Object-oriented Programming
in
JavaScript

JavaScript is (was) a class-free, object-oriented language, and as such, it uses prototypal inheritance instead of classical inheritance

Agenda

- Member functions
- The Constructor
- Prototype Inheritance
- Private Members
- Properties
- Classes in ES2015

Adding Functions to Objects

- We have used objects as loose aggregations of values
  - adding and altering their properties whenever we saw fit
- But in OOP objects can also have functionality
- One way to give an object methods is to simply attach function values to it:

```
var rabbit = {};
rabbit.speak = function (line) {
    console.log("The rabbit says '", line, "'");
};

rabbit.speak("Well, now you're asking me.");
```

The rabbit says 'Well, now you're asking me.'

Calling a Function as a Method

- There is a special variable called **this**, which is always present when a function is called
  - and which points at the relevant object when the function is called as a method
  - A function is called as a method when it is looked up as a property, and immediately called, as in `object.method()`

```
function speak(line) {
    console.log("The "+this.type+" rabbit says '"+line+ "'");
}
var whiteRabbit = {type: "white", speak: speak};
var fatRabbit = {type: "fat", speak: speak};

whiteRabbit.speak("Oh my ears and whiskers, how late it's
getting!");
fatRabbit.speak("I could sure use a carrot right now.");
speak("Spooky");
```

The white rabbit says 'Oh my ears and whiskers, how late it's getting!'
The fat rabbit says 'I could sure use a carrot right now.'
The undefined rabbit says 'Spooky'

apply() and call()

- The apply and call functions can be used to call a function on a specific object:

```
speak.apply(fatRabbit, ["Yum."]);

speak.call(fatRabbit, "Burp.");
```

Object to act on *Arguments*

- apply
  - you give the arguments for the function as an array
- call
  - you give the arguments for the function separately

THE CONSTRUCTOR

new

- Use the **new** keyword to create new objects
- When a function is called with the word new in front of it, its ***this*** variable will point at a *new* object
  - which the function will automatically return
    - unless it explicitly returns something else
- Functions used to create new objects are called **constructors**

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

var killerRabbit = new Rabbit("killer");
killerRabbit.speak("GRAAAAAAAAAH!");
```

  - It is a convention to start the names of constructors with a capital letter!

Why Use new?

- We could simply write this:

```
function makeRabbit(type) {
    return {
        type: type,
        speak: function (line) {/*etc*/ }
    };
}
var blackRabbit = makeRabbit("black");
```

- But that is not entirely the same
  The implicit property constructor is set to different functions:
  - For our killerRabbit the constructor property points at the Rabbit function that created it
  - For the blackRabbit the constructor property points at the Object function

```
console.log(killerRabbit.constructor);
console.log(blackRabbit.constructor);
```

```
<function Rabbit(type)>
<function Object()>
```

Prototypes

- Where did the constructor property come from?
  - It is part of the **prototype** of a rabbit
- Every object is based on a prototype, which gives it a set of inherent properties

```
var simpleObject = {};
console.log(simpleObject.constructor);
console.log(simpleObject.toString);
```

```
<function Object()>
<function toString()>
```

- You can use a constructor's prototype property to get access to their prototype:

```
console.log(Rabbit.prototype);
console.log(Rabbit.prototype.constructor);
```

```
{}
<function Rabbit(type)>
```

Object

- Most objects inherits some stuff from Object.prototype
  - Object is the build-in constructor for making new objects

```
var obj = {};
console.log(typeof obj.toString);
console.log(obj.toString === Object.prototype.toString);
console.log(Object.prototype);
```

"function"
True
{}

Public Members

- The members of an object are all *public* members
  - But may be hidden/internal
- There are two main ways of putting members in a new object:
  - **In the constructor:**
    - usually used to initialize public instance variables

