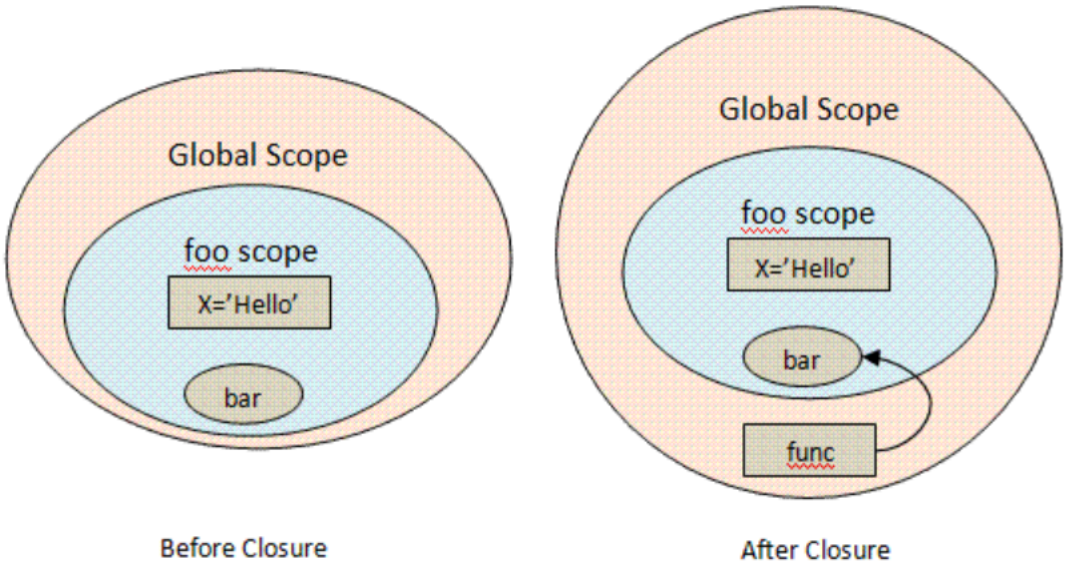
Closures

Functions that retain access to outer variables even after the outer function has finished executing.

Closures capture and preserve lexical scope.



Examples of Closures:

Example 1:

```
function Outer(num){
    let i =0;
    return function(addition){
      i++;
      return num + addition + i
    }
  }

  const myFunc=Outer(8)
  console.log(myFunc(4))
  console.log(myFunc(5))
```

Example 2: Event handling with closures:

```
function createButton() {

  const button = document.createElement("button");

  button.addEventListener("click", function () {

    console.log("Button clicked!");

  });

  return button;

}

const button = createButton();

document.body.appendChild(button);
```

Advantages that closure brings with it:

1. Encapsulation with Closures:
    - Closures create a private scope for variables and functions within an outer function, ensuring encapsulation.
    - Variables defined within the outer function are not accessible from outside the closure.
    - Inner functions have access to the outer variables even after the outer function has finished executing.
    - Example: Creating private variables in a closure:

```
function createPerson(name) {

let age = 0;

  return {

    getName: function () {

      return name;

    },

    getAge: function () {

      return age;

    },

    setAge: function (newAge) {

      age = newAge;

    },

  };

}

const person = createPerson("John");

console.log(person.getName()); // Output: John
console.log(person.getAge()); // Output: 0 person.setAge(30);
console.log(person.getAge()); // Output: 30
```

2. Closures are memory efficient

- When a closure is created, it captures the necessary variables by reference, rather than making copies of them. This means that closures can access variables without duplicating their memory footprint, thus saving memory.

```
function someHeavyWork(i){
    const arr=new
Array(2000).fill(Math.round(Math.random()*100));
    return arr[i]
  }
  console.log(someHeavyWork(4))
  console.log(someHeavyWork(8))
  console.log(someHeavyWork(478))

  function someHeavyWork2(){
     const arr=new
Array(2000).fill(Math.round(Math.random()*100));
    return function(i){
      return arr[i]
    }
  }

  const myFunc=someHeavyWork2()
  console.log(myFunc(6))
  console.log(myFunc(46))
  console.log(myFunc(68))
```

