Eloquent JavaScript (3rd Edition)

Sources

- [Textbook] Eloquent JavaScript (3rd Edition) (Link)

Chapter 1 – Fundamentals

- The console.log operation is typically used to write out a result to the browser console
    - Note that for Google Chrome, 'Control + Shift + I' opens the Developer Tools dock, which has a console window included
- JavaScript was first introduced in 1995, and the original development team thought it would be a good idea to have a name similar to Java due to the popularity of that programming language
- The language contains several built-in value types which are created by invoking its name:
    - The number value type is 64 bits, which means that there will very rarely be issues with overflow
    - The string value type can be contained within single quotes, double quotes or backticks
        - Note that whenever a backslash (\) is found in quoted text, it indicates that the character after it has a special meaning, and is called escaping the character
        - Strings can be concatenated by using the '+' operator between two strings
        - Backtick-quoted strings, usually called template literals, can embed other values inside of ${} which can include calculations
    - Unary operators consist of operators such as typeof, which returns the type of a value
        - Note that unary operators will operate on only one value, while binary operators operate on two values
    - The Boolean type has only two values, true and false, which are written in those words
        - The && (and), || (or), and ! (not) operators can all be applied to Boolean values
        - Note that true && false result in false, while true || false result in true
        - The logical operators && and || convert the value on their left side to Boolean type in order to decide what to do, but depending on the operator and the result of that conversion, they will return either the original left-hand value or the right-hand value

```
console.log(null || "user")
// → user
console.log("Agnes" || "user")
// → Agnes
```

        - The ternary operator is a conditional operator, where the value on the left of the question mark determines whether the middle value will be the result (in the case of true), or the value on the right (in the case of false)

```
console.log(true ? 1 : 2);
// → 1
console.log(false ? 1 : 2);
// → 2
```

    - Two empty values are null and undefined, which carry no information but are produced when a meaningful value cannot be produced
- When an operator is applied to the "wrong" type of value, JavaScript will quietly convert that value to the type it needs using type coercion, even if the set of rules used to convert aren't what the developer wants or expects

```
console.log(8 * null)
// → 0
console.log("5" - 1)
// → 4
console.log("5" + 1)
// → 51
```

    - When you do not want any type conversions to happen, the === and !== operators can be used, as they test whether two values are precisely equal to each other or not

- It is recommended to use the three-character comparison operators defensively to prevent unexpected type conversions from causing issues in the program

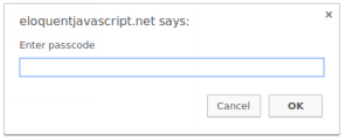
Chapter 2 – Program Structure

- A fragment of code that produces a value is called an expression, while a statement performs an action with the program being essentially a sequence of statements, each ending with a semicolon
  - Note that JavaScript allows you to omit the semicolon at the end of a statement in some cases, but it is best practice to always include the semicolon in all statement-ending situations
- A binding, or variable, stores the internal state of the program using the let, var, or const keywords

```
let one = 1, two = 2;
console.log(one + two);
// → 3
```

  - Note that var is the way that bindings were declared in pre-2015 JavaScript, but should rarely be used as it has some confusing properties (to be discussed in Chapter 3)
  - When creating a binding name, keep in mind that the name must not start with a digit
  - The collection of bindings and their values that exist at a given time is called the environment
- A function is a piece of program wrapped in a value, and a lot of the values provided in the default environment have the type function
  - Such values can be applied in order to run the wrapped program, and executing a function is called invoking, calling, or applying it
  - Values given to functions inside of the parentheses are called arguments, and different functions may require different numbers or types of values
  - Showing a dialog box or writing text to a screen is called a side effect of a function, whereas if a function returns a value, it does not need to have a side effect to be useful

```
prompt("Enter passcode");
```



- Most JavaScript systems (including all modern web browsers and Node.js) provide a console.log function that writes out its arguments to some text output device
  - Browsers typically open this part of the browser interface when you press F12
- Conditional execution is created with the if keyword in JavaScript, where the code afterward can be wrapped in braces (called a block)
  - The else keyword, together with if, can be used together to create two separate, alternative execution paths

```
if (num < 10) {
  console.log("Small");
} else if (num < 100) {
  console.log("Medium");
} else {
  console.log("Large");
}
```

