LESSON 10
CLASSES

Evolution through Layers of Abstraction

Wholeness: JavaScript classes are a new syntax that simplifies and enforces proper use of function constructors and prototypal inheritance for creating objects and using good object-oriented design principles such as encapsulation. Science of Consciousness: Classes are an evolution of JavaScript as a language to support the requirement for good object-oriented design principles needed to develop large applications found in modern web applications. The nature of life is to grow and evolve through layers of abstraction.

Main Points

1. Class syntax
2. Class Inheritance
3. Encapsulation

Main Point Preview: Class syntax

JavaScript classes are mainly a helpful syntax over JavaScript constructor functions. Science of Consciousness: JavaScript classes are an instance of abstracting away implementation details and issues of constructor functions with this improved syntax. Life is found in layers and more abstract layers of consciousness are simpler and more effective.

Classes

- often need to create many objects of the same kind
  - users, or goods or whatever.
  - object literal, constructor function, Object.create
- in modern JavaScript,
  - more advanced “class” construct,
  - introduces new features useful for object-oriented programming
- look like classes from Java and C#, but are different
  - Based on prototype inheritance and function constructors

Class syntax

```
class MyClass {
  // class methods
  constructor() { ... }
  method1() { ... }
  method2() { ... }
  method3() { ... }  ...
}  //no comma between methods (not an object literal)
```

Then use new MyClass() to create a new object with all the listed methods.

The constructor() method is called automatically by new, so we can initialize the object there.

Class syntax

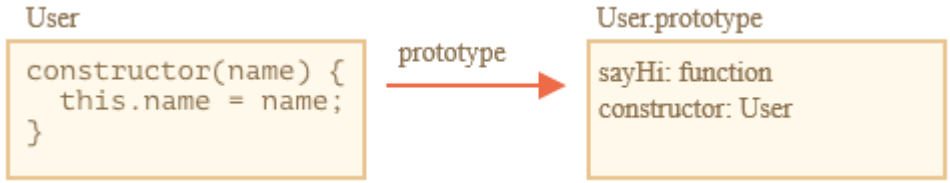
```
class User {
  constructor(name) {
    this.name = name;
  }
  sayHi() {
    alert(this.name);
  }
}

// Usage:
let user = new User("John");
user.sayHi();
```

- When new User("John") is called:
  - A new object is created.
  - The constructor runs with the given argument and assigns it to this.name
  - …Then we can call object methods, such as user.sayHi().

JavaScript classes are (constructor) functions

```
class User {
  constructor(name) { this.name = name; }
  sayHi() { alert(this.name); }
}
// proof: User is a function
alert(typeof User); // function

// Usage:
let user = new User("John");
user.sayHi();
```

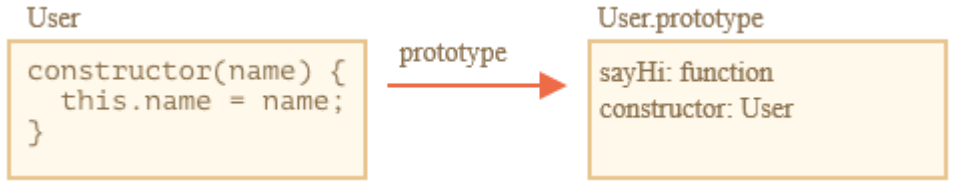
- Creates a *function* named User,
  - result of the class declaration.
  - function code taken from the constructor method
    - assumed empty if we don't write such method).
  - Stores class methods, such as sayHi, in User.prototype.



- Afterwards, for new User objects,
  - call a method, it's taken from the prototype
  - object has access to class methods.

