# CPSC 1045: Objects

We have used many objects throughout the course. One is the drawing context object, which we often named **ctx.** Like all object **ctx** contains methods and properties. Methods are just code that is associated with the object. We have used ctx.beginPath(), ctx.translate(), ctx.save(). These are all methods that trigger some code to execute. Objects also have Properties. These are variables associated with an object, such as **.innerHTML** or **.value**. We can obtain information from these properties, or we can set the properties to some value.

We can create our own objects and there are two methods what we will describe.

1. Literal objects
2. Using constructors

## Creating literal objects

A literal object is like a literal string, or a literal number. We describe in JavaScript the object. To create a literal object we use what is called JavaScript object notation or JSON for short. The template as follows:

```
var <objectname> = {
    <property name> : <property value>,
        //More literal values
    <last property name> : <last property value>
}
```

The definition is a list of name/value pairs each separated by a comma. So, if I want to create an object called **square,** where the properties color is equal to blue and size is equal to 6 we would write the following:

```
var square = {
    color : “blue”,
    size : 6
}
```

## Creating objects with constructor

A constructor is a function used to create objects. On the surface it begins like any other function, and the main difference is in how we use the function. It’s a good idea to put comment on top of the constructor to denote the fact and have the name of the function begin with a capital letter. The reason we would use a constructor over JSON is because constructors allows us to create multiple similar objects using parameters. That is an object with the same properties but different values for the properties. This helps us because once we have tested that our constructor works and creates the object with the correct properties we no longer have to worry that our properties names are incorrect.

So, we wanted to create objects that have properties of **color** and **size** we would write the following constructor.

```
//Constructor to create square objects
function SquareConstructor( color, size){
   this.color = color;
   this.size = size;
}
```

Notice the **this** keyword. This refers to the object, and **this.color** refers the color property in the object that we are creating. The **this** statement will create the property in the object and assign it whatever is stored in the color parameter. Similarly **this.size** creates the property in the object and assign it whatever is stored in the size parameter.

To create an object with a constructor we use the **new** keyword. To create the same object as the above example we would write:

```
var blueSquare = new SquareConstructor(“blue”,6);
```

The advantage of using the constructor is that we can use it to create other objects like a red square:

```
var redSquare = new SquareConstructor(“red”,6);
```

## Array of objects using constructors

If we use the constructor we just wrote above, we can also use it in array creation as follows:

```
var mySquareArray = [new squareConstructor(“red”,6),
                     new squareConstructor(“yellow”,6),
                     new squareConstructor(“green”,7),
                     new squareConstructor(“orange”,9)];
```

## Adding objects to an array

Similarly, we can add an object to an array by using the push method. We can do this in two ways.

**Method 1:**

```
var myObject = new squareConstructor("magenta",2);
mySquareArray.push(myObject);
```

**Method 2:**

```
mySquareArray.push(new squareConstructor("purple",100);
```

## Accessing object Properties

The access properties in an object we use the dot like we have done in the pass.
If we wanted to access the color value of the blueSquare we created above we would write:

```
blueSquare.color
```

We have seen this pattern many times:
eg:

```
element.innerHTML
```

or

```
element.value
```

## A simple web-app that uses objects:

Let's create a simple photo-viewing app. Each object will represent the photo, containing a URL to the photo

Our constructor will create an object with a URL, short description, width and height. We will have a global array of objects, and our next and previous buttons will simply cycle through the array.

```
function CreateImage(src, blurb, width, height) {
    "use strict";
    this.src = src;
    this.blurb = blurb;
    this.width = width;
    this.height = height;
}
```

The above code is the constructor. As mentioned earlier, we use the **this** keyword to refer to the object that we are creating. In the constructor **this.src** refers to the object's source property, while **src** by itself refers to the parameter.

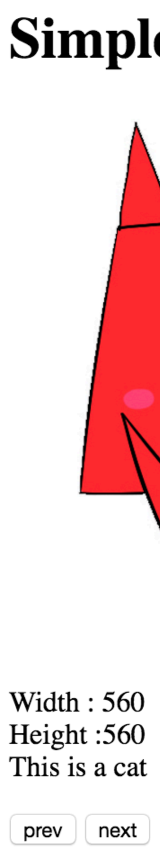
To create an simple photo-viewing app, we need

- a global array to hold our object representing the images
- a global variable representing the current image
- A HTML page to
  - display the image
  - Buttons to navigate the images

The HTML file for the photo-viewer is as follows.

```
<!DOCTYPE html>
<head>
   <title>Imageviewer</title>
   <meta charset="UTF-8">
   <script src="iv.js"></script>
</head>
<body onload = "setup()">
   <h1>Simple image gallery</h1>
    <div>
       <img id="pic" src="" width="300" height="300">
       <p id = "blurb">Text block</p>
    </div>
    <input type="button" value="prev" onclick="prev()">
    <input type="button" value="next" onclick="next()">
</body>
```

Which initially might look like the following:



Simple image gallery

Width : 560
Height :560
This is a cat

The image has been hard coded into the JavaScript for this example as a matter of convenience. The next button will display the next image, and the previous button will display the previous image.

The JavaScript is below. If you want try this independently, you will need to the change the image URLs.

```
/*jslint
  vars : true
*/
var imgArray = [];
var currentIndex = 0;
function updateOutput(imageToSee) {
    "use strict";
    var imgElement = document.getElementById("pic");
    var blurbElement = document.getElementById("blurb");

    imgElement.src = imageToSee.src;
    var text = "Width : " + imageToSee.width + "<br>" +
        "Height :" + imageToSee.height + "<br>" +
        imageToSee.blurb;
    blurbElement.innerHTML = text;
}
function CreateImage(src, blurb, width, height) {
    "use strict";
    this.src = src;
    this.blurb = blurb;
    this.width = width;
    this.height = height;
}
function next() {
    "use strict";
    currentIndex = currentIndex + 1;
    if (currentIndex >= imgArray.length) {
        currentIndex = 0;
    }
    updateOutput(imgArray[currentIndex]);
}
function prev() {
    "use strict";
    currentIndex = currentIndex - 1;
    if (currentIndex < 0) {
        currentIndex = imgArray.length - 1;
    }
    updateOutput(imgArray[currentIndex]);
}

function setup() {
    "use strict";
    //create the array
    //need images
    imgArray.push(new CreateImage("cat.jpg", "This is a
cat", 560, 560));
    imgArray.push(new CreateImage("ball.jpg", "This is a
ball", 560, 560));
    imgArray.push(new CreateImage("wheel.jpg", "This is a
wheel", 560, 560));

    currentIndex = 0;
    updateOutput(imgArray[currentIndex]);
}
```

The code contains a next and prev function navigate through the images. There is global variable **currentIndex** which stores the index of the current image.

## Methods

Methods are functions associated with objects.  Because a function is a data type in JavaScript, you add methods the same way you add any data.

### The this keyword again

To access an object property from within a method, you use the this keyword. The **this** keyword refers to the current object.

Example:

```
var point = { x: 100,
              y:100,
              distance: function(){
                 return Math.pow(this.x,2)+
                        Math.pow(this.y,2);
              }
             }
```

The above object represents a point, with a method called distance that calculates the distance from the origin.