- Looped statements, such as while and do loops, allow us to go back to some point in the program where we were before and repeat it with our current program state
  - Note that a do loop differs from a while loop only in that it always executes its body at least once, and it starts testing whether it should stop only after that first execution
- A for loop is a more compact and comprehensible form of the typical while loop pattern, and has one part of the statement that initializes the loop, one part that checks whether the loop must continue, and a final part that updates the state of the loop after every iteration (although this is not required)

```
for (let current = 20; ; current = current + 1) {
  if (current % 7 == 0) {
    console.log(current);
    break;
  }
}
// → 21
```

  - Note that a special statement called break has the effect of immediately jumping out of the enclosing loop, while continue jumps the control out of the body of the loop and continues with the next iteration
  - In order to update a binding to hold a value based on the binding's previous value in a loop, JavaScript provides shortcuts in the form of counter += 1, or simply counter++

- A switch statement can be used in place of an if-else chain, although it is important to keep in mind that execution will continue through the case labels until a break statement is hit
- Binding naming convention typically follows that every word of the binding name be capitalized, apart from the first word in the name, which remains lowercase throughout
- To allow for related or additional information to be conveyed, comments are provided by the language in the form of // for single-line comments, and /* matched further along with */ for multi-line comments

Chapter 3 – Functions

- As stated in Chapter 2, functions allow for wrapping a piece of program in a value
    - This permits us to structure larger programs, reduce repetition, to associate names with subprograms, and to isolate these subprograms from one another
- A function definition is a regular binding where the value of the binding is a function

```
const square = function(x) {
  return x * x;
};

console.log(square(12));
// → 144
```

    - They are created with an expression that starts with the keyword function, and have a set of parameters (in this case, only x) and a body, which contains the statements that are to be executed when the function is called
    - The function body of a function created this way must always be wrapped in braces, even when it consists of only a single statement
    - Parameters to a function behave like regular bindings, but their initial values are given by the caller of the function and not by the code in the function itself
        - A function can have multiple parameters, or no parameters at all
    - A return statement determines the value that the function returns, although functions are not required to return a result
- Each binding has a scope, which is the part of the program in which the binding is visible
    - For bindings defined outside of any function or block, the scope is the whole program, and these are known as global bindings
    - Bindings created for function parameters or declared inside a function can be referenced only in that function, and are known as local bindings
    - Each time the function is called, new instances of these bindings are created, providing some isolation between functions
    - When multiple bindings have the same name, code can only see the innermost one
- **Important Note:** Bindings declared with let and const are local to the block that they are declared in, so if you create one of those inside of a loop, the code before and after the loop cannot "see" it

```
let x = 10;
if (true) {
  let y = 20;
  var z = 30;
  console.log(x + y + z);
  // → 60
}
// y is not visible here
console.log(x + z);
// → 40
```

    - In pre-2015 JavaScript, only functions created new scopes, so old-style bindings, created with the var keyword, are visible throughout the whole function that they appear in – or throughout the global scope, if they are not in a function
- Blocks and functions can be created inside other blocks and function, producing multiple degrees of locality
    - Each local scope can see all local scopes that contain it, and all scopes can see the global scope, with this approach to binding visibility being called lexical scoping
- A slightly shorter way to create a function binding is to use the function keyword at the start of a statement, which is called a function declaration

```
console.log("The future says:", future());

function future() {
  return "You'll never have flying cars";
}
```

    - Function declarations are not part of the regular top-to-bottom flow of control, as they are conceptually moved to the top of their scope and can be used by all the code in that scope
    - Note that a function binding simply acts as a name for a specific piece of the program, and can also do all things that other values can do, such as storing a function value in a new binding, passing it as an argument to a function, and so on
        - This needs to be contrasted to a function value, where the function keyword is used as an expression rather than a statement, which produces a binding that gives a function as its value

```
// Define f to hold a function value
const f = function(a) {
  console.log(a + 2);
};

// Declare g to be a function
function g(a, b) {
  return a * b * 3.5;
}

// A less verbose function value
let h = a => a % 3;
```

- Arrow functions provide a third notation for functions, in that an arrow (=>) is used in place of the function keyword
    - The arrow comes after the list of parameters and is followed by the function's body
    - When there is only one parameter name, the parentheses can be omitted from the parameter list, and if the body is a single expression, the expression will be returned from the function without braces
    - Arrow functions were added in 2015 in order to make it possible to write small function expressions in a less verbose way, but otherwise do not have any deep reason to have been included in the language

```
const power = (base, exponent) => {
  let result = 1;
  for (let count = 0; count < exponent; count++) {
    result *= base;
  }
  return result;
};
```

```
const square1 = (x) => { return x * x; };
const square2 = x => x * x;
```

