JavaScript Functions are First-Class Citizens

Storing functions in variables

Functions are first-class citizens in JavaScript. In other words, you can `treat functions like values of other types`.

The following defines the `add()` function and assigns the function name to the variable `sum`:

```
function add(a, b) {
    return a + b;
}

let sum = add; // now, sum is the add function
```

We can have two ways to execute the same function. For example, we can call it normally as follows:

```
let result = add(10, 20);
```

Alternatively, we can all the `add()` function via the `sum` variable like this:

```
let result = sum(10,20);
```

Passing a function to another function

Because functions are values, you can pass a function as an argument into another function.

The following declares the `average()` function that takes three arguments. The third argument is a function:

```
function average(a, b, fn) {
    return fn(a, b) / 2;
}
```

Now, you can pass the sum function to the `average()` function as follows:

```
let result = average(10, 20, sum);
```

Put it all together:

```
function add(a, b) {
    return a + b;
}

let sum = add;

function average(a, b, fn) {
    return fn(a, b) / 2;
}

let result = average(10, 20, sum);

console.log(result);
```

Output:

```
15
```

Returning functions from functions

Since `functions are values`, you can `return a function from another function`.

The following `compareBy()` function returns a function that compares two objects by a property:

```
function compareBy(propertyName) {
  return function (a, b) {
    let x = a[propertyName],
      y = b[propertyName];

    if (x > y) {
      return 1;
    } else if (x < y) {
      return -1;
    } else {
      return 0;
    }
  };
}
```

Note that `a[propertyName]` returns the value of the propertyName of the a object. It’s equivalent to `a.propertyName`. However, if the propertyName contains a space like `'Discount Price'`, you need to use the `square bracket notation` to access it.

Suppose that you have an array of product objects where each product object has two properties: `name` and `price`.

```
let products = [
    {name: 'iPhone', price: 900},
    {name: 'Samsung Galaxy', price: 850},
    {name: 'Sony Xperia', price: 700}
];
```

You can sort an array by calling the `sort()` method. The `sort()` method accepts a function that compares two elements of the array as an argument.

For example, you can sort the product objects based on the name by passing a function returned from the `compareBy()` function as follows:

```
console.log('Products sorted by name:');
products.sort(compareBy('name'));

console.table(products);
```

```
Products sorted by name:
┌─────────┬──────────────────┬───────┐
│ (index) │       name       │ price │
├─────────┼──────────────────┼───────┤
│    0    │ 'Samsung Galaxy' │  850  │
│    1    │  'Sony Xperia'   │  700  │
│    2    │     'iPhone'     │  900  │
└─────────┴──────────────────┴───────┘
```

Similarly, you can sort the product objects by price:

```
// sort products by prices

console.log('Products sorted by price:');
products.sort(compareBy('price'));
console.table(products);
```

Output:

```
Products sorted by price:
┌─────────┬──────────────────┬───────┐
│ (index) │       name       │ price │
├─────────┼──────────────────┼───────┤
│    0    │  'Sony Xperia'   │  700  │
│    1    │ 'Samsung Galaxy' │  850  │
│    2    │     'iPhone'     │  900  │
└─────────┴──────────────────┴───────┘
```

Put it all together.

```
function compareBy(propertyName) {
  return function (a, b) {
    let x = a[propertyName],
      y = b[propertyName];

    if (x > y) {
      return 1;
    } else if (x < y) {
      return -1;
    } else {
      return 0;
    }
  };
}
let products = [
  { name: 'iPhone', price: 900 },
  { name: 'Samsung Galaxy', price: 850 },
  { name: 'Sony Xperia', price: 700 },
];

// sort products by name
console.log('Products sorted by name:');
products.sort(compareBy('name'));

console.table(products);

// sort products by price
console.log('Products sorted by price:');
products.sort(compareBy('price'));
console.table(products);
```