```
function Rabbit(type) {
    this.type = type;
    this.size = 27;
}
```

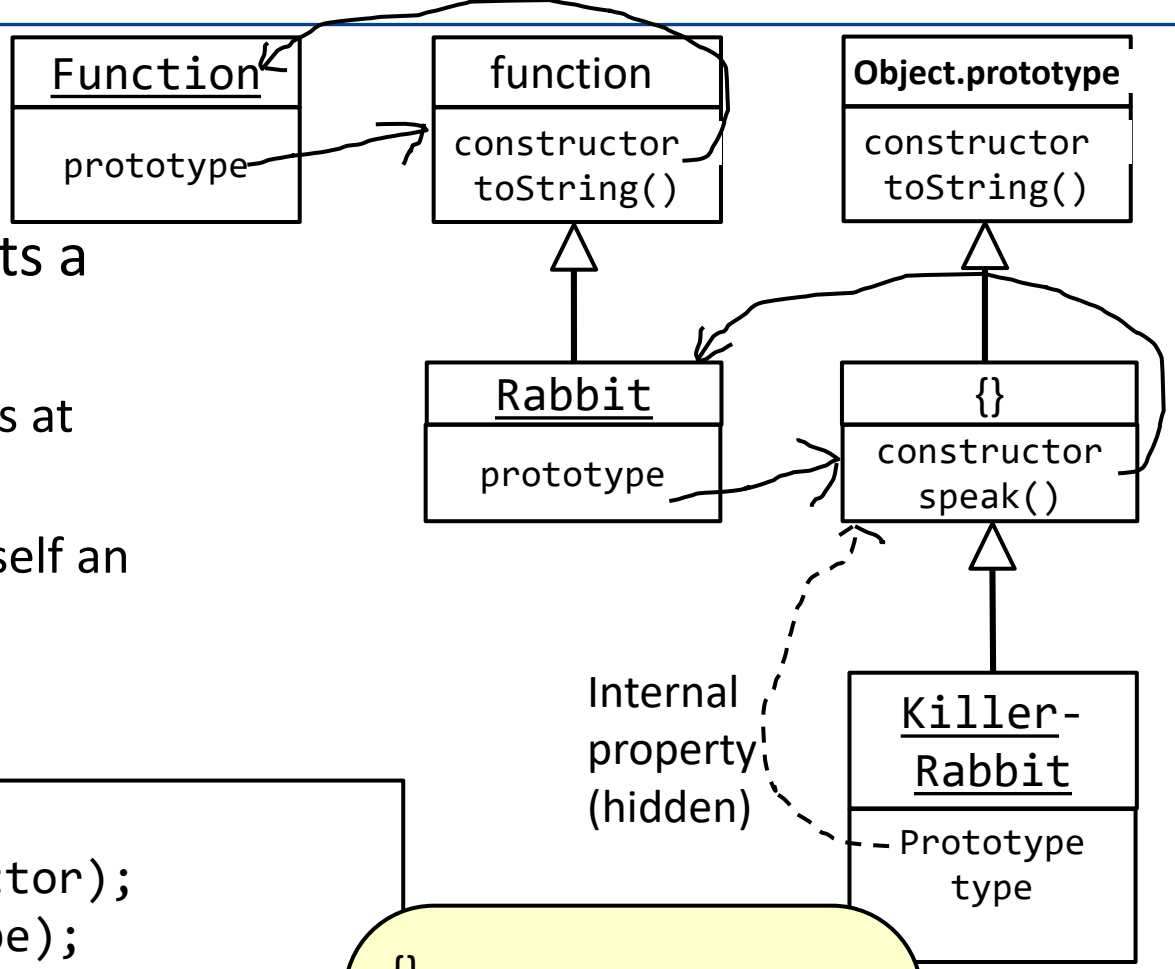
  - **In the prototype**
    - usually used to add public methods and static data

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

PROTOTYPE INHERITANCE

prototype property