- A call stack in the computer's memory stores the current context
    - Every time a function is called, the current context is stored on top of this stack, and when a function is returned, it removes the top context from the stack and uses that context to continue execution
- JavaScript is extremely broad-minded about the number of arguments that you pass to a function, allowing for optional arguments to be included
    - If too many arguments are passed, the extra ones are ignored, and if too few are passed, the missing parameters get assigned the value undefined
    - Although this can lead to errors, the upside of the argument versatility is that this behavior can be used to allow a function to be called with different numbers of arguments
    - Default values can be assigned if an = operator, followed by an expression, is included after a parameter

```
function minus(a, b) {
  if (b === undefined) return -a;
  else return a - b;
}

console.log(minus(10));
// → -10
console.log(minus(10, 5));
// → 5
```

- Being able to reference a specific instance of a local binding in an enclosing scope is called a closure, which put differently, means that local bindings are created anew for every call, and different calls cannot trample on one another's local bindings
    - A function that references bindings from local scopes around it is called a closure
    - This behavior not only frees you from having to worry about lifetimes of bindings, but also makes it possible to use function values in creative ways
    - From this YouTube video, "closures are nothing but functions that preserve data"

```
function wrapValue(n) {
  let local = n;
  return () => local;
}

let wrap1 = wrapValue(1);
let wrap2 = wrapValue(2);
console.log(wrap1());
// → 1
console.log(wrap2());
// → 2
```

- With a slight change to the above, we can turn the previous example into a way to create functions that multiply by an arbitrary amount
    - In this case, the explicit local binding from the wrapValue example isn't really needed since a parameter is itself a local binding
    - Thinking about programs like this takes some practice, and a good mental model is to think of function values as containing both the code in their body and the environment in which they are created

```
function multiplier(factor) {
  return number => number * factor;
}

let twice = multiplier(2);
console.log(twice(5));
// → 10
```

    - When called, the function body sees the environment in which it was created, not the environment in which it is called
    - In the example, multiplier is called and creates an environment in which its factor parameter is bound to 2; The function value it returns, which is stored in twice, remembers this environment so that when it is called, it multiplies its argument by 2
- It's ok for a function to call itself through recursion so long as it does not do it enough to overflow the stack

```
function power(base, exponent) {
  if (exponent == 0) {
    return 1;
  } else {
    return base * power(base, exponent - 1);
  }
}

console.log(power(2, 3));
// → 8
```

    - Recursion allows functions to be written in a different style, although the downside is that in typical JavaScript applications, it is about three times slower than the looping version
    - Keep in mind when writing recursive functions that all paths must return a value
    - The dilemma of speed versus elegance is an interesting one, and the programmer must decide on an appropriate balance
    - Worrying about efficiency can be a distraction and can lead to paralysis, so always start by writing something that's correct and easy to understand, improving on it in the rare case that it is too slow
- There are two natural way for functions to be introduced into programs:
    - The first is that you find yourself writing code multiple times, making mistakes more likely and giving more material for people to have to read to understand the program, and hence you take the repeated functionality and place it into an aptly named function
    - The second is that you find the need for functionality that you do not yet have, so you make a name and start writing the body
        - How difficult it is to find a good name for a function is a good indication of how clear a concept is that you're trying to wrap, where a function should really focus on a single generalized concept
- A useful principle is to not add cleverness unless you are sure that you're going to need it
    - It can be tempting to write general "frameworks" for every bit of functionality you come across, but that urge should be resisted as it will waste time on code that will never be used
- Functions can be roughly divided into those that are called for their side effects (ex. printing lines of strings) and those that are called for their return value
    - Functions that create values are easier to combine in new ways than functions that directly perform side effects
    - A pure function is a specific kind of value-producing function that not only has no side effects, but also doesn't rely on side effects from other code – for example, it doesn't read global bindings whose value might change
    - There is no need to feel bad when writing functions that are not pure, or to wage a holy war to purge them from the code, since side effects can often be useful