More JavaScript Functions are First-Class Citizens example

The following example defines two functions that convert a length in centimeters to inches and vice versa:

```
function cmToIn(length) {
    return length / 2.54;
}

function inToCm(length) {
    return length * 2.54;
}
```

The following convert() function has two parameters. The first parameter is a function and the second one is the length that will be converted based on the first argument:

```
function convert(fn, length) {
    return fn(length);
}
```

To convert cm to in, you can call the convert() function and pass the cmToIn function into the convert() function as the first argument:

```
let inches = convert(cmToIn, 10);
console.log(inches);
```

Output:

```
3.937007874015748
```

Similarly, to convert a length from inches to centimeters, you can pass the inToCm function into the convert() function, like this:

```
let cm = convert(inToCm, 10);
console.log(cm);
```

Output:

```
25.4
```

Put it all together.

```
function cmToIn(length) {
  return length / 2.54;
}

function inToCm(length) {
  return length * 2.54;
}

function convert(fn, length) {
  return fn(length);
}

let inches = convert(cmToIn, 10);
console.log(inches);
```

```
let cm = convert(inToCm, 10);
console.log(cm);
```

Output:

```
3.937007874015748
25.4
```

Summary

- Functions are first-class citizens in JavaScript.
- You can pass functions to other functions as arguments, return them from other functions as values, and store them in variables.

JavaScript Recursive Function

Introduction to the JavaScript recursive functions

A recursive function is a function that calls itself until it doesn't. This technique is called recursion.

Suppose that you have a function called `recurse()`. The `recurse()` is a recursive function if it calls itself inside its body, like this:

```
function recurse() {
    // ...
    recurse();
    // ...
}
```

A recursive function always has a condition to stop calling itself. Otherwise, it will call itself indefinitely. So a recursive function typically looks like the following:

```
function recurse() {
    if(condition) {
        // stop calling itself
        //...
    } else {
        recurse();
    }
}
```

Generally, you use recursive functions to break down a big problem into smaller ones. Typically, you will find the recursive functions in data structures like binary trees and graphs and algorithms such as binary search and quicksort.

JavaScript recursive function examples

Let’s take some examples of using recursive functions.

1) A simple JavaScript recursive function example

Suppose that you need to develop a function that counts down from a specified number to 1. For example, to count down from 3 to 1:

```
3
2
1
```

The following shows the `countDown()` function:

```
function countDown(fromNumber) {
    console.log(fromNumber);
}

countDown(3);
```

This `countDown(3)` shows only the number 3.

To count down from the number 3 to 1, you can:

1. show the number 3.
2. and call the `countDown(2)` that shows the number 2.
3. and call the `countDown(1)` that shows the number 1.

The following changes the `countDown()` to a recursive function:

```
function countDown(fromNumber) {
    console.log(fromNumber);
    countDown(fromNumber-1);
}

countDown(3);
```

This `countDown(3)` will run until the call stack size is exceeded, like this:

```
Uncaught RangeError: Maximum call stack size exceeded.
```

... because it doesn’t have the condition to stop calling itself.

The countdown will stop when the next number is zero. Therefore, you add an `if condition` as follows:

```
function countDown(fromNumber) {
    console.log(fromNumber);

    let nextNumber = fromNumber - 1;

    if (nextNumber > 0) {
        countDown(nextNumber);
    }
}
countDown(3);
```

Output:

```
3
2
1
```

The `countDown()` seems to work as expected.

However, as mentioned in the Function type tutorial, the function’s name is a reference to the actual function object.

If the function name is set to null somewhere in the code, the recursive function will stop working.

For example, the following code will result in an error:

```
let newYearCountDown = countDown;
// somewhere in the code
countDown = null;
// the following function call will cause an error
newYearCountDown(10);
```

Error:

```
Uncaught TypeError: countDown is not a function
```

How the script works:

- First, assign the countDown function name to the variable newYearCountDown.
- Second, set the countDown function reference to null.
- Third, call the newYearCountDown function.

The code causes an error because the body of the `countDown()` function references the `countDown` function name, which was set to `null` at the time of calling the function.

