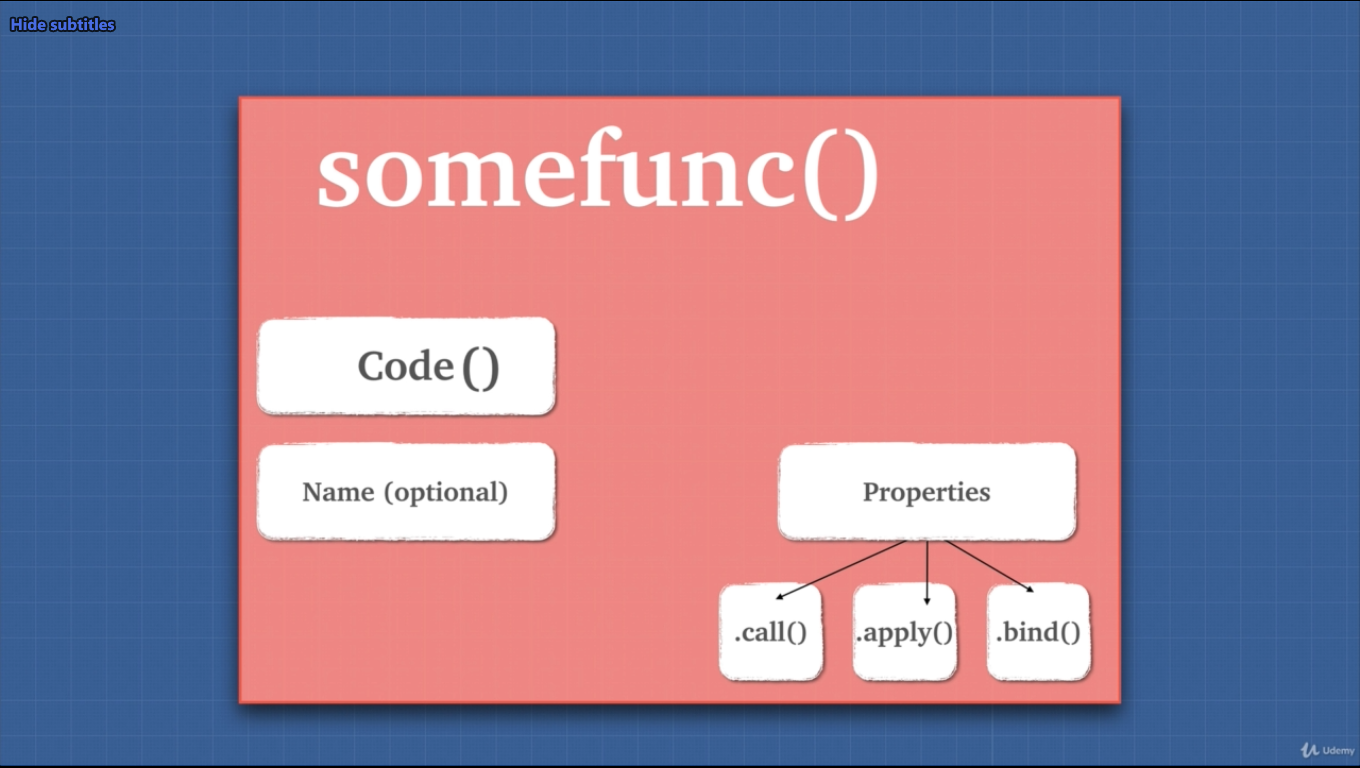
functions():

* When we invoke or call a function, we get this and argument keyword.
* argument is an array like object that has some wired behavior for looping and iteration.
* We can grab parameters using argument keyword despite of not defining any parameter at first. Using this (…args).
* When we define our function, compiler looks at our code lexically to determine what variable we have access to in our variable environment. It also add scope chain.

Ways of invoking a function:

1. Normal way, foo().
2. Inside of an object, a method(), obj.foo(). In this case the this keyword will be updated to the current object.
3. Using call() and apply() method, foo.call().
4. Using the built-in object Function(), it’s a constructor. It also accepts parameter.

* In JS functions are objects it is not common in other languages.
* We can move them around, store them as data.



3. First Class Citizens

* functions are first class citizens is JS.

Behavior of functions:

1. Assigning to a variable or a method inside of an object.
2. We can pass a function as a parameter.
3. We can return functions as values from other functions.