Chapter 4 – Data Structures: Objects and Arrays

- An array is a data type created specifically for storing sequences of values
    - A pair of square brackets immediately after an expression, with another expression inside of them, will look up the element in the left-hand expression that corresponds to the index given by the expression in the brackets

```
let listOfNumbers = [2, 3, 5, 7, 11];
console.log(listOfNumbers[2]);
// → 5
```

- Expressions such as myString.length and Math.max access a property of some value

    - Almost all JavaScript values have properties, with the exception being null and undefined
- The two main ways to access properties in JavaScript are with a dot and with square brackets
    - Both value.x and value[x] access a property on a value – but not necessarily the same property
    - When using a dot, the word after the dot is the literal name of the property, while when using square brackets, the expression between the brackets is evaluated to get the property name
    - The elements in an array are stored as the array's properties, using numbers as property names
- Properties that contain functions are generally called methods of the value that they belong to, as in "toUpperCase is a method of a string"
    - A stack, as shown in the example, is a data structure that allows you to push values into it and pop them out again in the opposite order so that the thing that was added last is removed first

```
let sequence = [1, 2, 3];
sequence.push(4);
sequence.push(5);
console.log(sequence);
// → [1, 2, 3, 4, 5]
console.log(sequence.pop());
// → 5
console.log(sequence);
// → [1, 2, 3, 4]
```

- Objects allow us to group values – including other objects – into a single value
    - Values of the type object are arbitrary collections of properties
    - It is possible to assign a value to a property expression with the = operator, which will replace the property's value if it already existed or create a new property on the object if it didn't
    - Arrays are just a kind of object specialized for storing sequences of things

```
let day1 = {
  squirrel: false,
  events: ["work", "touched tree", "pizza", "running"]
};
console.log(day1.squirrel);
// → false
console.log(day1.wolf);
// → undefined
day1.wolf = false;
console.log(day1.wolf);
// → false
```

- The built-in value types, such as numbers, strings, and Booleans, are all immutable, meaning that it is impossible to change values of those types
    - You can combine them and derive new values from them, but when you take a specific string value, that value will always remain the same
    - Bindings can be constant or changeable (by using the let binding to change the value that the binding points at), but this is separate from the way that their values behave
- Objects, however, are mutable, which means that their properties can be changed
    - In the example to the right, object1 and object2 are said to share the same identity
    - Keep in mind that while primitives, such as numbers, Boolean values, and strings are copied when reassigned from one binding to another, objects and arrays are not copied, and any bindings will reference the same location in memory
    - Spreading the properties of the object into another object will perform a deep copy of the object's properties

```
let object1 = {value: 10};
let object2 = object1;
let object3 = {value: 10};
console.log(object1 == object2);
// → true
console.log(object1 == object3);
// → false

object1.value = 15;
console.log(object2.value);
// → 15
console.log(object3.value);
// → 10
```

```
const person = {
  name: 'Max'
};

const secondPerson = {
  ...person
};

person.name = 'Manu';

console.log(secondPerson);
```

- Rather than declaring properties like 'events: events', a property name can be given which allows for a shorthand version of the same thing – if a property name in brace notation isn't followed by a value, its value is taken from the binding with the same name
- A classic loop in JavaScript goes over arrays one element at a time by using a counter over the length of the array
    - **Important Note:** A simpler way to write such loops in modern JavaScript is to use a for loop with the of keyword, which will loop over the elements of the value given after that keyword

```
let journal = [];
function addEntry(events, squirrel) {
  journal.push({events, squirrel});
}
addEntry(["work", "touched tree", "pizza", "running",
          "television"], false);
addEntry(["work", "ice cream", "cauliflower", "lasagna",
          "touched tree", "brushed teeth"], false);
addEntry(["weekend", "cycling", "break", "peanuts",
          "beer"], true);
```

```
for (let entry of JOURNAL) {
  console.log(`${entry.events.length} events.`);
}
```

- Besides push and pop, which add and remove elements at the end of an array, unshift and shift can be used to add and remove elements at the start of an array
    - The indexOf method can be used to search through the array for a specific value, while the concat method can be used to glue arrays together to create a new array
- Some common methods when working with strings include indexOf, slice, trim (removes whitespace), zeroPad (pads zeros to start of string), split, join, and repeat
- It can be useful for a function to accept any number of arguments (for example, Math.max computes the maximum of all arguments given)
    - To write such a function, three dots need to be placed before the function's last parameter, which is an array containing all arguments termed the rest parameter

```
function max(...numbers) {
  let result = -Infinity;
  for (let number of numbers) {
    if (number > result) result = number;
  }
  return result;
}
console.log(max(4, 1, 9, -2));
// → 9
```