Prototypes

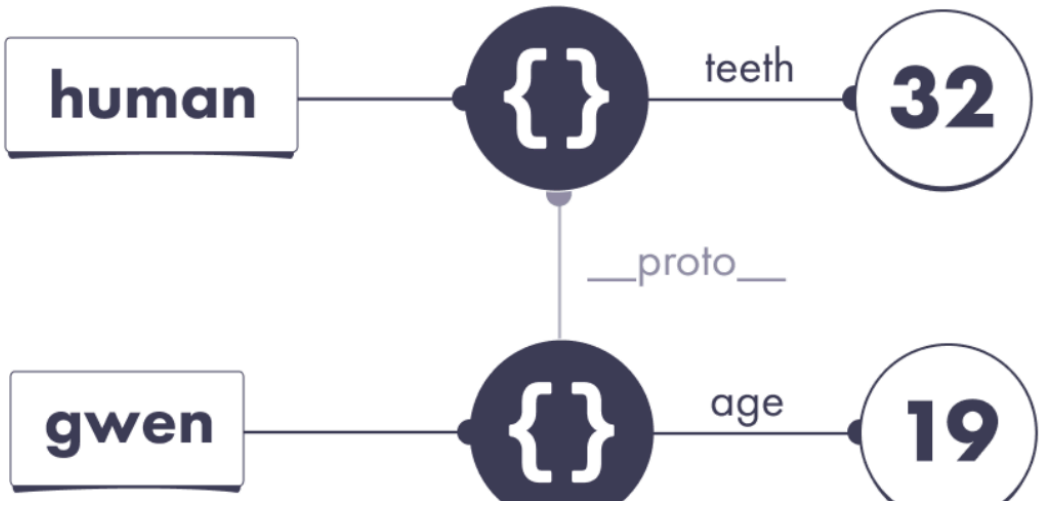
In Javascript, Prototypes way objects can share properties and methods with each other.

```
let human = {
  teeth: 32,
};
console.log(human.age)
```

If we consider above example, ideally this piece of code should throw an error, but what if i tell u it does not.

```
let gwen = {
  // We added this line:
  __proto__: human,
  age: 19,
};
```

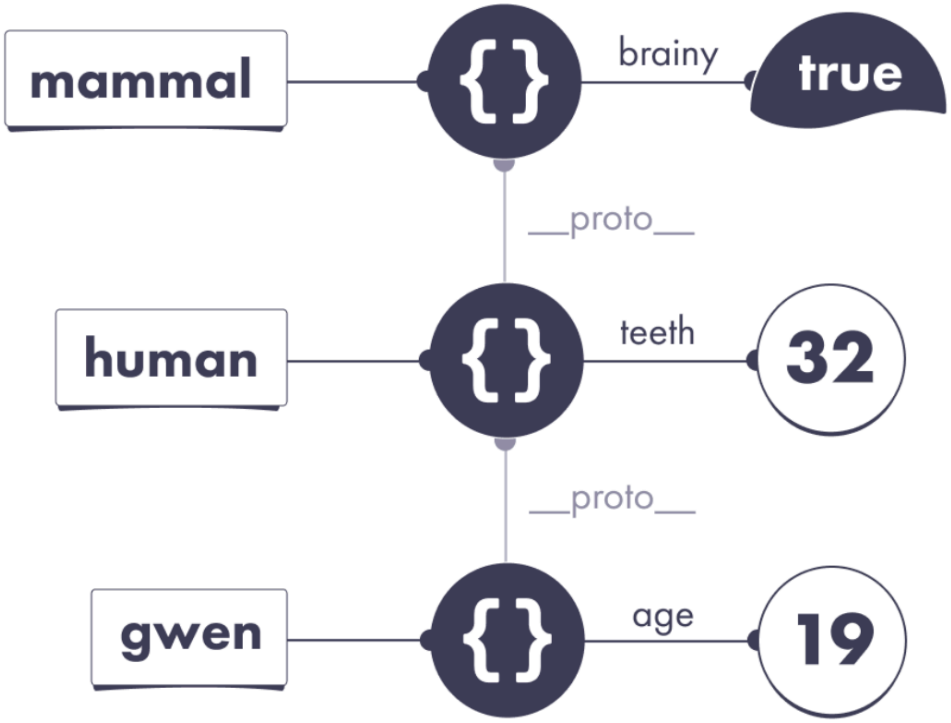
__proto__ represents the JavaScript concept of a prototype. Any JavaScript object may choose another object as a prototype. We will discuss what that means in practice but for now, let's think of it as a special __proto__ wire:



This is similar to saying, "I don't know, but Alice might know." With `__proto__`, you instruct JavaScript to "ask another object."

A prototype isn't a special "thing" in JavaScript. A prototype is more like a *relationship*. An object may point to another object as its prototype.

```
let mammal = {
  brainy: true,
};

let human = {
  __proto__: mammal,
  teeth: 32,
};

let gwen = {
  __proto__: human,
  age: 19,
};

console.log(gwen.brainy); // true
```



Advantages of Prototypes:

PolyFills:

// Check if Array.prototype.includes() is supported natively

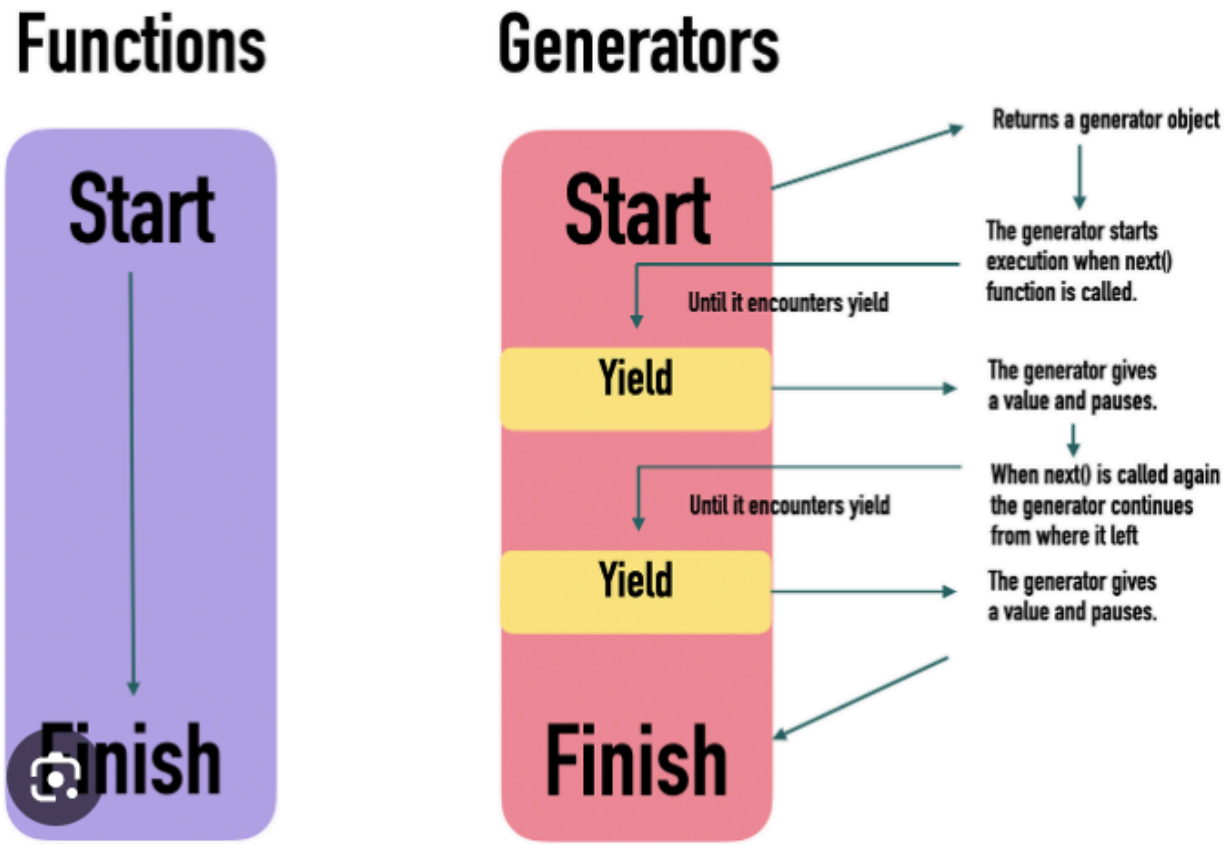
```
if (!Array.prototype.includes) {
  Array.prototype.includes = function(searchElement, fromIndex) {
    if (this == null) {
      throw new TypeError('Cannot convert undefined or null to object');
    }

    var array = Object(this);
    var len = array.length >>> 0;
    if (len === 0) {
      return false;
    }

    var n = fromIndex || 0;
    var k;
    if (n >= 0) {
      k = n;
    } else {
```

```
    k = len + n;
    if (k < 0) {
      k = 0;
    }
  }

  while (k < len) {
    var currentElement = array[k];
    if (searchElement === currentElement ||
        (searchElement !== searchElement && currentElement !== currentElement)) {
      return true;
    }
    k++;
  }

  return false;
};
}
```

