More on Objects, Methods, and Intro to OOP

Reuben Simpson

Quick Recap

Datatypes, Object

Accessing object properties - Dot operator, Square bracket notation

Adding/removing/modifying properties

Iterating over Objects

Datatypes

- Describes the different types or kinds of data that you will be working with and storing in variables.
- The main datatypes in JavaScript include
  - String type
  - Number type
  - Boolean type
  - Objects
  - Arrays

Objects

- Objects are an ***unordered*** collections of **key/value pairs**, where the
  - **Keys** are strings
  - **Values** can be any type, even other objects.
- Objects are defined by the list of pairs key: value, comma-separated and enclosed by curly braces.

```
const person = {
    firstName: 'John',
    lastName: 'Doe'
};
```

Dot Operator

```
objectName.propertyName
```

The dot notation can be used to access the property of an object.

For example, to access the firstName property of the person object, you use the following expression:

```
person.firstName
```

Square bracket notation - [ ]

The square brackets ***property accessor*** has the following syntax.

```
objectName['propertyName']
```

To access the value of an object's property via the array-like/square bracket notation, we use

```
person['firstName']
```

Adding/Removing an object property

- A JavaScript object is a collection of unordered properties.
  - Properties are the values associated with a JavaScript object.
- You can ***add new properties*** to an existing object by simply giving it a value.

```
person.favouriteColour = "Purple";
```

- The **`delete`** keyword deletes a property from an object.
  - The delete keyword deletes both the value of the property and the property itself.

```
delete person.lastname
```

Iterating over properties of an object

- The **`for...in`** statement iterates over the properties of an object.

```
const user = {
   name: "John",
   age: 5,
   isAdmin: true
};

for (const key in user) {
      console.log(key); // name, age, isAdmin
      console.log(user[key]); // John, 5, true
}
```

Functions

- A function is a block of organized, reusable lines of code that is used to perform a single, related action.
- A function definition (also called a function declaration, or function statement) consists of the **function keyword**, followed by:
  - The ***name*** of the function.
  - A ***list of parameters*** to the function, enclosed in parentheses and separated by commas.
  - The JavaScript ***statements*** that define the function, enclosed in curly brackets, { . . . } .

```
function square(num) {
 return num * num;
}
```

What we'll look at today

- Execution Context & Call Stack
- More on Methods
  - The `this` keyword
  - The `new` keyword
- Object oriented programming
  - Classes & Objects
  - Constructors