Could write using just constructor function

```
function User(name) {
  this.name = name;
}
// any function prototype has constructor property by default,
// so we don't need to create it

// Add the method to prototype
User.prototype.sayHi = function() {
  alert(this.name);
};

// Usage
let user = new User("John");
user.sayHi();
```



Differences and advantages of ES6 class syntax

- function created by class is labelled by a special internal property [[FunctionKind]]:"classConstructor".
- Error message if forget to use ‘new’

```
class User {
  constructor() {}
}
alert(typeof User); // function
User(); // Error: Class constructor User cannot be invoked without 'new’
```

- string representation of a class constructor in most JavaScript engines starts with the “class…”

```
class User {
  constructor() {}
}
alert(User); // class User { ... }
```

- Class methods are non-enumerable. sets enumerable flag to false for all methods in "prototype".
  - good, because if we for..in over object, usually don’t want class methods
- Classes always use strict
- …

Getters and setters

- Traditional class-oriented languages typically have methods called getters and setters for accessing state information in properties
  - Often contain small amounts of code for constraint checking

```
class User {
  constructor(name) {
  this.name = name; // calls the setter
  }
  get name() {
    return this._name;  //property must match the name used in the setter
  }
  set name(value) {
    if (value.length < 4) {
      alert("Name is too short.");
      return;
    }
    this._name = value;  //must set a property name different from the setter name
  }
}
let user = new User("John");
alert(user.name); // calling the getter
user.name = “Fred”;  calling the setter
user2 = new User(""); // Name too short.
```

Getters and setters

- Some people object to class get and set versus methods setName() and getName()
  - latter will throw error if misspell
  - former will create a new property with the misspelling

- TypeScript will have accessors that throw an error in such cases
- For now we recommend to use setName and getName or directly access the properties

Class properties versus methods

- Class declaration creates getter, setters, methods in the prototype.
  - They are accessible by all objects created from this class (constructor)
  - Properties are created as properties of the object when the class is created

```
class User {
    constructor(name = "Anonymous") {
        this.name = name;
    }
    sayHi() {
        console.log(`Hello, ${this.name}!`);
    }}

fred = new User();
console.log(fred);
console.log(fred.__proto__);
console.log(User.prototype);
fred.sayHi();
bob = new User("Bob");
console.log(bob);
bob.sayHi();
```

Exercise: Rewrite to class

- You can simplify the constructor parameter to just take a string instead of using the object destructuring syntax
- Also add a line of code that will stop the clock after 10 ticks
- What is the inner function of the constructor function?
- What is the local variable of the constructor function?
- What is the clock “interface” returned by the constructor function?
- What are the closures?
- What are the private variables and functions?
- What are the public methods?
- How does this example illustrate that a JavaScript class is really a function and not an object?

Main Point: Class syntax

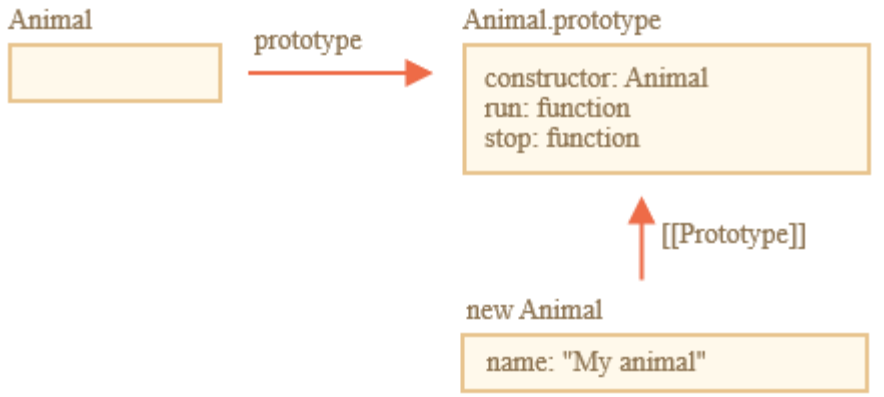
JavaScript classes are mainly a helpful syntax over JavaScript constructor functions. Science of Consciousness: JavaScript classes are an instance of abstracting away implementation details and issues of constructor functions with this improved syntax. Life is found in layers and more abstract layers of consciousness are simpler and more effective.