- Every function automatically gets a prototype property
  - whose constructor property points at the constructor function
  - Because the rabbit prototype is itself an object, it is based on the Object prototype, and shares its toString method

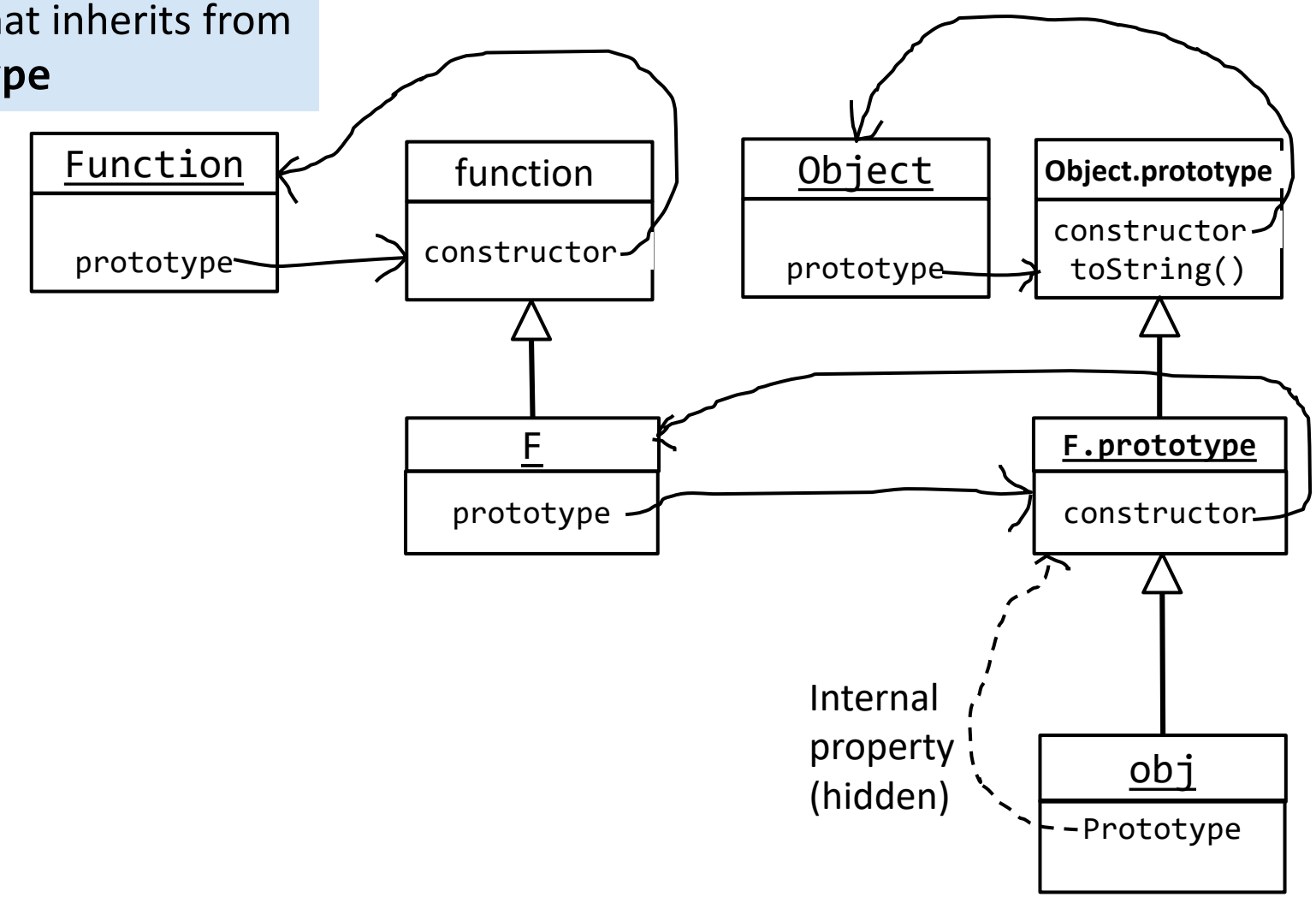