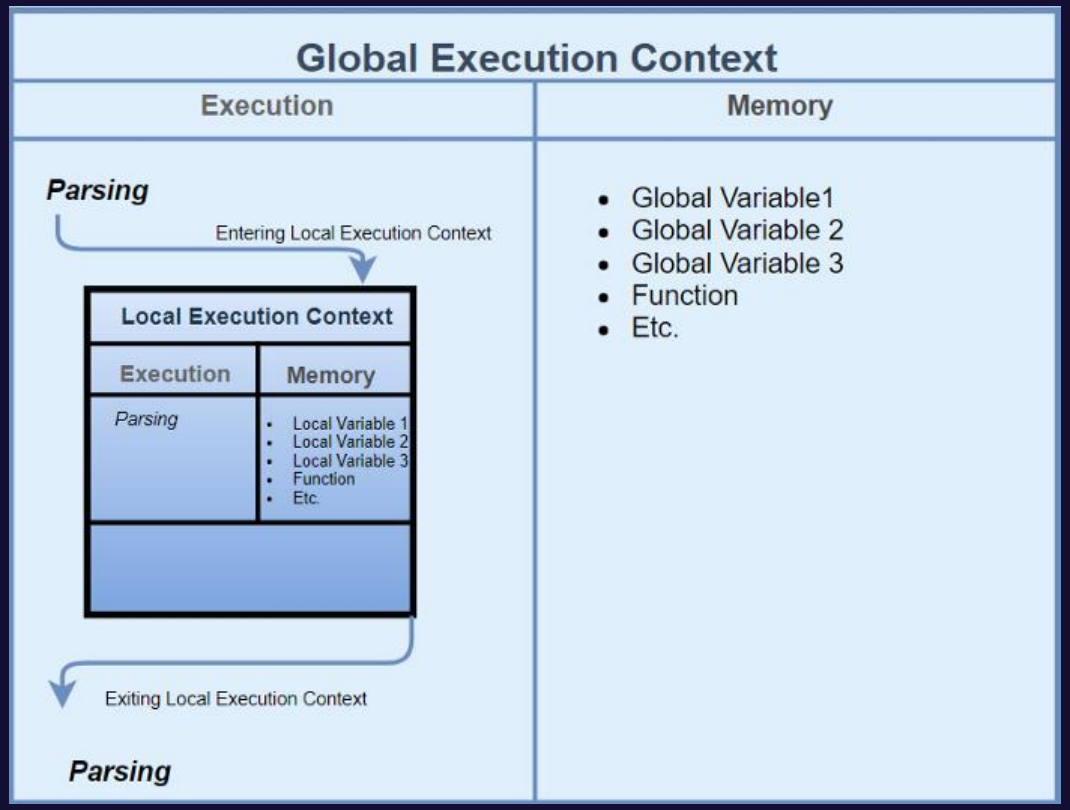
Execution contexts

- Execution context is the *environment* in which JavaScript code is evaluated and executed
  - All code in JavaScript runs inside an execution context
  - Ultimately; what happens inside the execution context is the parsing of code line by line and the storing of variables and functions into memory.
- There are two types of *context* in JavaScript:
  - Global context
  - Function/Local context

https://javascript.plainenglish.io/understanding-javascript-execution-context-call-stack-and-stack-overflow-157b7b358e88

Global Execution Context

- It is the first thing that is created when you write JavaScript code.
  - It is the default context.
- When the JS engine starts reading your code, it creates the global execution context.
  - It starts parsing line by line and adds your variables to memory also known as *global variable environment*.



```
const x = 5;
function addOne(num) {
  const answer = num + 1;
  return answer;
}
addOne(x)
```

Global Execution Context	
Execution (thread of execution)	Memory (variable environment)
	x : undefined; addOne : { *fn's codeblock* }

Let, const and var declarations
Function declarations

Local Execution Context

- While the JavaScript engine is parsing, if it needs to ***execute*** a function, a new local execution context is created.
  - In that execution context, parsing takes place and the number variable is added to ***local*** memory, and then parsing continues.
  - After this, the engine returns to the previous execution context.
- Exiting the local execution context and continuing parsing in the previous execution context is achieved with the return keyword.
- Every time a function gets called, this happens again.
  - Every function call results in a different local execution context.

```
const x = 5;
function addOne(num) {
  const answer = num + 1;
  return answer;
}
addOne(x)
```