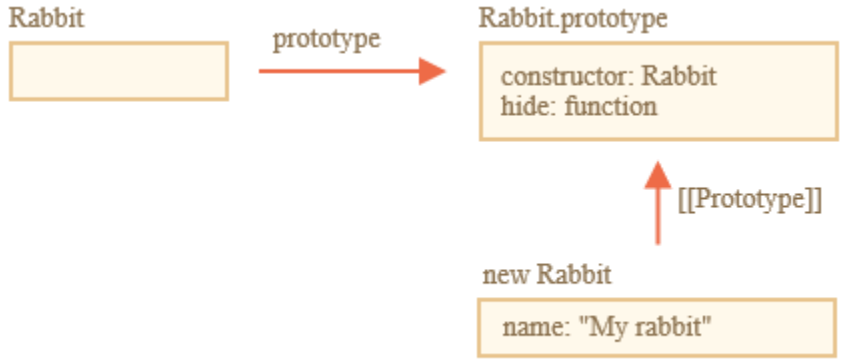
Main Point Preview: Class inheritance

JavaScript classes add keywords extend and super that are abstractions over the details of inheritance with function constructors and prototype chains.

Class inheritance

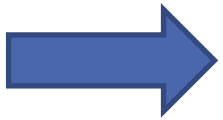
```
class Animal {
  constructor(name) {
    this.speed = 0;
    this.name = name;
  }
  run(speed) {
    this.speed += speed;
    alert(`${this.name} runs with speed ${this.speed}.`);
  }
  stop() {
    this.speed = 0;
    alert(`${this.name} stands still.`);
  }}

class Rabbit {
  constructor(name) {
    this.name = name;
  }
  hide() {
    alert(`${this.name} hides!`);
  }
}
```





Inherit from Animal by specifying "extends" Animal

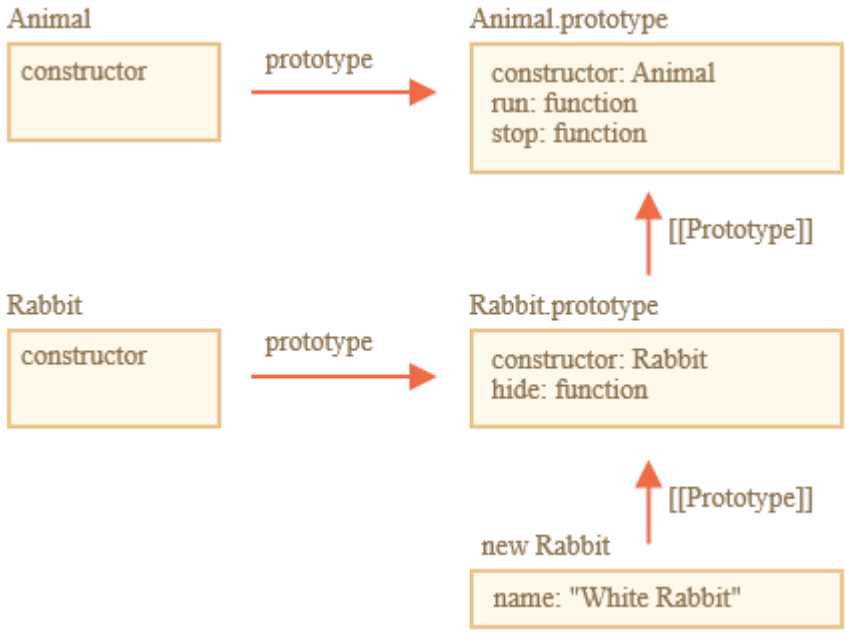
```
class Rabbit {
  constructor(name) {
    this.name = name;
  }
  hide() {    alert(`${this.name} hides!`);  }}
```



```
class Rabbit extends Animal {
  hide() {  alert(`${this.name} hides!`);  }}
```

```
let rabbit = new Rabbit("White Rabbit");
rabbit.run(5); // White Rabbit runs with speed 5.
rabbit.hide(); // White Rabbit hides!
```