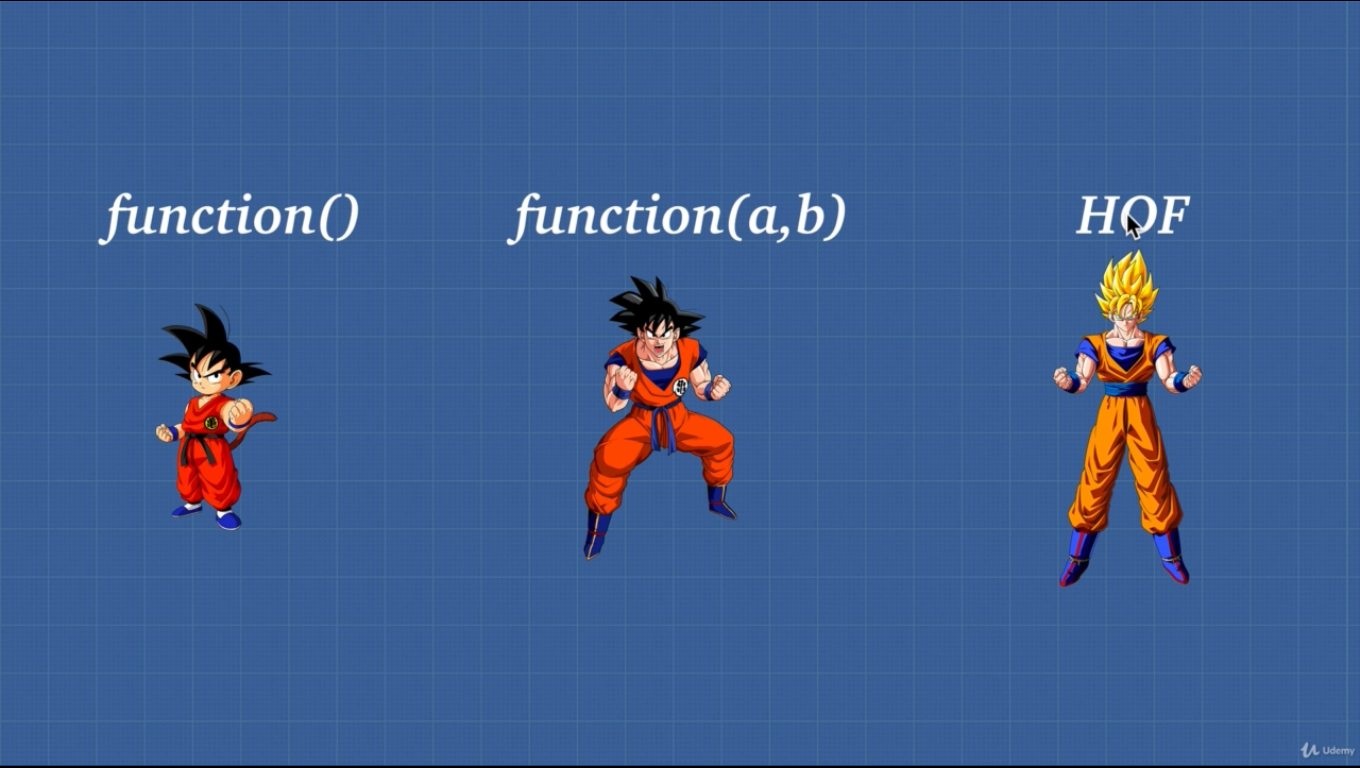
These behaviors makes functions as if they are JS types. Anything we can do with other types we can do with functions. This idea of first-class citizen property, makes JS a functional programming language.

4. Extra Bits Functions

Careful using function:

* Don’t initialize function inside of a loop, instead we need to move them somewhere else.
* Is variable accessible or not. To avoid this, we can check if variable is present or give a default param (ES6).

5. Higher Order Functions (HOF)



* HOF is a function that can take another function as an argument or a function that returns another function.

Three types of function use cases:

1. A normal function that is, every time we tell it what to do and it’s not dry.
2. Function with param that we tell the function what data to use later on. It can also be not dry.
3. HOF which can do both of these things and takes and/or returns another function.

6. Exercise Higher Order Functions

* Arrow function is cleaner visually.

7. Closures

Two things for closures:

1. In JS functions are first class citizen (HOF).
2. Lexical scope, where the code is written.

Closures is combination of function and the lexical environment from where it was declared.