Global Execution Context

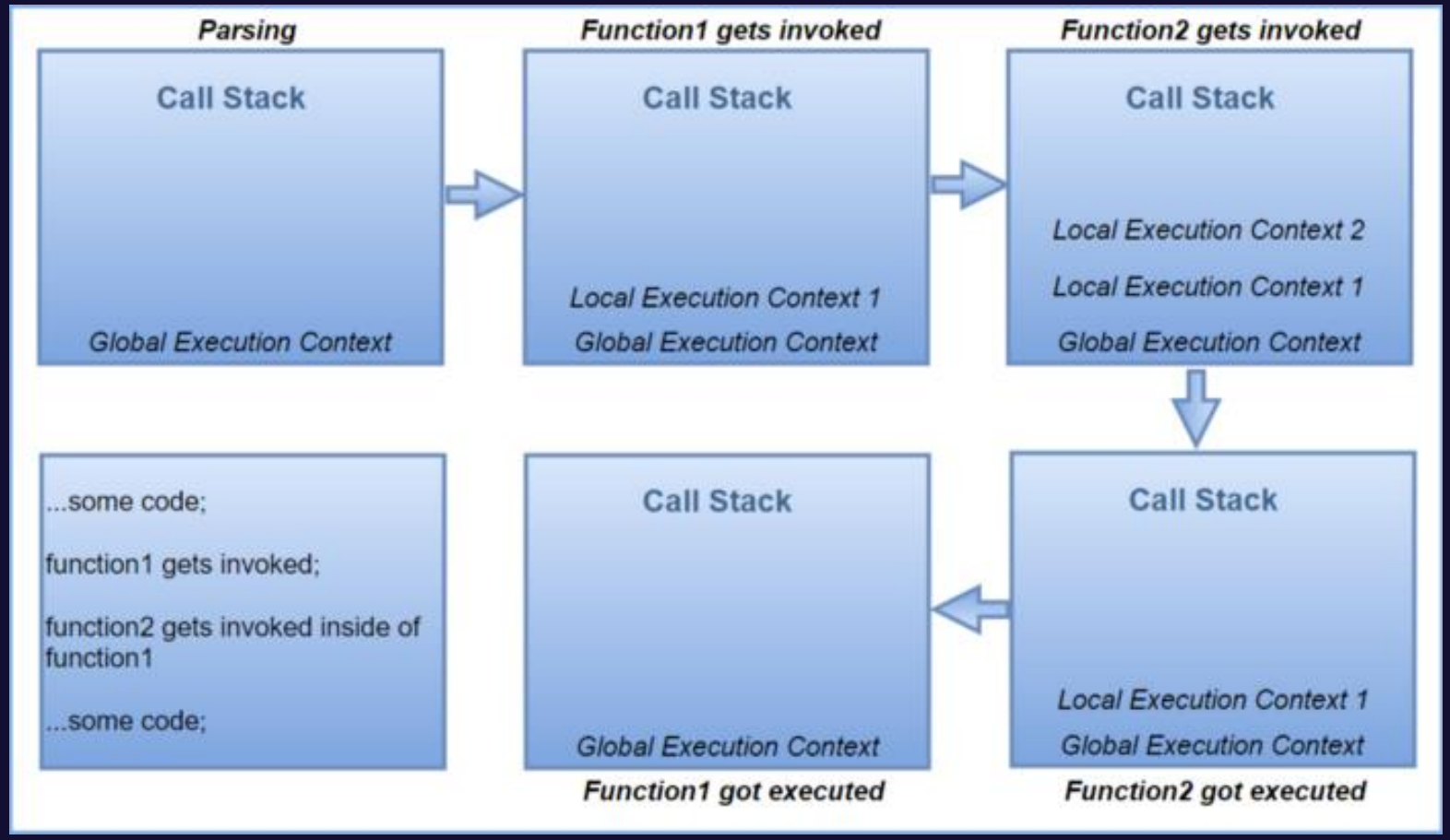
Execution (thread of execution)	Memory (variable environment)
	x : undefined ↓ x : 5 addOne : { *fn's codeblock* } ↓ addOne : 6

Local Execution Context (addOne)

Execution	Memory
num = 5 answer = 5+1	num : 5 answer : 6

The Call Stack

- The Call Stack is a mechanism for the JavaScript Engine to ***keep track of execution contexts***, which to enter, which to exit or which to return to.
- At the bottom of the stack is the global execution context



http://latentflip.com/loupe/

`this` keyword

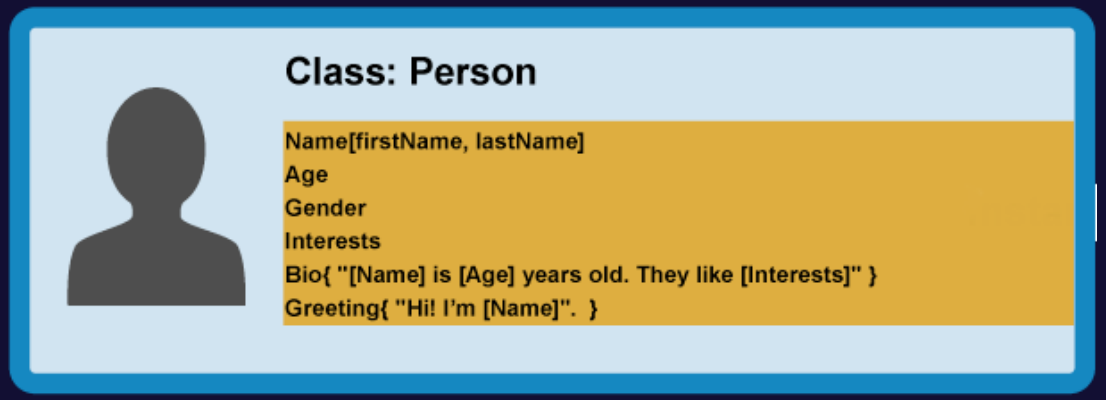
- ***`this`*** refers to an object to which the currently executing code belongs.
  - Every execution context provides the *this* variable.
- The value of *this* is determined by how a function is called (runtime binding).
  - The global execution context has the global object. ***`this`*** variable refers to the global object, when used in the global execution context.
  - Within a local execution context, the ***`this`*** variable refers to the function object depending on how the function was invoked.

```
// In web browsers, the window object is also the global object:
console.log(this === window); // true
a = 37; // Assigning a value to the window object
console.log(window.a); // 37
this.b = "MDN";
console.log(window.b); // "MDN"
console.log(b); // "MDN"
```