To fix it, you can use a named function expression as follows:

```
let countDown = function f(fromNumber) {
    console.log(fromNumber);

    let nextNumber = fromNumber - 1;

    if (nextNumber > 0) {
        f(nextNumber);
    }
}

let newYearCountDown = countDown;
countDown = null;
newYearCountDown(10);
```

2) Calculate the sum of n natural numbers example

Suppose you need to calculate the sum of natural numbers from 1 to n using the recursion technique. To do that, you need to define the `sum()` recursively as follows:

```
sum(n) = n + sum(n-1)
sum(n-1) = n - 1 + sum(n-2)
...
sum(1) = 1
```

The following illustrates the `sum()` recursive function:

```
function sum(n) {
  if (n <= 1) {
    return n;
  }
  return n + sum(n - 1);
}
```

Base Case:

- The function starts with an “if” statement that checks if `n` is less than or equal to 1.
- If n is 1 or less, the function simply returns `n`. This is the `base case`, which serves as the stopping condition for the recursion.

Recursive Case:

- If the `base case` is not met (i.e., `n` is greater than 1), the function enters the block after the if statement.
- The function returns the sum of `n` and the result of calling itself with the argument `(n - 1)`. This is where the recursion happens.

How it Works:

- For example, if you call `sum(3)`, the function first checks if 3 is less than or equal to 1 (base case not met). Since it’s not the base case, it calculates `3 + sum(2)`. Now, it calls itself with the argument 2. In the next recursive call with `sum(2)`, it calculates `2 + sum(1)`. Again, in the next recursive call with `sum(1)`, it reaches the base case and returns 1. Now, the previous calls are resolved: `2 + 1` gives 3, and `3 + 3` gives the final result of 6.

Termination:

- The recursion keeps happening, reducing the problem to smaller sub-problems until it reaches the base case.
- Once the base case is reached, the function starts to unwind, combining the results from each level of recursion until the final result is obtained.

Summary

- A recursive function is a function that calls itself until it doesn’t
- A recursive function always has a condition that stops the function from calling itself.

JavaScript Computed Property

Summary: in this tutorial, you’ll learn about the JavaScript computed properties introduced in ES6.

Introduction to JavaScript Computed Property ES6 allows you to use an expression in brackets `[]`. It’ll then use the result of the expression as the property name of an object. For example:

```
let propName = 'c';

const rank = {
  a: 1,
  b: 2,
  [propName]: 3,
};

console.log(rank.c); // 3
```

In this example, the `[propName]` is a computed property of the `rank` object. The property name is derived from the value of the `propName` variable.

When you access c property of the rank object, JavaScript evaluates propName and returns the property’s value.

Like an object literal, you can use computed properties for `getters and setters` of a `class`. For example:

```
let name = 'fullName';

class Person {
  constructor(firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
  }
  get [name]() {
    return `${this.firstName} ${this.lastName}`;
  }
}

let person = new Person('John', 'Doe');
console.log(person.fullName);
```

Output:

```
John Doe
```

How it works:

The `get[name]` is a computed property name of a getter of the `Person` class. At runtime, when you access the `fullName` property, the person object calls the getter and returns the full name.

Summary

Computed properties allow you to use the values of expressions as property names of an object.

JavaScript Inheritance Using extends & super

Implementing JavaScript inheritance using `extends` and `super` Prior to ES6, implementing a proper inheritance required multiple steps. One of the most commonly used strategies is prototypal inheritance.

The following illustrates how the `Bird` inherits properties from the `Animal` using the prototypal inheritance technique:

```
function Animal(legs) {
    this.legs = legs;
}

Animal.prototype.walk = function() {
    console.log('walking on ' + this.legs + ' legs');
}

function Bird(legs) {
    Animal.call(this, legs);
}
```

```
Bird.prototype = Object.create(Animal.prototype);
Bird.prototype.constructor = Animal;


Bird.prototype.fly = function() {
    console.log('flying');
}

var pigeon = new Bird(2);
pigeon.walk(); // walking on 2 legs
pigeon.fly();  // flying
```

ES6 simplified these steps by using the `extends` and `super` keywords.