* Closures is one of the most confusing things but important topic.
* When we execute a function, it is popped off from the stack and its variable environment is cleared.
  + Even though the function is popped off form the stack and variable environment is cleared, since another function is referencing some variables of the cleared function, closures takes the variables and stores them in the heap, so that the referencing function can have the variables that it needs.
  + JS engine will keep anything that is being refenced, inside of the closures.
* Closures are also called lexical scoping.
  + Lexical, where it is written.
  + Scoping, what variable it has access to.
* JS is lexically/statically scoped.

8. Exercise Closures

* If we use setTimeout() closures will work.
* Even if we declare variable after the setTimeout() closures will have the variable for us.

9. Closures and Memory

Two benefits of closures:

1. Memory efficient.
2. Encapsulation.

Memory efficiency:

* To use memory efficiency, we need to assign the function to a variable and call it later.

10. Closures and Encapsulation

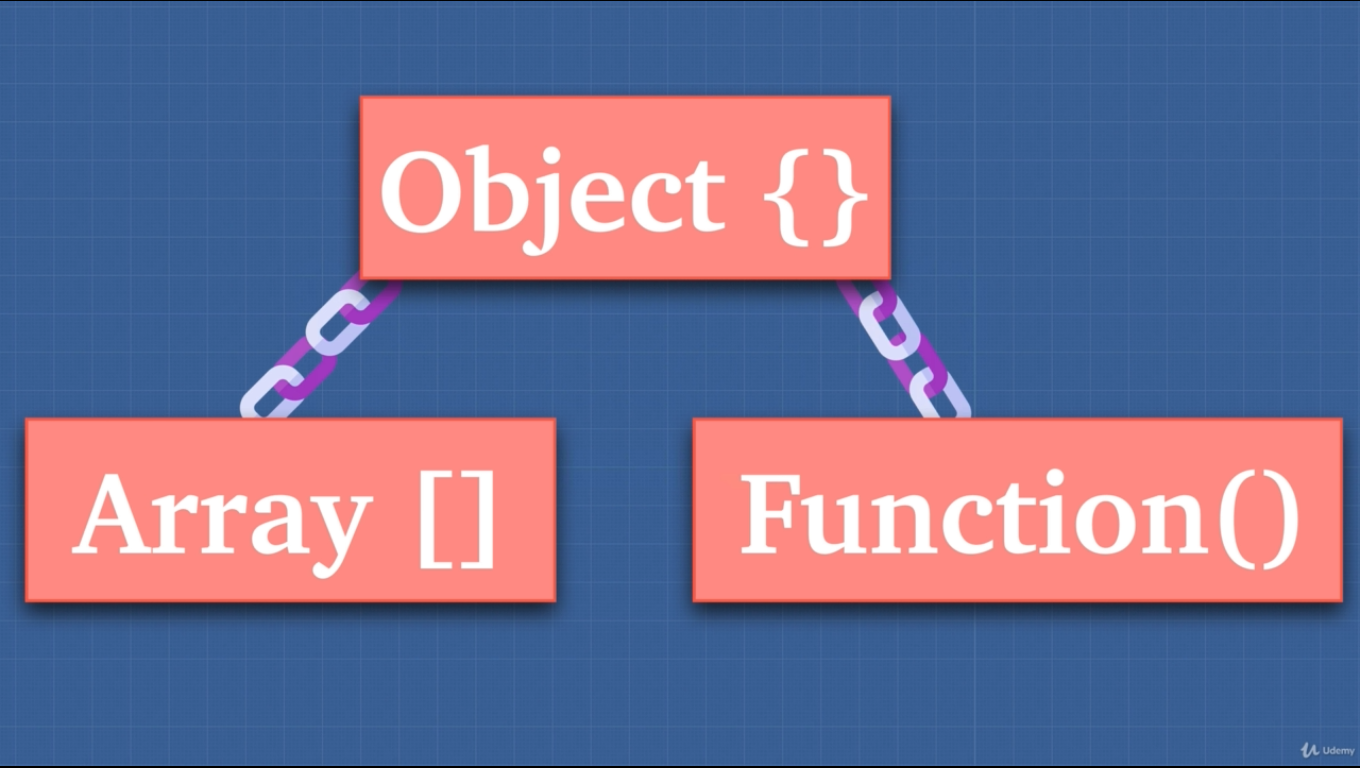
* We can return object, and function will have nested functions to be accessed with dot.
* Least privilege principle reduces the chance to unwanted attacks. Users should have the bare minimum amount of data access.