- The Math object is used as a container to group a bunch of functionalities, providing a namespace so that all these functions and values do not have to be global bindings
- Note that JavaScript will warn you if you are defining a binding with a name already taken, but only for bindings declared with let or const (but not var or function)
- Destructuring an array in JavaScript allows bindings for the elements of the array, rather than having a binding pointing at the entire array
    - A similar trick, called Object Destructuring, also be applied on objects

```
var foo = ['one', 'two', 'three'];

var [one, two, three] = foo;
console.log(one); // "one"
console.log(two); // "two"
console.log(three); // "three"
```

- Keep in mind that properties only grasp their values, rather than contain them, and objects/arrays are stored in the computer's memory as sequences of bits holding the addresses – the place in memory – of their contents
- When sending data from one computer to another over a network, we can serialize the data by converting it into a flat description
    - JSON (pronounced "Jason"), which stands for JavaScript Object Notation, is a popular serialization format that is widely used for data storage and communication across the internet

```
{
  "squirrel": false,
  "events": ["work", "touched tree", "pizza", "running"]
}
```

    - JSON looks similar to JavaScript's way of writing arrays and objects, with a few restrictions, such as surrounding property names by double quotes and allowing only simple expressions

Chapter 5 – Higher-Order Functions

- "There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies." (C.A.R. Hoare, 1980 ACM Turing Award Lecture)
- A large program is a costly program, and not just because of the time that it takes to build
    - Size almost always involves complexity, complexity confuses programmers, and confused programmers in turn introduce mistakes (bugs) into the program
    - A large program then provides a lot of space for these bugs to hide, making them difficult to find
    - Although in the example to the right, the second program may be bigger due to the size of the definitions of sum and range, it is more likely to be correct because

The first is self-contained and six lines long.

```
let total = 0, count = 1;
while (count <= 10) {
  total += count;
  count += 1;
}
console.log(total);
```

The second relies on two external functions and is one line long.

```
console.log(sum(range(1, 10)));
```

the solution is expressed in a vocabulary that corresponds to the problem being solved (it's about ranges and sums rather than loops and counters)

- In the context of programming, these kinds of vocabularies are usually called abstractions
  - Abstractions hide details and give us the ability to talk about problems at a higher (or more abstract) level
  - It is a useful skill, in programming, to notice when you are working at too low a level of abstraction and getting caught up in lower-level implementations when higher-level implementations exist
- Functions that operate on other functions, either by taking them as arguments or by returning them, are called higher-order functions
  - Since we have already seen that functions are regular values, there is nothing particularly remarkable about the fact that such functions exist
  - Higher-order functions allow us to abstract over actions, not just values
  - In the example above, a higher-order function is built that changes other functions

```
function noisy(f) {
  return (...args) => {
    console.log("calling with", args);
    let result = f(...args);
    console.log("called with", args, ", returned", result);
    return result;
  };
}
noisy(Math.min)(3, 2, 1);
// → calling with [3, 2, 1]
// → called with [3, 2, 1] , returned 1
```

- One area where higher-order functions shine is data processing
  - The forEach, filter, map, reduce, some, and findIndex methods are all higher-order standard array methods that can be used to iterate through or extract information from an array
  - The filter method uses a test argument to build up a new array with only the elements that pass the test, while the map method transforms an array by applying a function to all its elements and building a new array from the returned values
  - The higher order reduce method is a pattern that allows a single value to be computed from an array by repeatedly combining a single element from the array with the current value
- Higher-order functions are most useful when composing operations, meaning that operations are chained together for greater effect
  - This composition typically results in readable and compact code, which can be essentially viewed as a pipeline of code
  - Be careful when abstracting large datasets, though, as a less abstract style will likely result in fewer computations and additional processing speed

```
function reduce(array, combine, start) {
  let current = start;
  for (let element of array) {
    current = combine(current, element);
  }
  return current;
}

console.log(reduce([1, 2, 3, 4], (a, b) => a + b, 0));
// → 10
```

```
function average(array) {
  return array.reduce((a, b) => a + b) / array.length;
}

console.log(Math.round(average(
  SCRIPTS.filter(s => s.living).map(s => s.year))));
// → 1188
console.log(Math.round(average(
  SCRIPTS.filter(s => !s.living).map(s => s.year))));
// → 188
```