The following example defines the `Animal` and `Bird` classes and establishes the inheritance through the `extends` and `super` keywords.

```
class Animal {
    constructor(legs) {
        this.legs = legs;
    }
    walk() {
        console.log('walking on ' + this.legs + ' legs');
    }
}

class Bird extends Animal {
    constructor(legs) {
        super(legs);
    }
    fly() {
        console.log('flying');
    }
}


let bird = new Bird(2);

bird.walk();
bird.fly();
```

How it works.

First, use the `extends` keyword to make the `Bird` class inheriting from the `Animal` class:

```
class Bird extends Animal {
   // ...
}
```

The Animal class is called a base class or parent class while the Bird class is known as a derived class or child class. By doing this, the Bird class inherits all methods and properties of the Animal class.

Second, in the Bird's constructor, call super() to invoke the Animal's constructor with the legs argument.

JavaScript requires the child class to call super() if it has a constructor. As you can see in the Bird class, the super(legs) is equivalent to the following statement in ES5:

```
Animal.call(this, legs);
```

If the Bird class doesn't have a constructor, you don't need to do anything else:

```
class Bird extends Animal {
    fly() {
        console.log('flying');
    }
}
```

It is equivalent to the following class:

```
class Bird extends Animal {
    constructor(...args) {
        super(...args);
    }
    fly() {
        console.log('flying');
    }
}
```

However, the child class has a constructor, it needs to call super(). For example, the following code results in an error:

```
class Bird extends Animal {
    constructor(legs) {
    }
    fly() {
        console.log('flying');
    }
}
```

Error:

```
ReferenceError: Must call super constructor in derived class before accessing
'this' or returning from derived constructor
```

Because the `super()` initializes the `this` object, you need to call the `super()` before accessing the `this` object. Trying to access this before calling `super()` also results in an error.

For example, if you want to initialize the `color` property of the `Bird` class, you can do it as follows:

```
class Bird extends Animal {
    constructor(legs, color) {
        super(legs);
        this.color = color;
    }
    fly() {
        console.log("flying");
    }
    getColor() {
        return this.color;
    }
}

let pegion = new Bird(2, "white");
console.log(pegion.getColor());
```

Shadowing methods

ES6 allows the child class and parent class to have methods with the same name. In this case, when you call the method of an object of the child class, the method in the child class will shadow the method in the parent class.

The following `Dog` class extends the `Animal` class and redefines the `walk()` method:

```
class Dog extends Animal {
    constructor() {
        super(4);
    }
    walk() {
        console.log(`go walking`);
    }
}

let bingo = new Dog();
bingo.walk(); // go walking
```

To call the method of the parent class in the child class, you use `super.method(arguments)` like this:

```
class Dog extends Animal {
    constructor() {
        super(4);
    }
    walk() {
        super.walk();
        console.log(`go walking`);
    }
}

let bingo = new Dog();
bingo.walk();
// walking on 4 legs
// go walking
```

Inheriting static members

Besides the properties and methods, the child class also inherits all static properties and methods of the parent class. For example:

```
class Animal {
    constructor(legs) {
        this.legs = legs;
    }
    walk() {
        console.log('walking on ' + this.legs + ' legs');
    }
    static helloWorld() {
        console.log('Hello World');
    }
}

class Bird extends Animal {
    fly() {
        console.log('flying');
    }
}
```

In this example, the `Animal` class has the `helloWorld()` static method and this method is available as `Bird.helloWorld()` and behaves the same as the Animal.helloWorld() method:

```
Bird.helloWorld(); // Hello World
```

Inheriting from built-in types

JavaScript allows you to extend a built-in type such as Array, String, Map, and Set through inheritance.