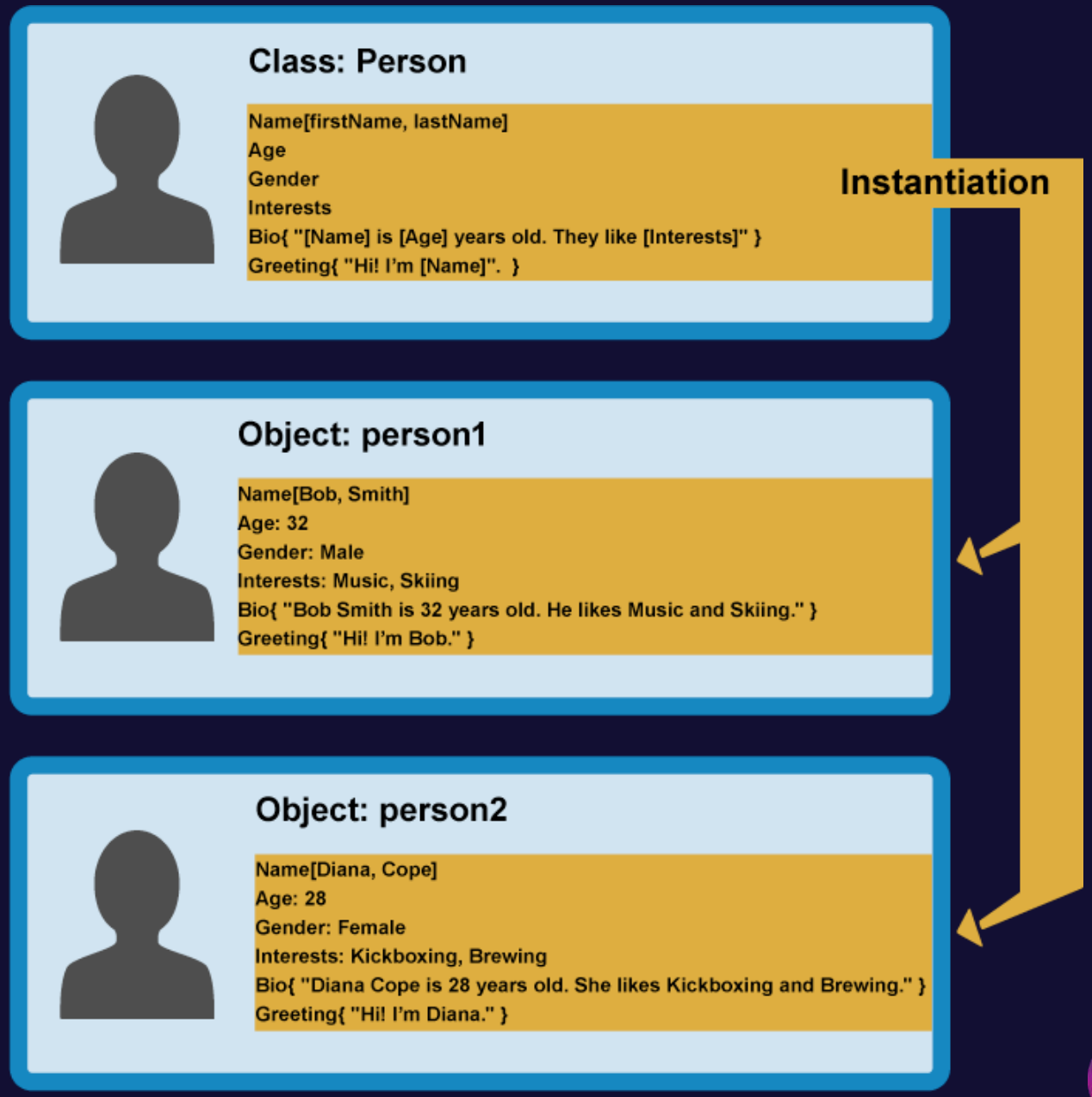
```
const person = {
  firstName: "Rob",
  lastName: "Petrie",
  greetings: function () {
    console.log("Hello", this.firstName);
  },
};

person.greetings();
```