15. Closures Review

*Statement*: Closures allow a function to access variables from an enclosing scope or outer scope environment even after it leaves the scope in which it was declared.

16. Prototypal Inheritance

* JS uses prototypal inheritance.
* Inheritance is object getting access of the properties and methods of another object.
* array.\_\_proto\_\_ goes up the prototypal chain. If we do multiple time we can get to the base object.
  + In JS from base object everything gets created.
* An object gets access of properties and methods of another object through the prototype chain.
  + something.\_\_proto\_\_ can give us the base object or a method form which we are making stuff.
* It is unique to JS. Not all popular language has it.



17. Prototypal Inheritance 2

* We can borrow a method from another object using .bind().
* But manually borrowing stuff it can get complicated, that’s why we can use prototypal inheritance.
  + obj1.\_\_ptoto\_\_ = obj2.
* We will get access to another object property but native property will be intact.

18. Prototypal Inheritance 3

* We are not coping the property of another object JS just goes up the chain and finds it for us.
* We don’t need to say \_\_proto\_\_ multiple times because JS automatically goes up and does its thing.

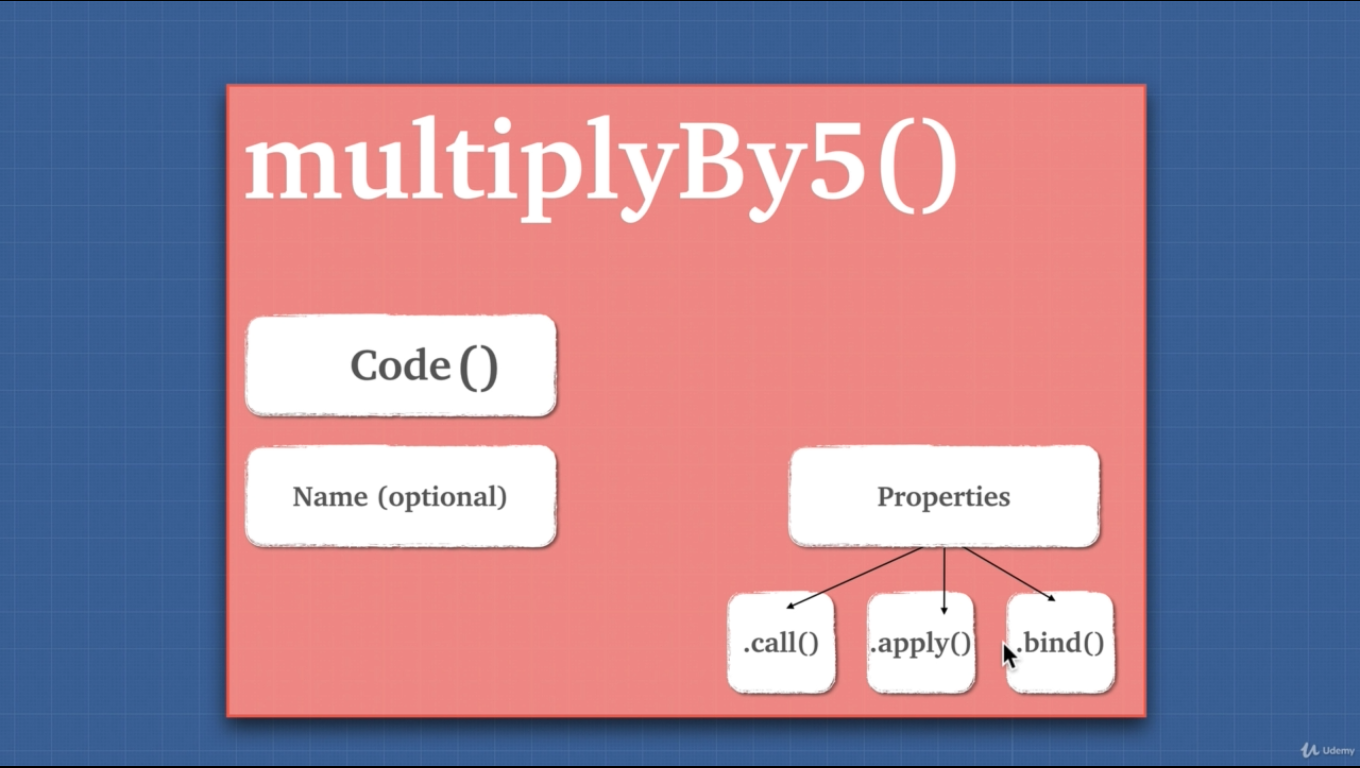
Warning: we should never use \_\_proto\_\_ . Bad performance.