The following `Queue` class extends the `Array` reference type. The syntax is much cleaner than the `Queue` implemented using the constructor/prototype pattern.

```
class Queue extends Array {
    enqueue(e) {
        super.push(e);
    }
    dequeue() {
        return super.shift();
    }
    peek() {
        return !this.empty() ? this[0] : undefined;
    }
    empty() {
        return this.length === 0;
    }
}

var customers = new Queue();
customers.enqueue('A');
customers.enqueue('B');
customers.enqueue('C');

while (!customers.empty()) {
    console.log(customers.dequeue());
}
```

Summary

- Use the `extends` keyword to implement the inheritance in ES6. The class to be extended is called a base class or parent class. The class that extends the base class or parent class is called the derived class or child class.
- Call the `super(arguments)` in the child class's constructor to invoke the parent class's constructor.
- Use `super` keyword to call methods of the parent class in the methods of the child class.

Introduction to JavaScript `new.target` Metaproperty

Introduction to JavaScript `new.target`

ES6 provides a metaproperty named `new.target` that allows you to detect whether a function or constructor was called using `the` new operator.

The `new.target` consists of the `new` keyword, a dot, and `target` property. The `new.target` is available in all functions.

However, in arrow functions, the `new.target` is the one that belongs to the surrounding function.

The `new.target` is very useful to inspect at runtime whether a function is being executed as a function or as a constructor. It is also handy to determine a specific derived class that was called by using the `new` operator from within a parent class.

JavaScript `new.target` in functions

Let’s see the following `Person` constructor function:

```
function Person(name) {
    this.name = name;
}
```

You can create a new object from the `Person` function by using the `new` operator as follows:

```
let john = new Person('John');
console.log(john.name); // john
```

Or you can call the `Person` as a function:

```
Person('Lily');
```

Because the `this` is set to the global object i.e., the `window` object when you run JavaScript in the web browser, the `name` property is added to the `window` object as follows:

```
console.log(window.name); //Lily
```

To help you detect whether a function was called using the new operator, you use the `new.target` metaproperty.

In a regular function call, the `new.target` returns `undefined`. If the function was called with the `new` operator, the `new.target` returns a reference to the function.

Suppose you don’t want the `Person` to be called as a function, you can use the `new.target` as follows:

```
function Person(name) {
    if (!`new.target`) {
        throw "must use new operator with Person";
    }
    this.name = name;
}
```

Now, the only way to use `Person` is to instantiate an object from it by using the `new` operator. If you try to invoke it like a regular function, you will encounter an error.

JavaScript `new.target` in constructors In a class constructor, the `new.target` refers to the constructor that was invoked directly by the `new` operator. It is `true` if the constructor is in the parent class and was delegated from the constructor of the child class:

```
class Person {
    constructor(name) {
        this.name = name;
        console.log(`new.target`.name);
    }
}

class Employee extends Person {
    constructor(name, title) {
        super(name);
        this.title = title;
    }
}

let john = new Person('John Doe'); // Person
let lily = new Employee('Lily Bush', 'Programmer'); // Employee
```

In this example, `new.target.name` is the human-friendly name of the constructor reference of `new.target`

In this tutorial, you have learned how to use the JavaScript `new.target` metaproperty to detect whether a function or constructor was called using the `new` operator.

JavaScript Static Methods

Introduction to the JavaScript static methods

By definition, static methods are bound to a class, not the instances of that class. Therefore, static methods are useful for defining helper or utility methods.

To define a static method before ES6, you add it directly to the constructor of the class. For example, suppose you have a Person type as follows:

```
function Person(name) {
    this.name = name;
}

Person.prototype.getName = function () {
    return this.name;
};
```

The following adds a static method called `createAnonymous()` to the `Person` type:

```
Person.createAnonymous = function (gender) {
    let name = gender == "male" ? "John Doe" : "Jane Doe";
    return new Person(name);
};
```

The `createAnonymous()` method is considered a static method because it doesn't depend on any instance of the `Person` type for its property values.