```
console.log(Rabbit.prototype);
console.log(Rabbit.prototype.constructor);
console.log(Rabbit.prototype.prototype);
console.log(Rabbit.constructor);
console.log(Rabbit.constructor.prototype);
console.log(killerRabbit.constructor);
console.log(killerRabbit.constructor.constructor);
console.log(killerRabbit.prototype);
```

```
{}
<function Rabbit(type)>
undefined
<function Function()>
<function ()>
<function Rabbit(type)>
<function Function()>
undefined
```

Prototype Inheritance

`var obj = new F()`
produces a new object that inherits from
`F.prototype`



Prototype Inheritance - *How it works*

- When looking up the value of a property:
  - JavaScript first looks at the properties that the object itself has
    - If there is a property that has the name we are looking for, that is the value we get
  - If there is no such property, it continues searching the prototype of the object
  - and then the prototype of the prototype, and so on
  - If no property is found, the value undefined is given
- When setting the value of a property:
  - JavaScript never goes to the prototype, but always sets the property in the object itself

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

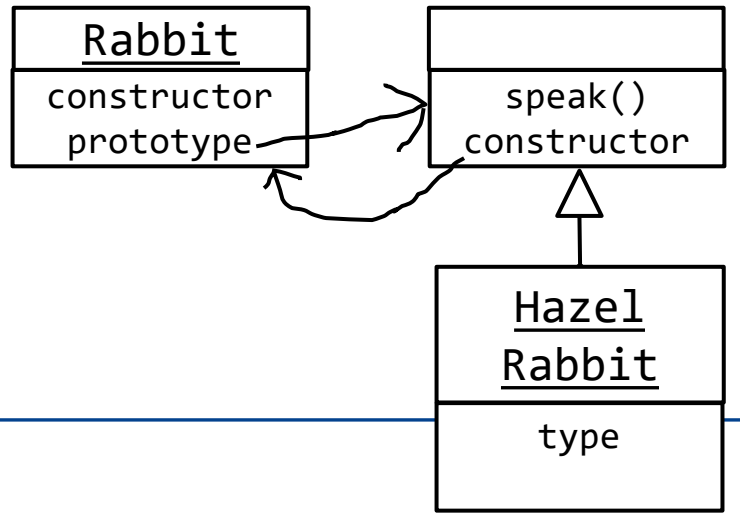
"small“

"long, sharp, and bloody"
"small"

Prototype Inheritance – How to Use

- The prototypical rabbit is the perfect place for values that all rabbits have in common, such as the speak method

```
function Rabbit(type) {
    this.type = type;
}
Rabbit.prototype.speak = function (line) {
    print("The ", this.type, " rabbit says '", line, "'");
};

var hazelRabbit = new Rabbit("hazel");
hazelRabbit.speak("Good Frith!");
```



Prototype Inheritance – Extensibility

- It can often be practical to extend the prototypes of standard constructors such as Object and Array with new useful functions
- For example, we could give all objects a method called properties, which returns an array with the names of the (non-hidden) properties that the object has:

```
Object.prototype.properties = function () {
    var result = [];
    for (var property in this)
        result.push(property);
    return result;
};

var test = { x: 10, y: 3 };
console.log(test.properties());
```

["x", "y", "properties"]

hasOwnProperty

- There is a way to find out whether a property belongs to the object itself or to one of its prototypes:
  - Every object has a method called hasOwnProperty , which tells us whether the object has a property with a given name

```
Object.prototype.properties = function () {
    var result = [];
    for (var property in this) {
        if (this.hasOwnProperty(property))
            result.push(property);
    }
    return result;
};

var test = { x: 10, y: 3 };
console.log(test.properties());
```

["x", "y"]

Prototypal Inheritance

- From EcmaScript v5 JavaScript has a build-in feature:

```
Object.create ( O [, Properties] )
```

- The **create** function creates a new object with a specified prototype
- When the **create** function is called, the following steps are taken:
  1. If Type(O) is not Object or Null throw a TypeError exception
  2. Let *obj* be the result of creating a new object
  3. Set the `Prototype` internal property of *obj* to O
  4. If the argument Properties is present, add own properties to obj
  5. Return *obj*

Object.create

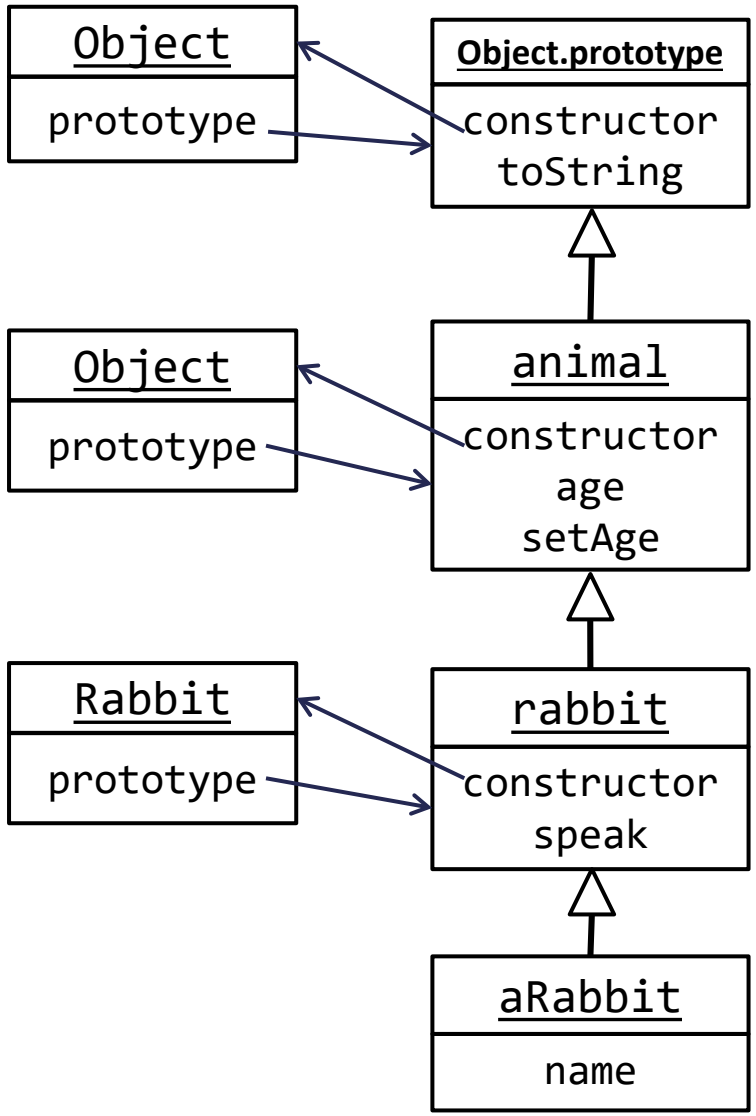
- You can use Object.create to create an object with a specific prototype

```
var protoRabbit = {
  speak: function(line) {
    console.log("The " + this.type + " rabbit says '" +
                line + "'");
  }
};
var killerRabbit = Object.create(protoRabbit);
killerRabbit.type = "killer";
killerRabbit.speak("SKREEEE!");
```

The killer rabbit says 'SKREEEE!'

Object.create Example - 1

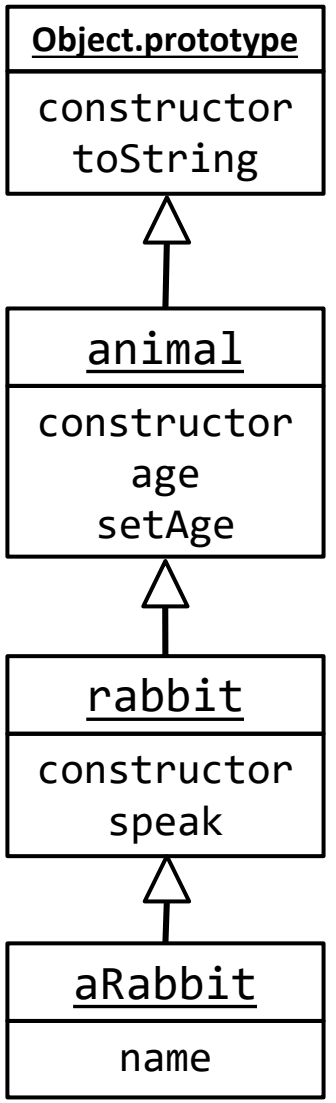
```
var animal = {
    age: null,
    setAge: function (age) {
        this.age = age;
    }
};

var rabbit = Object.create(animal);
rabbit.speak = function (line) {
    print("The ", this.name,
    " rabbit says '", line, "'");
};

function Rabbit(name) {
    this.name = name;
}

Rabbit.prototype = rabbit;
rabbit.constructor = Rabbit;

var aRabbit = new Rabbit("A");
```



Object.create Example - 2

```
var killerRabbit = new Rabbit("killer");
killerRabbit.speak("GRAAAAAAAAAH!");
killerRabbit.setAge(3);
console.log(killerRabbit.name);
console.log(killerRabbit.age);
console.log("----------------------");
for (var prop in killerRabbit)
    if (killerRabbit.hasOwnProperty(prop))
        console.log(prop);
console.log("----------------------");
for (var property in killerRabbit)
    console.log(prope
```

The killer rabbit says 'GRAAAAAAAAAH!'
"killer"
3
"----------"
"name"
"age"
"----------"
"name"
"age"
"speak"
"constructor"
"setAge"



Object.create(null)

- Can be used to create an object without Object.prototype properties:

```
var obj = Object.create(null);
console.log(obj.toString);
console.log(Object.prototype.isPrototypeOf(obj));
```

undefined
false

PRIVATE MEMBERS

Private

- *Private* members are made by the constructor
  - Ordinary vars and parameters of the constructor becomes the private members (because JavaScript has *closures*)
- They are attached to the object
  - but they are not accessible to the outside
  - nor are they accessible to the object's own public methods
  - They are only accessible to private methods and *privileged* methods
  - Private methods are inner functions of the constructor
  - A *privileged* method is able to access the private variables and methods, and is itself accessible to the public methods and the outside

Private Example

- This constructor makes three private instance variables:
  - param
  - secret
  - that
- They are attached to the object, but they are not accessible to the outside, nor are they accessible to the object's own public methods

```
function Container(param) {
    function dec() {
        if (secret > 0) {
            secret -= 1;
            return true;
        } else {
            return false;
        }
    }

    this.member = param;
    var secret = 3;
    var that = this;
}
```

- By convention, we make a private `that` variable
  - This is used to make the object available to the private methods
  - This is a workaround for an error in the ECMAScript Language Specification which causes this to be set incorrectly for inner functions

Privileged Methods

- To make private methods useful, we need a privileged method
- Privileged methods are assigned with *this* within the constructor
- A *privileged* method is able to access the private variables and methods, and is itself accessible to the public methods and the outside

- Private and privileged members can only be made when an object is constructed!

Privileged Method Example

```
function Container(param) {

    function dec() {
        if (secret > 0) {
            secret -= 1;
            return true;
        } else {
            return false;
        }
    }

    this.member = param;
    var secret = 3;
    var that = this;

    this.service = function () {
        return dec() ? that.member : null;
    };
}
```

PROPERTIES

JavaScript objects can have properties just like C# - but the syntax needed to define them is different

defineProperty

```
var obj = {};
Object.defineProperty(obj, "name", {
    value: "Spot", // default: undefined
    writable: true, // default: false
    enumerable: true, // default: false
    configurable: true // default: false
});
console.log(obj.name);
```

"Spot"

writable?

```
var obj = {};
Object.defineProperty(obj, "name", {
    value: "Spot", // default: undefined
    writable: false, // default: false
    enumerable: true, // default: false
    configurable: true // default: false
});
obj.name = "KilleRabbit";
console.log(obj.name);
```

"Spot"

getters and setters

In ES5 (and later):

```
var rect = {
    x: 5,
    y: 2
};
Object.defineProperty(rect, "area", {
    set: function (value) {
        throw new Error("Cannot set a value");
    },
    get: function () {
        return this.x * this.y;
    }
});
console.log(rect.area); // 10
rect.area = 20; // Exception: Error:
```

10
Error: Cannot set a value (line 8 in function area)

When a getter but no setter is defined, writing to the property is simply ignored

In ES6/2015:

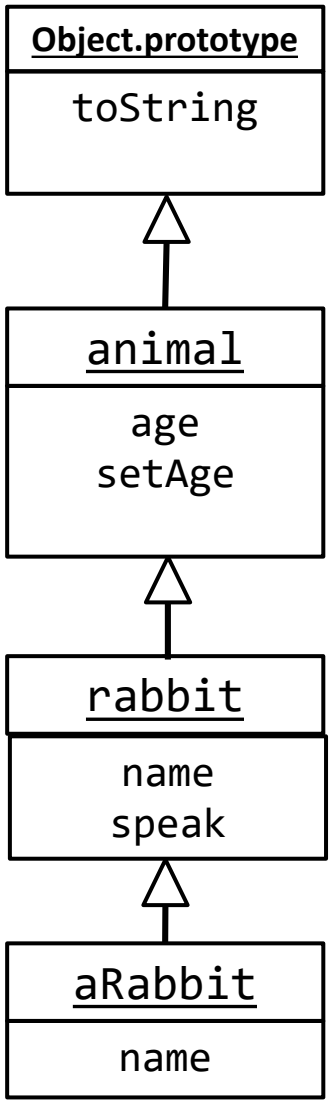
```
var rect = {
    x: 5,
    y: 2,
    set area(value) {
        throw new Error("Cannot set a value");
    },
    get area() {
        return this.x * this.y;
    }
});
console.log(rect.area); // 10
rect.area = 20; // Exception: Error: Cannot set a value
```

Object.defineProperties