- Rabbit code shorter
  - inherits run and stop and constructor
- adds [[Prototype]] reference from Rabbit.prototype to Animal.prototype:
  - if method not found in Rabbit.prototype
    - get from Animal.prototype



Overriding a method

- specify our own stop in Rabbit, it will be used instead

- often don’t want to totally replace a parent method, but build on it
  - do something in our method,
  - call the parent method before/after it or in the process.

- Classes provide "super" keyword for that.
  - super.method(...) to call a parent method.
  - super(...) to call a parent constructor (inside our constructor only)

Overriding a method with super

- Rabbit has the stop method that calls the parent super.stop() in the process.

```
class Animal {
  constructor(name) {
    this.speed = 0;
    this.name = name;
  }
  run(speed) {
    this.speed += speed;
    alert(`${this.name} runs with speed ${this.speed}.`);
  }
  stop() {
    this.speed = 0;
    alert(`${this.name} stands still.`);
  }}

class Rabbit extends Animal {
  hide() {
    alert(`${this.name} hides!`);
  }
  stop() {
    super.stop(); // call parent stop
    this.hide(); // and then hide
  }}
```

Overriding constructor with super

- Till now, Rabbit did not have its own constructor.
- if a class extends another class and has no constructor, then an“empty” constructor is generated

```
class Rabbit extends Animal {
  // generated for extending classes without own constructors
  constructor(...args) {
    super(...args);  }}
```

- add a custom constructor to Rabbit. It will specify the earLength in addition to name
  - needs to call super() before using this
  - When a normal constructor runs, it creates an empty object and assigns it to this.
  - when a derived constructor runs it expects parent constructor to do this job.

```
class Rabbit extends Animal {
  constructor(name, earLength) {
    super(name);
    this.earLength = earLength;
  }
```

Exercises

- Error creating an instance
- Extended clock

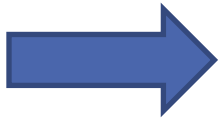
Static properties and methods

- can assign a method to the class function itself, not to its "prototype".
  - Such methods called static

- value of this in User.staticMethod() call is the class constructor User itself
  - “object before dot” rule

- static methods used for functions that belong to the class
  - not to any particular object

```
class User {
  static staticMethod() {
    alert(this === User);
  }
}
User.staticMethod(); // true
```



```
//same as assigning it as a property directly

class User() { }

User.staticMethod = function() {
  alert(this === User);
};
```

Static methods

- Article.compare is a means to compare articles
  - not a method of an article, but rather of the whole class.

```
class Article {
  constructor(title, date) {
    this.title = title;
    this.date = date;
  }
  static compare(articleA, articleB) {
    return articleA.date - articleB.date;
  }
}
// usage
let articles = [
  new Article("HTML", new Date(2019, 1, 1)),
  new Article("CSS", new Date(2019, 0, 1)),
  new Article("JavaScript", new Date(2019, 11, 1))
];
articles.sort(Article.compare);
alert( articles[0].title ); // CSS
```

Static properties

- Static properties are also possible,
  - look like regular class properties, but prepended by static:

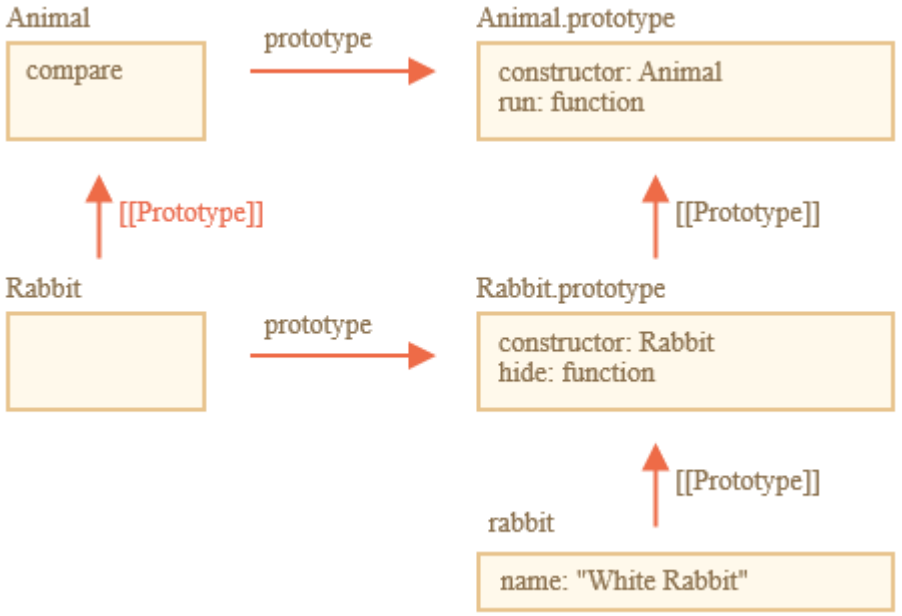
```
class Article {
  static publisher = "Ilya Kantor";
}

alert( Article.publisher ); // Ilya Kantor
```

That is the same as a direct assignment to Article:

```
Article.publisher = "Ilya Kantor";
```

Inheritance of static methods

```
class Animal {
  constructor(name, speed) {
    this.speed = speed;
    this.name = name;
  }
  run(speed = 0) {
    this.speed += speed;
    alert(`${this.name} runs with speed ${this.speed}.`);
  }
  static compare(animalA, animalB) {
    return animalA.speed - animalB.speed;
  }
}
// Inherit from Animal
class Rabbit extends Animal {
  hide() {
    alert(`${this.name} hides!`);
  }
}
let rabbits = [
  new Rabbit("White Rabbit", 10),
  new Rabbit("Black Rabbit", 5)
];
rabbits.sort(Rabbit.compare);
rabbits[0].run(); // Black Rabbit runs with speed 5.
```



- Rabbit extends Animal creates two [[Prototype]] references
  - Rabbit constructor prototypally inherits from Animal constructor
  - Rabbit.prototype prototypally inherits from Animal.prototype.

- Static methods belong to class “as a whole”,
  - Not accessible to specific object instances as object methods
  - i.e., Rabbit.compare, not longEar.compare

Main Point: Class inheritance

JavaScript classes add keywords extend and super that are abstractions over the details of inheritance with function constructors and prototype chains.

Main Point Preview: Encapsulation

Encapsulation is a cornerstone principle of good object-oriented design.

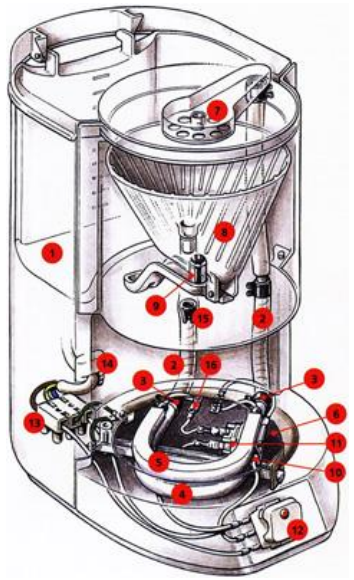
Encapsulation hides internal component implementation details and provides a well delimited external interface for external components. Science of Consciousness: Experience of the simplest state of awareness eliminates stress and allows us to only have thoughts that are relevant to a given situation. This is like providing a well delimited external interface appropriate to the external environment.

Private and protected properties and methods

- One of the most important principles of object-oriented programming
  - delimiting internal interface from the external one.
  - Internal interface – methods and properties, accessible from other methods of the class, but not from the outside.
  - External interface – methods and properties, accessible also from outside the class.