- Being able to pass function values to other functions is a deeply useful aspect of JavaScript
  - It allows us to write functions that model computations with "gaps" in them
  - The code that calls these functions can fill in the gaps by providing function values

Chapter 6 – The Secret Life of Objects

- Object-oriented programming is a set of techniques that use objects (and related concepts) as the central principle of program organization
  - The core idea in object-oriented programming is to divide programs into smaller pieces and make each piece responsible for managing its own state

    - Because of this, knowledge about the way a piece of the program works can be kept local to that piece, and someone working on the rest of the program does not have to remember or even be aware of that knowledge
- Different pieces of such a program interact with each other through interfaces, which are limited sets of functions or bindings that provide useful functionality at a more abstract level, hiding their precise implementation
    - Such program pieces are modeled using objects, and their interface consists of a specific set of methods and public properties (private properties are used to protect data that outside code should not access)
    - Note that in JavaScript, the use of the private keyword is not currently available, so programmers are placing an underscore (_) character at the start of the property name to indicate that it is a private variable
- Separating interface from implementation is a great idea, and is generally referred to as encapsulation
- Methods are nothing more than properties that hold function values
    - When a function is called as a method, the method usually needs to do something with the object that it was called on, and the binding called this in its body automatically points at the calling object, exposing other properties to the method
    - Note that arrow functions do not bind their own this, but can see the this binding of the scope around them

```
let rabbit = {};
rabbit.speak = function(line) {
  console.log(`The rabbit says '${line}'`);
};

rabbit.speak("I'm alive.");
// → The rabbit says 'I'm alive.'
```

```
function normalize() {
  console.log(this.coords.map(n => n / this.length));
}
normalize.call({coords: [0, 2, 3], length: 5});
// → [0, 0.4, 0.6]
```

- In addition to their set of properties, most objects also have a prototype, which is another object that is used as a fallback source of properties
    - When an object gets a request for a property that it does not have, its prototype will be searched for the property, then the prototype's prototype, and so on
    - The prototype behind an empty object is the entity behind almost all objects, Object.prototype

```
let empty = {};
console.log(empty.toString);
// → function toString()…{}
console.log(empty.toString());
// → [object Object]
```

- A shorthand way of creating a method inside of an object is shown to the right
- The object-oriented concept of a class defines the shape of a type of object – what methods and properties it has – such an object being called an instance of a class

```
let protoRabbit = {
  speak(line) {
    console.log(`The ${this.type} rabbit says '${line}'`);
  }
};
```

    - Prototypes are useful for defining properties for which all instances of a class share the same value, such as methods
    - Properties that differ per instance, such as our rabbits' type property, need to be stored directly in the objects themselves

```
function Rabbit(type) {
  this.type = type;
}
Rabbit.prototype.speak = function(line) {
  console.log(`The ${this.type} rabbit says '${line}'`);
};

let weirdRabbit = new Rabbit("weird");
```

    - By convention, the names of constructors are capitalized so that they can easily be distinguished from other functions
- JavaScript classes are constructor functions with a prototype property, and until 2015, that is how you had to write them
    - As of 2015, the class keyword starts a class declaration, which allows us to define a constructor and a set of methods all in a single place
    - Any number of methods may be written inside the declaration's braces, although the method named

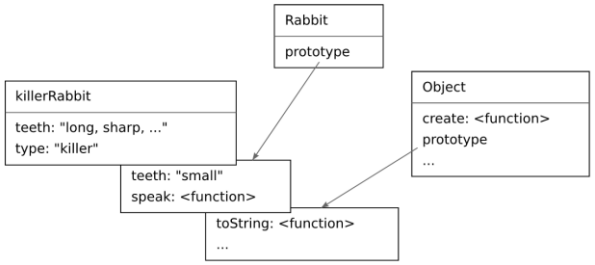
```
class Rabbit {
  constructor(type) {
    this.type = type;
  }
  speak(line) {
    console.log(`The ${this.type} rabbit says '${line}'`);
  }
}

let killerRabbit = new Rabbit("killer");
let blackRabbit = new Rabbit("black");
```

constructor is treated differently in that it provides the actual constructor function, which will be bound to the name Rabbit (in this example)
    - Note that the other methods are packaged into that constructor's prototype, and that this method of class declaration is equivalent the previous method detailed above, but just looks nicer
    - Like a function, a class can be used both in statements and in expressions, and you can omit the class name in a class function

```
let object = new class { getWord() { return "hello"; } };
console.log(object.getWord());
// → hello
```

