Objects

Objects as Key/Value Pairs

- **Objects** are a common way to show related key-value pairs
- Allow you to associate a group of labeled data into a single structure

Object Syntax

- Enclose with `{ }`
- Keys (properties) can be strings or not

```
const dog = {
  'name': 'Freida',
  'color': 'brown/black',
  'hunger': 40,
  'mood': 'feisty',
  'age': 9,
};
```

```
const dog = {
  name: 'Freida',
  color: 'brown/black',
  hunger: 40,
  mood: 'feisty',
  age: 9,
};
```

Access values in an object

Can use **dot notation** or **square brackets**

```
const dog = {
      name: 'Freida',
      color: 'brown/black',
      hunger: 40,
      mood: 'feisty',
      age: 9,
    };

console.log(dog.name)    // prints Freida

console.log(dog['name']) // also prints Freida
```

Saving values to a variable

Sometimes it's useful to save a value to a variable so that you can use it outside of the object easily

```
const dog = {
  name: 'Freida',
  color: 'brown/black',
  hunger: 40,
  mood: 'feisty',
  age: 9
}

let freidaAge = dog.age
```

Accessing values using destructuring

We can also access values using a syntax called destructuring

```
const dog = {
  name: 'Freida',
  color: 'brown/black',
  hunger: 40,
  mood: 'feisty',
  age: 9
}

let {age} = dog
```

On the left side of the equals sign, you destructure the properties from the object, which goes on the right

We can also destructure multiple properties at a time

```
let {name, color, age} = dog

console.log(`${name} is a ${color} dog who is ${age} years old.`)
```

If you need to, you can also rename destructured properties in line

```
const dog = {
  name: 'Freida',
  color: 'brown/black',
  age: 9
}

const dog2 = {
  name: 'Buddy',
  color: 'gray',
  age: 1
}

const {age: freidaAge} = dog
```

```
const {age: buddyAge} = dog2

console.log(freidaAge) // 9
console.log(buddyAge) // 1
console.log(age) // undefined
```

Loop over properties in an object

```
// use for ... in to loop over object keys
for (let attribute in dog) {
  console.log(`The dog's ${attribute} is ${dog[attribute]}.`);
}
```

Add new key/value pair to existing object

Assign using **dot notation** or **square brackets**

```
const dog = {
  name: 'Freida',
  color: 'brown/black',
  hunger: 40,
  mood: 'feisty',
  age: 9,
};


dog.nickname = 'Puppers';     // dot notation

dog['nickname'] = 'Puppers'; // square brackets
```

Note: Map

Maps are a semi-recent addition to the JavaScript standard library.

In JavaScript, Maps are *ordered* collections of key-value pairs. They're used for key-value pairs that need to maintain order, like transaction histories.

Removing a value from an object

We can remove values from object using the `delete` keyword

```
const dog = {
  name: 'Freida',
  color: 'brown/black',
  hunger: 40,
  mood: 'feisty',
  age: 9,
};

delete dog.hunger
```

Classes

Benefits of objects

- Flexible - all ***dog*** objects don't have to have the same properties
- Can write functions, loops to do useful things with objects

```
let dog1 = {
  'name': 'Freida',
  'age': 9,
  'color': 'brown'
};

let dog2 = {
  'name': 'Libby',
  'age': 3
  'nickname': 'Libbers'
};
```

```
function bark(dog){
  console.log('Arf! I am', dog.name, 'the dog!');
}
```

Issues with making your own objects

- It's helpful for all dog objects to have the **exact same** properties
  - Prevent bugs in your code
- It would be nice if you could store data about dogs in the same place as dog **behaviors**
  - Things that dog objects know how to do, like ***bark***

Enter… classes!

- A **class** is like a little mini factory that knows how to make objects of a single type
- Let's you define the **blueprint** for future objects
  - For example, all dogs have properties color, name, hunger, and mood
- Also lets you define object behaviors, or **methods**

Example

```
class Dog {
    constructor(name, color){
      this.name = name;
      this.color = color;
    }

    bark(){
      console.log('Arf! I am', this.name, 'the dog!');
    }
}
```

- ***Dog*** class defines the template or blueprint for **all** dogs
- ***constructor*** allows you to assign data specific to each individual dog
- ***bark*** is the behavior, or method, that each dog knows
  - `this.name` means “get the name of whatever dog is trying to bark right now”

Making Objects with Classes

```
class Dog {
    // ...
}


let dog1 = new Dog('Freida', 'brown');
let dog2 = new Dog('Sally', 'pink');

console.log(dog1.name)  // Freida
console.log(dog2.name)  // Sally

dog1.bark()    // Arf, I'm Freida the dog!
dog2.bark()    // Arf, I'm Sally the dog!
```

- ***dog1*** and ***dog2*** are **instances** of the class ***Dog***
- ***dog1*** and ***dog2*** are also **objects** of the type ***Dog***
- When you make a dog object using the ***Dog*** class, it's called **instantiation**

Extending Classes

- Making a new class based on an existing class
- Can add extra details or functionality to the extended class
- Uses the `extends` keyword

```
class Dog {
  ...
}

class Puppy extends Dog {
  ...
}
```

- Extended classes can inherit properties and methods from their predecessors
- For example, any instances of ***Puppy*** will have a ***bark*** method

```
class Dog {
  ...
}

class Puppy extends Dog {
  constructor(name, color, trainingLevel) {
    super(name, color)

    this.trainingLevel = trainingLevel
  }

  levelUp(num) {
    this.trainingLevel += num
  }
}
```

- Invoke the parent's `constructor` using the `super` method
- `super` is required, extended classes won't work without it

- Properties and methods added to extended classes are **not** available on their parents (no `levelUp` method on ***Dog*** objects)

The End