* This \_\_proto\_\_ is useful because it does not copy stuffs it just use the first instance of code.

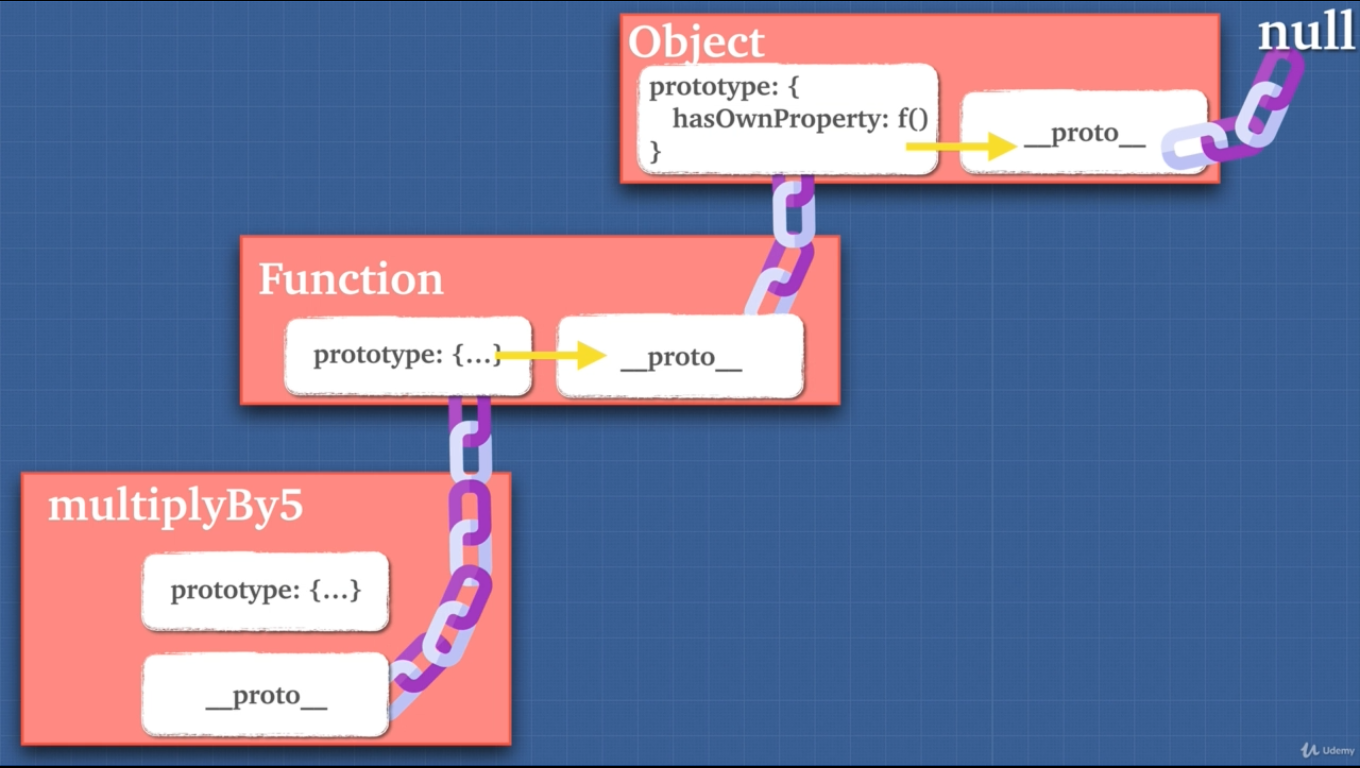
19. Prototypal Inheritance 4

Function:

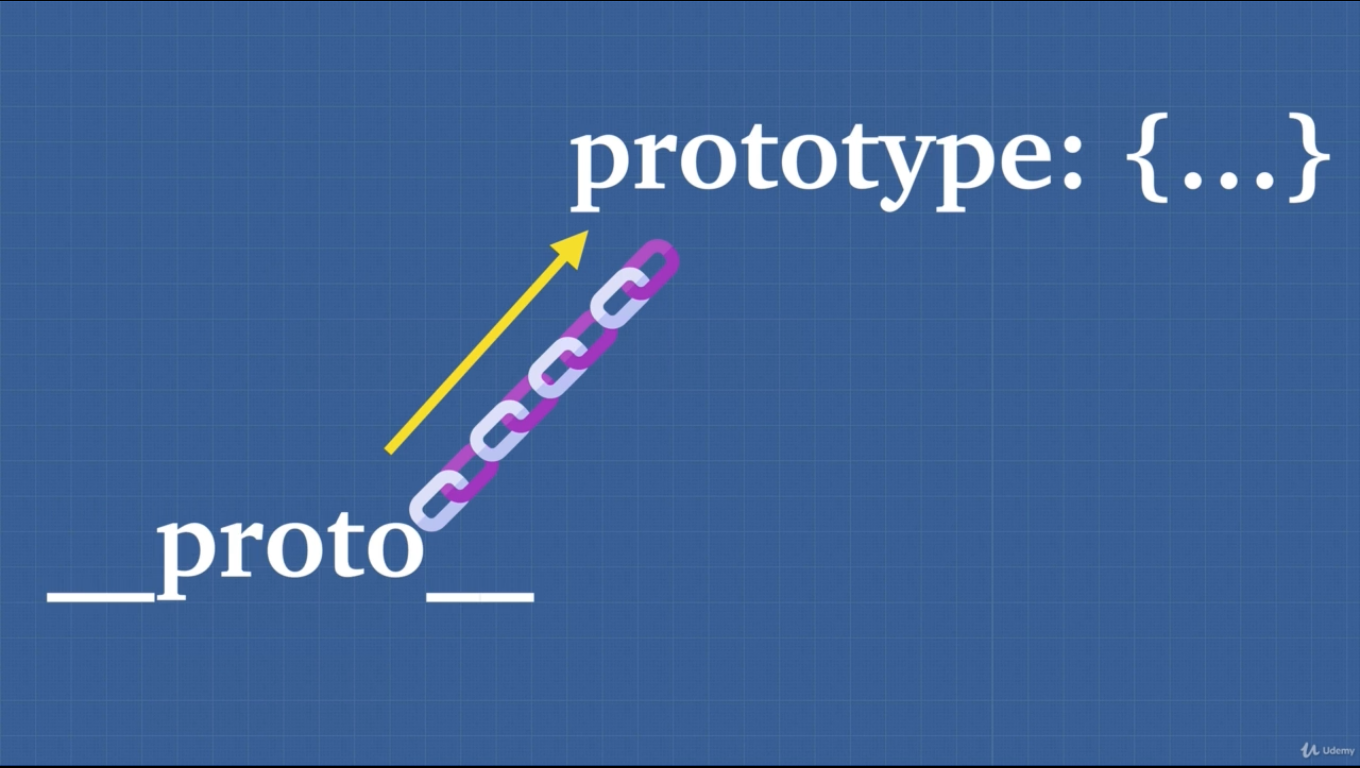
* Function is a callable object that can be invoked.
* Call(), apply(), bind() property.
* It has optional name property.
* We can add property just like an object.