Above is a simple polyfill for the Array.prototype.includes() method, which adds support for browsers that don't natively support it, such as older versions of Firefox and Chrome:

Generators

JavaScript generators can be understood as special functions that allow you to pause and resume their execution. Unlike regular functions that run to completion, generators provide a way to generate a sequence of values over time.



When you invoke a generator function, it doesn't immediately execute the code inside it. Instead, it returns a special iterator object, which you can use to control the execution of the generator. By calling the next() method on the iterator, you can instruct the generator to execute until it reaches a yield statement.

```
function* createFlow() {
  yield 4;
  yield 5;
  yield 6;
}
const returnNextElement = createFlow();
const element1 = returnNextElement.next();
const element2 = returnNextElement.next();
```

What do we hope returnNextElement.next() will

return? But how?

By calling the next() method on the iterator object, you instruct the generator to resume its execution until the next yield statement. In this case, the first yield 4; statement is encountered, and the generator pauses, returning an object with a value property of 4. This value is assigned to the variable element1.

****

you call next() on the iterator again. The generator continues its execution from where it left off, encountering the second yield 5; statement. The generator pauses again, returning an object with a value property of 5. This value is assigned to the variable element2.
At this point, the generator has yielded two values, and you can continue calling next() to retrieve the remaining values (if any) from the generator.
In this example, element1 will be { value: 4, done: false }, and element2 will be { value: 5, done: false }. The done property indicates whether the generator has finished generating values (false in this case) or if there are more values to come.

Generators use cases:

1. Generator Delegation:

Using yield* to delegate to another generator.
Example:

```
function* generatorOne() {

  yield 1;

  yield 2;

}

function* generatorTwo() {

  yield* generatorOne();

  yield 3;
```

```
}

const generator = generatorTwo();

console.log(generator.next()); // { value: 1, done: false }
console.log(generator.next()); // { value: 2, done: false }
console.log(generator.next()); // { value: 3, done: false }
console.log(generator.next()); // { value: undefined, done:
true }
```

2. Passing a value to a generator

```
function* createFlow() {

  const num = 10;

  const newNum = yield num;

  yield 5 + newNum;

  yield 6;

}

const returnNextElement = createFlow();

const element1 = returnNextElement.next(); // 10

const element2 = returnNextElement.next(2); // 7
```

returnNextElement is a special object (a generator object) that when its next method is run starts (or continues) running createFlow until it hits yield and returns out the value being ‘yielded’

We pause the execution of the function when we yield from it and when we pass a value in the upcoming next() call it gets returned at the yielded place.

Applications:

- Redux Saga - State management package used in many popular frameworks like React, Angular is based on generators.
- Async programming - Generators make async programming code more readable.