- delimiting of the internal interface from the external one is called encapsulation.

.



Encapsulation benefits

Supports modification and extension

- The situation in programming is more complex than a coffee machine,
  - don’t just buy it once.
  - code constantly undergoes development and improvement.

- If strictly encapsulate internal interface
  - developer of the class can freely change
  - without informing the users.

- If you’re a developer of such a class, it’s great to know that no external code depends on them.
  - private methods can be safely renamed,
  - parameters can be changed, and even removed,

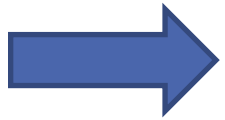
- For users, when a new version comes out, it may be a total overhaul internally,
  - still simple to upgrade if the external interface is the same.

Hides complexity

- People adore using things that are simple
  - Programmers too
  - always good when implementation details are hidden
  - simple, well-documented external interface

Encapsulating a property

```
class CoffeeMachine {
constructor(power) {
   this.waterAmount = 0;
   this.power = power;
    alert( `Created a coffee-machine, power: ${power}` );
  }
}

// create the coffee machine
let coffeeMachine = new CoffeeMachine(100);

// add water
coffeeMachine.waterAmount = 200;
```



```
class CoffeeMachine {

  set waterAmount(value) {
    if (value < 0) throw new Error("Negative water");
    this._waterAmount = value;
  }
  get waterAmount() {
    return this._waterAmount;
  }
  constructor(power) {
    this._waterAmount = 0;
    this._power = power;
  }
}

// create the coffee machine
let coffeeMachine = new CoffeeMachine(100);
// add water
coffeeMachine.waterAmount = -10; // Error: Negative
water
```

Read-only “power”

- For power property, make it read-only
  - for a coffee machine: power never changes
  - make getter, but not setter

```
class CoffeeMachine {
  // ...
  constructor(power) {
    this._power = power;
  }
  get power() {
    return this._power;
  }
}

// create the coffee machine
let coffeeMachine = new CoffeeMachine(100);

alert(`Power is: ${coffeeMachine.power}W`); // Power is: 100W

coffeeMachine.power = 25; // Error (no setter)
```

get.../set... functions preferred to JavaScript getter/setter

```
class CoffeeMachine {
  constructor(){   this._waterAmount = 0; }

  setWaterAmount(value) {
    if (value < 0) throw new Error("Negative water");
    this._waterAmount = value;
  }
  getWaterAmount() {
    return this._waterAmount;
  }
}
new CoffeeMachine().setWaterAmount(100);
```

- longer, but functions are more flexible.
  - can accept multiple arguments
- Get error message if misspell the function name
  - With get and set properties no error if misspell
  - Get a new property with the misspelled name!!
  - Bug waiting to happen!!

Module pattern for encapsulation with classes

- Use the pattern from the CoffeeMachine class with setWaterAmount …
- if use class syntax then do not get to have local variables and closures.
- Closure solution (condensed) from exam2
- Constructor function solution using module pattern
- solution using class keyword and _ convention to indicate private member

Private #properties (coming)

- There’s a finished JavaScript proposal, almost in the standard, that provides language-level support for private properties and methods.
- Privates should start with #. They are only accessible from inside the class.
- Unlike protected ones, private fields are enforced by the language itself.