```
var rect = {};
Object.defineProperties(rect, {
    "x": {
        value: 5,
        writable: true,
        enumerable: true
    },
    "y": {
        value: 2,
        writable: true,
        enumerable: true
    },
    "area": {
        set: function (value) {
            throw new Error("Cannot set a value");
        },
        get: function () {
            return this.x * this.y;
        }
    }
});
console.log(rect.area); // 10
```

Object.create Revisited

```
var animal = {
    age: null,
    setAge: function (age) {
        this.age = age;
    }
};
var rabbit = Object.create(animal, {
    name: {
        value: "",
        writable: true,
        enumerable: true
    },
    "speak": {
        value: function (line) {
            print("The ", this.name, " rabbit says '", line, "'");
        }
    }
});

var aRabbit = Object.create(rabbit);
aRabbit.name = "Killer";
aRabbit.speak("GRAAAAAAAAAH!");
```



CLASSES IN ES2015

ES2015 adds classes to JavaScript

Classes in ES2015

- Classes are a simple sugar over the prototype-based OO pattern
  - But having a single convenient declarative form makes class patterns easier to use, and encourages interoperability
- Classes in ES2015 support:
  - prototype-based inheritance
  - super calls
  - instance and static methods
  - constructors

Defining classes

- The class syntax has two components:
  - class expressions
  - class declarations
- Class declaration:

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
}
```

- Class expressions:

```
var Rectangle = class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
};
```

```
var Rectangle = class {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
};
```

Method Definitions

- Notice how we still set properties through this.property, but defining methods on the class is done very differently to how you might be used to:
  - Functions are defined by putting their name, followed by any arguments within brackets, and then a set of braces

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
  get area() {
    return this.calcArea();
  }
  calcArea() {
    return this.height * this.width;
  }
}

const square = new Rectangle (10, 10);
console.log(square.area);
console.log(square.calcArea());
```

Static methods

- The static keyword defines a static method for a class

```
class Point {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }

    static distance(a, b) {
        const dx = a.x - b.x;
        const dy = a.y - b.y;

        return Math.sqrt(dx*dx + dy*dy);
    }
}

const p1 = new Point(5, 5);
const p2 = new Point(10, 10);
console.log(Point.distance(p1, p2));
```

Sub classing

- The **extends** keyword is used to create a class as a child of another class

```
class Animal {
  constructor(name) {
    this.name = name;
  }

  speak() {
    console.log(this.name + ' makes a noise.');
  }
}

class Dog extends Animal {
  speak() {
    console.log(this.name + ' barks.');
  }
}

var collie = new Dog("Lassie");
collie.speak();
```

```
Lassie barks.
```

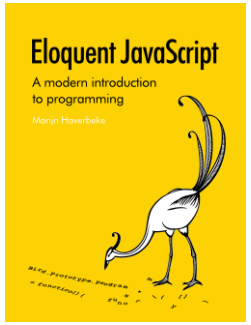
Sub classing traditional function-based "classes"

- You may also extend traditional function-based "classes"

```
function Animal (name) {
  this.name = name;
}
Animal.prototype.speak = function () {
  console.log(this.name + ' makes a noise.');
}

class Dog extends Animal {
  speak() {
    super.speak();
    console.log(this.name + ' barks.');
  }
}

var d = new Dog('Terry');
d.speak();
```

```
Terry makes a noise.
Terry barks.
```

References and Links



- **Eloquent JavaScript** by Marijn Haverbeke
  http://eloquentjavascript.net
- **ES2015 classes**
  https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Classes
- The JavaScript guru: **Douglas Crockford**'s blog
  http://javascript.crockford.com/private.html
- ECMA-262 ECMAScript Language Specification
  http://www.ecma-international.org/publications/standards/Ecma-262.htm
- JavaScript Patterns Collection
  http://shichuan.github.io/javascript-patterns/