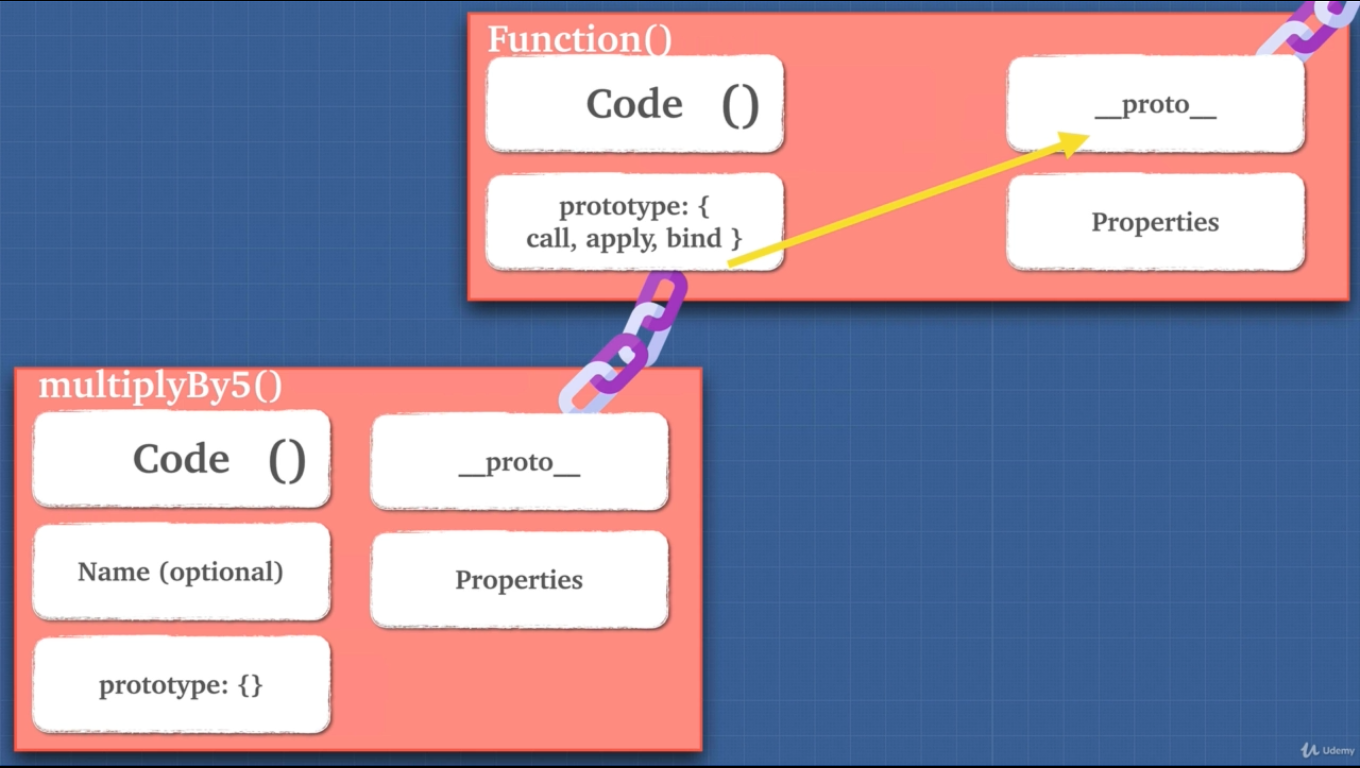


Now this is not accurate.



This is accurate.





(Yellow line \_\_proto\_\_ lives with call, apply, bind)

* \_\_proto\_\_ is a pointer which points to native Function, Array, object.
  + array.\_\_proto\_\_ points to Array.prototype (father).
* Pointer This lets us use to be efficient with our memory, we are able to use map, foreach function. We just write them once and not coping them around.

20. Prototypal Inheritance 5

* One of the safe ways to do \_\_proto\_\_:
  + let newton = Object.create(human);

21. Prototypal Inheritance 6

***Statement***: Only functions have prototype property.

* Oh JS! typeof Object gives “function”. 😐
  + It shows function because it has prototype property. (only function has it)
* Built-in Object constructor creates an object wrapper.
  + typeof {} gives object.
* Main take way is every function has a prototype property and it references to an object used to attach properties that will be inherited by object further down. Last object is the built-in object.ptototype.
* Object is a function and Object.prototype is the base object.
* “string”.toString try to access a method of string, underneath the hood JS converts it into String object (String.prototype), so that JS can look up the prototype chain and look for that method.

Review:

* Everything in JS is an object.
  + Array and function are objects they inherit through the prototype chain from the base object.
* Going up with prototype chain we have properties that we can use. These properties are located inside of prototype property. Prototype property has \_\_proto\_\_ property.
  + chain 🡪 prototype:{...} > \_\_proto\_\_ 🡪 chain 🡪 prototype:{}
  + \_\_Proto\_\_ 🡪 prototype:{...}
* Only functions have the prototype property.
* With prototype we avoid repeating code and efficient memory use.

23. Solution Prototypal Inheritance

* Extending the built-in object is a bad thing unless we make sure that names of the methods will never be overridden in vanilla JS.
* Arrow function will not work in extending functionality.
  + In arrow function this is lexically scoped.
  + This is a rare case, we want this to be determined at call time, when we actually run the function.
  + In this case we want to keep the dynamic scoping of this keyword.
* Whatever that is left of the method is this. Whatever called it.