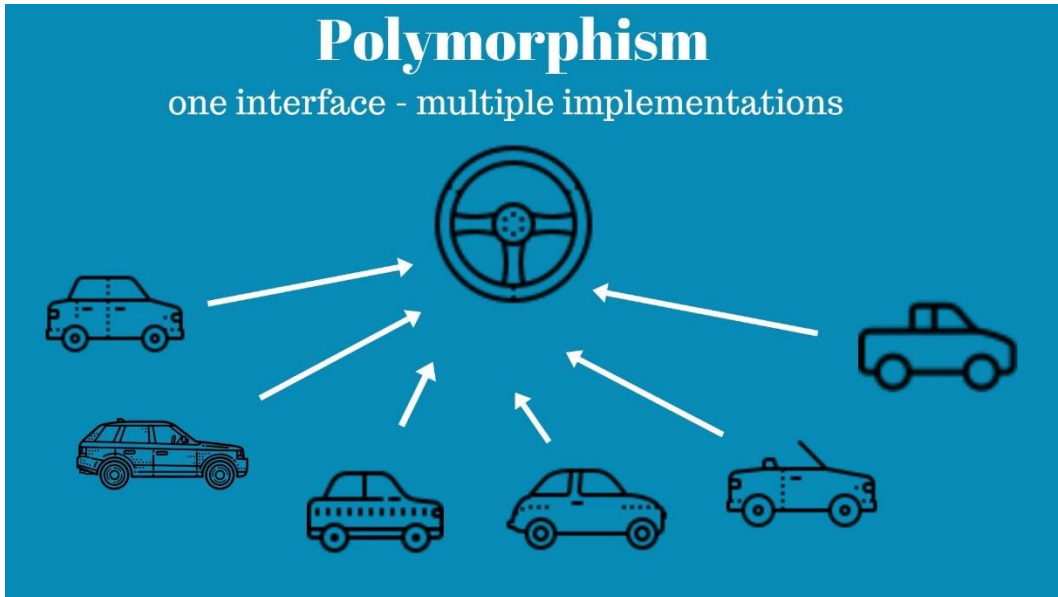
Polymorphism

- ability to substitute classes that have common functionality in sense of methods and data.
  - ability of multiple object types to implement the same functionality
  - can work in a different way but support a common interface

- E.g. function that expects a super class instance as an argument can work correctly with subclass instance as well

- real-life example of polymorphism
  - If you have learned how to drive one car,
  - able to drive any other car;
  - doesn't depend on the make of car it's configuration or inner implementation.
  - has the same driver interface.

- Polymorphism and encapsulation
  - "program to interface, not to implementation"



Polymorphism in JavaScript

- standard prototypes define their own version of toString so they can create a string that contains more useful information than "[object Object]".

```
Rabbit.prototype.toString = function() {
  return `a ${this.type} rabbit`;
};
```

- simple instance of a powerful idea.
  - When code works with objects that have a certain interface
  - any object that supports this interface can be plugged into the code, and it will just work.

- technique is called polymorphism.
  - Polymorphic code can work with values of different shapes,
  - as long as they support the interface it expects

- for/of loop can loop over several kinds of data structures.
  - another case of polymorphism
  - such loops expect the data structure to expose a specific interface, which arrays and strings do
  - can also add this interface(enumerable) to your own objects

- function passing and this assuming different objects, is that polymorphism?
  - e.g., this.click() or this.foo() might be different operation depending on the object that becomes this ...

Main Point: Encapsulation

Encapsulation is a cornerstone principle of good object-oriented design.

Encapsulation hides internal component implementation details and provides a well delimited external interface for external components. Science of Consciousness: Experience of the simplest state of awareness eliminates stress and allows us to only have thoughts that are relevant to a given situation. This is like providing a well delimited external interface appropriate to the external environment.

CONNECTING THE PARTS OF KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE

Evolution through Layers of Abstraction

1. JavaScript classes are a helpful syntax that abstracts out details of function constructors and prototypal inheritance for creating objects.

2. The extends and super keywords cause objects and properties to be set in the function prototype and [[Prototype]] properties.

---

3. **Transcendental consciousness**. Is the simplest state of awareness. It abstracts away everything and is also the basis of everything.

4. **Impulses within the transcendental field:** Impulses at this level are the finest layer of existence and represent the first abstraction of knower, known, and process of knowing when consciousness is aware of itself.

5. **Wholeness moving within itself:** In unity consciousness one appreciates all layers of existence as expressions and abstractions over pure consciousness, the unified field of existence.