To call the createAnonymous() method, you use the `Person` type instead of its instances:

```
var anonymous = Person.createAnonymous();
```

JavaScript static methods in ES6

In ES6, you define `static` methods using the static keyword. The following example defines a static method called `createAnonymous()` for the `Person` class:

```
class Person {
    constructor(name) {
        this.name = name;
    }
    getName() {
        return this.name;
    }
    static createAnonymous(gender) {
        let name = gender == "male" ? "John Doe" : "Jane Doe";
        return new Person(name);
    }
}
```

To invoke the static method, you use the following syntax:

```
let anonymous = Person.createAnonymous("male");
```

If you attempt to call the static method from an instance of the class, you'll get an error. For example:

```
let person = new Person('James Doe');
let anonymous = person.createAnonymous("male");
```

Error:

```
TypeError: person.createAnonymous is not a function
```

Calling a static method from the class constructor or an instance method

To call a static method from a class constructor or an instance method, you use the class name, followed by the . and the static method:

```
className.staticMethodName();
```

Alternatively, you can use the following syntax:

```
this.constructor.staticMethodName();
```

Summary

- JavaScript static methods are shared among instances of a class. Therefore, they are bound to the class.
- Call the static methods via the class name, not the instances of that class.
- Use the `className.staticMethodName()` or this.`constructor.staticMethodName()` to call a static method in a class constructor or an instance method.

JavaScript Static Properties

Introduction to the JavaScript static properties

Like a static method, a static property is shared by all instances of a class. To define static property, you use the `static` keyword followed by the property name like this:

```
class Item {
  static count = 0;
}
```

To access a static property, you use the class name followed by the . operator and the static property name. For example:

```
console.log(Item.count); // 0
```

To access the static property in a static method, you use the class name followed by the `.` operator and the static property name. For example:

```
class Item {
  static count = 0;
  static getCount() {
    return Item.count;
  }
}

console.log(Item.getCount()); // 0
```

To access a static property in a class constructor or instance method, you use the following syntax:

```
className.staticPropertyName;
```

Or

```
this.constructor.staticPropertyName;
```

The following example increases the `count` static property in the class constructor:

```
class Item {
  constructor(name, quantity) {
    this.name = name;
    this.quantity = quantity;
    this.constructor.count++;
  }
  static count = 0;
  static getCount() {
    return Item.count;
  }
}
```

When you create a new instance of the `Item` class, the following statement increases the `count` static property by one:

```
this.constructor.count++;
Code language: CSS (css)
For example:

// Item class ...

let pen = new Item("Pen", 5);
let notebook = new Item("notebook", 10);

console.log(Item.getCount()); // 2
```

This example creates two instances of the `Item` class, which calls the class constructor. Since the class constructor increases the `count` property by one each time it's called, the value of the `count` is two.

Put it all together.

```
class Item {
  constructor(name, quantity) {
    this.name = name;
    this.quantity = quantity;
    this.constructor.count++;
  }
  static count = 0;
  static getCount() {
    return Item.count;
  }
}

let pen = new Item('Pen', 5);
let notebook = new Item('notebook', 10);

console.log(Item.getCount()); // 2
```

Summary

- A static property of a class is shared by all instances of that class.
- Use the `static` keyword to define a static property.
- Use the `className.staticPropertyName` to access the static property in a static method.
- Use the `this.constructor.staticPropertyName` or `className.staticPropertyName` to access the static property in a constructor.

JavaScript Private Fields

Introduction to the JavaScript private fields

ES2022 allows you to define private fields for a class. To define a private field, you prefix the field name with the `#` sign.

For example, the following defines the `Circle` class with a private field `radius`:

```
class Circle {
  `#radius`;
  constructor(value) {
    this.`#radius` = value;
  }
  get area() {
    return Math.PI * Math.pow(this.`#radius`, 2);
  }
}
```

In this example:

- First, define the private field `#radius` in the class body.
- Second, initialize the `#radius` field in the constructor with an argument.
- Third, calculate the area of the circle by accessing the `#radius` private field in the getter method.