Object Oriented Programming

- The basic idea of OOP is that we ***use objects to model real world things*** that we want to represent inside our programs, and/or provide a simple way to access functionality that would otherwise be hard or impossible to make use of.
- Classes are a ***template*** for creating objects. They encapsulate data with code to work on that data



- **Objects** can contain related ***data*** and ***code***, which represent information about the thing you are trying to model, and functionality or behavior that you want it to have.
  - So data and functions (attributes and methods) are bundled within an object.



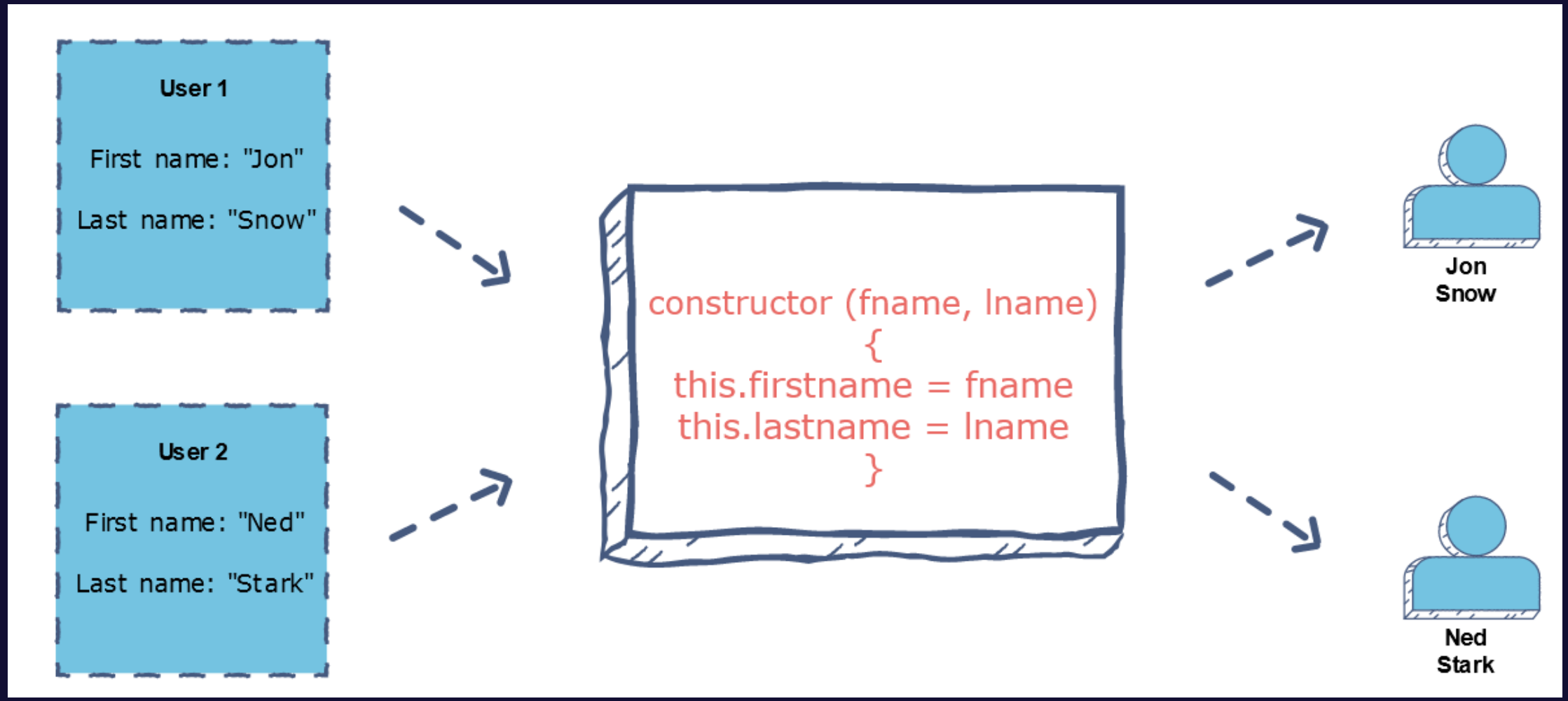
Classes in JavaScript

- A class is an extensible ***template*** for creating objects, providing initial values for state (member variables) and implementations of behavior (member functions or methods).
  - Classes are templates for JavaScript Objects.
  - Encapsulates data
  - This is different from the “class” in html/css
  - A Class name starts with capital letter
- We use the keyword `class` to create a class.

```
class ClassName {
        constructor() {
            // ...
        }
    }
```

Constructors and the *new* keyword

- A constructor is a ***function*** that creates an instance of a class which is typically called an “object”.
  - In JavaScript, a constructor gets called when you declare an object using the ***new*** keyword.
- The purpose of a constructor is to
  1. **Create** an object and
  2. **Set values** if there are any object properties present.


User 1
First name: "Jon"
Last name: "Snow"
User 2
First name: "Ned"
Last name: "Stark"
constructor (fname, lname)
{
this.firstname = fname
this.lastname = lname
}
Jon
Snow
Ned
Stark

Constructors

- The constructor method is a ***special method***:
  1. It has to have the exact name "constructor"
  2. It is executed automatically when a **`new`** object is created
  3. It is used to initialize object properties

In JavaScript, here's what happens when a constructor is invoked:

- A new empty object is created
- this keyword starts referring to that newly created object and hence it becomes the current instance object
- The newly created object is then returned as the constructor's returned value

Example 1

```
class User {
  constructor(first, last) {
    this.firstName = first;
    this.lastName = last;
  }
}

const user1 = new User("Buddy", "Sorrell");
console.log(user1);
