Title: Write a note on Constructor Functions

Answer:

Constructor functions are functions that are used to construct new objects. The `new` operator (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/new) is used to create new instances based off a constructor function. We have seen some built-in JavaScript constructors, such as `new Array()` and `new Date()`, but we can also create our own custom templates from which to build new objects.

As an example, let's say we are creating a very simple, text-based role-playing game. A user can select a character and then choose what character class they will have, such as warrior, healer, thief, and so on.

Since each character will share many characteristics, such as having a name, a level, and hit points, it makes sense to create a constructor as a template. However, since each character class may have vastly different abilities, we want to make sure each character only has access to their own abilities. Let's take a look at how we can accomplish this with prototype inheritance and constructors.

To begin, a constructor function is just a regular function. It becomes a constructor when it is called on by an instance with the `new` keyword. In JavaScript, we capitalize the first letter of a constructor function by convention.

characterSelect.js

```
// Initialize a constructor function for a new Hero
function Hero(name, level) {
  this.name = name;
  this.level = level;
}
```

We have created a constructor function called `Hero` with two parameters: `name` and `level`. Since every character will have a name and a level, it makes sense for each new character to have these properties. The `this` keyword will refer to the new instance that is created, so setting `this.name` to the `name` parameter ensures the new object will have a `name` property set.

Now we can create a new instance with `new`.

```
let hero1 = new Hero('Bjorn', 1);
```

If we console out `hero1`, we will see a new object has been created with the new properties set as expected.

```
 Output
Hero {name: "Bjorn", level: 1}
```

Now if we get the `[[Prototype]]` of `hero1`, we will be able to see the `constructor` as `Hero()`. (Remember, this has the same input as `hero1.__proto__`, but is the proper method to use.)

```
Object.getPrototypeOf(hero1);

Output
constructor: ƒ Hero(name, level)
```

You may notice that we've only defined properties and not methods in the constructor. It is a common practice in JavaScript to define methods on the prototype for increased efficiency and code readability.

We can add a method to `Hero` using `prototype`. We'll create a `greet()` method.

characterSelect.js

```
...
// Add greet method to the Hero prototype
Hero.prototype.greet = function () {
  return `${this.name} says hello.`;
}
```

Since `greet()` is in the `prototype` of `Hero`, and `hero1` is an instance of `Hero`, the method is available to `hero1`.

```
hero1.greet();

Output
"Bjorn says hello."
```

If you inspect the `[[Prototype]]` of Hero, you will see `greet()` as an available option now.

This is good, but now we want to create character classes for the heroes to use. It wouldn't make sense to put all the abilities for every class into the `Hero` constructor, because different classes will have different abilities. We want to create new constructor functions, but we also want them to be connected to the original `Hero`.

We can use the `call()` (https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/call) method to copy over properties from one constructor into another constructor. Let's create a Warrior and a Healer constructor.

characterSelect.js

```
...
// Initialize Warrior constructor
function Warrior(name, level, weapon) {
  // Chain constructor with call
  Hero.call(this, name, level);

  // Add a new property
  this.weapon = weapon;
}

// Initialize Healer constructor
function Healer(name, level, spell) {
  Hero.call(this, name, level);

  this.spell = spell;
}
```

Both new constructors now have the properties of `Hero` and a few unqiue ones. We'll add the `attack()` method to `Warrior`, and the `heal()` method to `Healer`.

characterSelect.js

```
...
Warrior.prototype.attack = function () {
  return `${this.name} attacks with the ${this.weapon}.`;
}

Healer.prototype.heal = function () {
  return `${this.name} casts ${this.spell}.`;
}
```

At this point, we'll create our characters with the two new character classes available.

characterSelect.js

```
const hero1 = new Warrior('Bjorn', 1, 'axe');
const hero2 = new Healer('Kanin', 1, 'cure');
```

`hero1` is now recognized as a `Warrior` with the new properties.

```
Output
Warrior {name: "Bjorn", level: 1, weapon: "axe"}
```

We can use the new methods we set on the `Warrior` prototype.

```
hero1.attack();
```

```
Console
"Bjorn attacks with the axe."
```

But what happens if we try to use methods further down the prototype chain?

```
hero1.greet();
```

```
Output
Uncaught TypeError: hero1.greet is not a function
```

Prototype properties and methods are not automatically linked when you use `call()` to chain constructors. We will use `Object.create()` to link the prototypes, making sure to put it before any additional methods are created and added to the prototype.

characterSelect.js

```
...
Warrior.prototype = Object.create(Hero.prototype);
Healer.prototype = Object.create(Hero.prototype);

// All other prototype methods added below
...
```

Now we can successfully use prototype methods from `Hero` on an instance of a `Warrior` or `Healer`.

```
hero1.greet();
```

```
Output
"Bjorn says hello."
```

Here is the full code for our character creation page.

characterSelect.js

```
// Initialize constructor functions
function Hero(name, level) {
  this.name = name;
  this.level = level;
}

function Warrior(name, level, weapon) {
  Hero.call(this, name, level);

  this.weapon = weapon;
}

function Healer(name, level, spell) {
  Hero.call(this, name, level);

  this.spell = spell;
}

// Link prototypes and add prototype methods
Warrior.prototype = Object.create(Hero.prototype);
Healer.prototype = Object.create(Hero.prototype);

Hero.prototype.greet = function () {
  return `${this.name} says hello.`;
}

Warrior.prototype.attack = function () {
  return `${this.name} attacks with the ${this.weapon}.`;
}

Healer.prototype.heal = function () {
  return `${this.name} casts ${this.spell}.`;
}

// Initialize individual character instances
const hero1 = new Warrior('Bjorn', 1, 'axe');
const hero2 = new Healer('Kanin', 1, 'cure');
```

With this code we've created our `Hero` class with the base properties, created two character classes called `Warrior` and `Healer` from the original constructor, added methods to the prototypes and created individual character instances.

Tags: functions / methods, javascript