The following creates a new instance of the `Circle` class and calculates its area:

```
let circle = new Circle(10);
console.log(circle.area); // 314.1592653589793
```

Because the `#radius` is a private field, you can only access it inside the `Circle` class. In other words, the `#radius` field is invisible outside of the `Circle` class.

Using getter and setter to access private fields

The following redefines the `Circle` class by adding the `radius` getter and setter to provide access to the `#radius` private field:

```
class Circle {
  `#radius` = 0;
  constructor(radius) {
    this.radius = radius; // calling setter
  }
  get area() {
    return Math.PI * Math.pow(this.`#radius`, 2);
  }
  set radius(value) {
    if (typeof value === 'number' && value > 0) {
      this.`#radius` = value;
    } else {
      throw 'The radius must be a positive number';
    }
  }
  get radius() {
    return this.`#radius`;
  }
}
```

How it works.

- The `radius` setter validates the argument before assigning it to the `#radius` private field. If the argument is not a positive number, the `radius` setter throws an error.
- The `radius` getter returns the value of the `#radius` private field.

- The constructor calls the radius setter to assign the argument to the #radius private field.

Private fields and subclasses

Private fields are only accessible inside the class where they're defined. Also, they're not accessible from the subclasses. For example, the following defines the Cylinder class that extends the Circle class:

```
class Cylinder extends Circle {
  #height;
  constructor(radius, height) {
    super(radius);
    this.#height = height;

    // cannot access the `#radius` of the Circle class here
  }
}
```

If you attempt to access the #radius private field in the Cylinder class, you'll get a SyntaxError.

The in operator: check private fields exist

To check if an object has a private field inside a class, you use the in operator:

fieldName in objectName For example, the following adds the hasRadius() static method to the Circle class that uses the in operator to check if the circle object has the #radius private field:

```
class Circle {
  `#radius` = 0;
  constructor(radius) {
    this.radius = radius;
  }
  get area() {
    return Math.PI * Math.pow(this.radius, 2);
  }
  set radius(value) {
    if (typeof value === 'number' && value > 0) {
      this.`#radius` = value;
    } else {
      throw 'The radius must be a positive number';
    }
  }
  get radius() {
    return this.`#radius`;
  }
  static hasRadius(circle) {
    return `#radius` in circle;
  }
}

let circle = new Circle(10);
```

```
console.log(Circle.hasRadius(circle));
```

Output:

```
true
```

Static private fields

The following example shows how to use a static private field:

```
class Circle {
  `#radius` = 0;
  static #count = 0;
  constructor(radius) {
    this.radius = radius; // calling setter
    Circle.#count++;
  }
  get area() {
    return Math.PI * Math.pow(this.radius, 2);
  }
  set radius(value) {
    if (typeof value === 'number' && value > 0) {
      this.`#radius` = value;
    } else {
      throw 'The radius must be a positive number';
    }
  }
  get radius() {
    return this.`#radius`;
  }
  static hasRadius(circle) {
    return `#radius` in circle;
  }
  static getCount() {
    return Circle.#count;
  }
}

let circles = [new Circle(10), new Circle(20), new Circle(30)];

console.log(Circle.getCount());
```

How it works.

First, add a private static field `#count` to the `Circle` class and initialize its value to zero:

```
static #count = 0;
```

Second, increase the `#count` by one in the constructor:

```
Circle.#count++;
```

Third, define a static method that returns the value of the `#count` private static field:

```
static getCount() {
    return Circle.#count;
}
```

Finally, create three instances of the `Circle` class and output the `count` value to the console:

```
let circles = [new Circle(10), new Circle(20), new Circle(30)];
console.log(Circle.getCount());
```

Summary

- Prefix the field name with `#` sign to make it private.
- Private fields are accessible only inside the class, not from outside of the class or subclasses.
- Use the `in` operator to check if an object has a private field.