console.log(user1.firstName, user1.lastName);

const user2 = new User("Sally", "Rogers");
console.log(user2);
console.log(user2.firstName, user2.lastName);
```

Two ways to declare a Class

Using function approach

```
function Country(name, traveled) {
  this.name = name ? name : "New Zealand";
  this.traveled = traveled;
}

Country.prototype.travel = function () {
  this.traveled = true;
}; // Create a method on the Country Object

// Create an instance of France
const france = new Country("France", false);
france.travel(); // Travel to France
```

Using Class approach

```
class Country {
  constructor(name, traveled) {
    this.name = name ? name : "New Zealand";
    this.traveled = traveled;
  }
  travel() {
    this.traveled = true;
  } // Create a method on the Country Object
}
// Create an instance of Australia
const australia = new Country("Australia", false);
australia.travel(); // Travel to Australia
```

Example 2

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

  introduce() {
    console.log("Hello, my name is " + this.name);
  }
}

const rob = new Person("Rob");
rob.introduce();
```

Exercise - 1

1. Create a Vehicle class
2. Add a constructor method to initialize all the values
   - Model
   - Manufacturer
   - Year of Manufacture
   - Colour
3. Add a method to print the colour of the car
4. Using the new keyword, create a few instances and run the method to print its properties.

Inheritance

- Keywords to remember:
  - extends
  - super()
- Inheritance
- Polymorphism

```
class City {
  constructor(name, traveled) {
    this.name = name;
    this.traveled = traveled;
  }

  travel() {
    // Will only get called if child does not have travel() method
    this.traveled = true;
  }
}

class CoastalCity extends City {
  constructor(name, traveled, coastLineLength) {
    super(name, traveled); // Calls the parent constructor
    this.coastLineLength = coastLineLength;
  }
  visitBeach() {
    // Child can have its own method parent doesn't have
    console.log("The coast line is " + this.coastLineLength + "km long.");
  }
  travel() {
    // Polymorphism - implements differently from parent method
    this.traveled = true;
    console.log("The coast line is beautiful");
  }
}

// Constructor invocation
const tau = new CoastalCity("Tauranga", false, 20);
tau.travel(); // prints "The coast line is beautiful"
tau.visitBeach();
```

MISSION READY

www.missionreadyhq.com

DARE TO DEVELOP

Thank you

Reuben Simpson