- Class declarations currently allow only methods – properties that hold functions – to be added to the prototype
    - This can be somewhat inconvenient when you want to save a non-function value in there, but the next version of the language will probably improve on this
    - For now, you can create such properties by directly manipulating the prototype after you've defined the class

```
Rabbit.prototype.teeth = "small";
console.log(killerRabbit.teeth);
// → small
killerRabbit.teeth = "long, sharp, and bloody";
console.log(killerRabbit.teeth);
// → long, sharp, and bloody
console.log(blackRabbit.teeth);
// → small
console.log(Rabbit.prototype.teeth);
// → small
```



- When you add a property to an object, whether it is present in the prototype or not, the property is added to the object itself
    - If there was already a property with the same name in the prototype, this property will no longer affect the object, as it is now hidden behind the object's own property
    - Overriding can be used to express exceptional properties in instances of a more generic class of objects, while letting the nonexceptional objects take a standard value from their prototype.
- JavaScript allows mapping, which is like a dictionary in Python
    - A map (noun) is a data structure that associates values (the keys) with other values
    - The methods set, get, and has are part of the interface of the Map object

```
let ages = new Map();
ages.set("Boris", 39);
ages.set("Liang", 22);
ages.set("Júlia", 62);

console.log(`Júlia is ${ages.get("Júlia")}`);
// → Júlia is 62
console.log("Is Jack's age known?", ages.has("Jack"));
// → Is Jack's age known? false
console.log(ages.has("toString"));
// → false
```

- Polymorphism is when a piece of code is written to work with objects that have a certain interface
    - Any kind of object that happens to support this interface can be plugged into the code, and it will just work
- Properties that are accessed directly may hide a method call in the form of getters and setters

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

console.log(String(blackRabbit));
// → a black rabbit
```

    - Whenever someone reads from the object's property, the associated method is called
    - In the example to the right, the Temperature class allows you to read and write the temperature in either degrees Celsius or degrees Fahrenheit, but internally only stores Celsius and automatically converts to and from Celsius in the Fahrenheit getter and setter

```
class Temperature {
  constructor(celsius) {
    this.celsius = celsius;
  }
  get fahrenheit() {
    return this.celsius * 1.8 + 32;
  }
  set fahrenheit(value) {
    this.celsius = (value - 32) / 1.8;
  }

  static fromFahrenheit(value) {
    return new Temperature((value - 32) / 1.8);
  }
}

let temp = new Temperature(22);
console.log(temp.fahrenheit);
// → 71.6
temp.fahrenheit = 86;
console.log(temp.celsius);
// → 30
```

- Inside a class declaration, methods that have static written before their name are stored on the constructor
- JavaScript's prototype system makes it possible to create a new class, much like the old class, but with new definitions for some of its properties
    - For example, the new class prototype could derive from the old class prototype, but add a new definition for the set method
    - This is called inheritance, as the new class inherits properties and behaviour from the old class
    - The use of the word extends indicates that this class (subclass) shouldn't be directly based on the default Object prototype, but on some other class (superclass)
    - Inside class methods, super provides a way to call methods as they were defined in the superclass

```
class Human {
    species = 'human';
}

class Person extends Human {
    name = 'Max';
    printMyName = () => {
        console.log(this.name);
    }
}

const person = new Person();
person.printMyName();
console.log(person.species); // prints 'human'
```

- Inheritance allows us to build slightly different data types from existing data types with relatively little work, and is a fundamental part of object-oriented tradition, alongside encapsulation and polymorphism
    - While the latter two are now generally regarded as wonderful ideas, inheritance is more controversial
    - Whereas encapsulation and polymorphism can be used to separate pieces of code from each other, reducing the coupling of the overall program, inheritance fundamentally ties classes together, creating more coupling
    - Inheritance can be a useful tool, and the author uses it now and then in his own programs, but it shouldn't be the first tool that is reached for, and generally shouldn't actively be sought out to construct class hierarchies (family trees of classes)
- The instanceof method is useful in determining whether an object was derived from a specific class