Boomshine ICE A

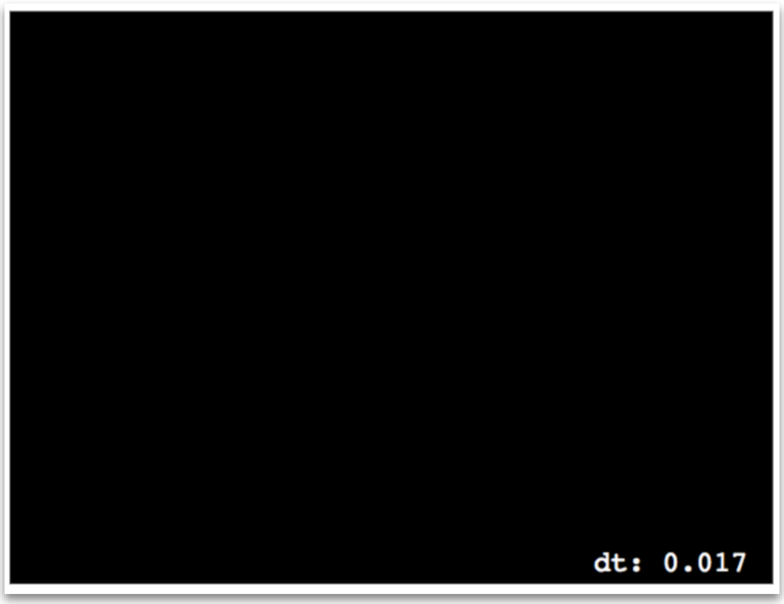
Part I - An Overview of the game and the start files

Over the next few classes we're going to be building a casual game called *Boomshine*.
The *Boomshine* ICE will give us a chance to work with moving sprites, collision detection, sound, procedural and bitmap graphics, particle systems, and more. It will likely be the basis of many of your project 2's.

Boomshine is from the "chain reaction" genre and was originally implemented in Flash - although it now is also available on iOS and Android. You can read about it and play it here:

http://jayisgames.com/archives/2007/03/boomshine.php

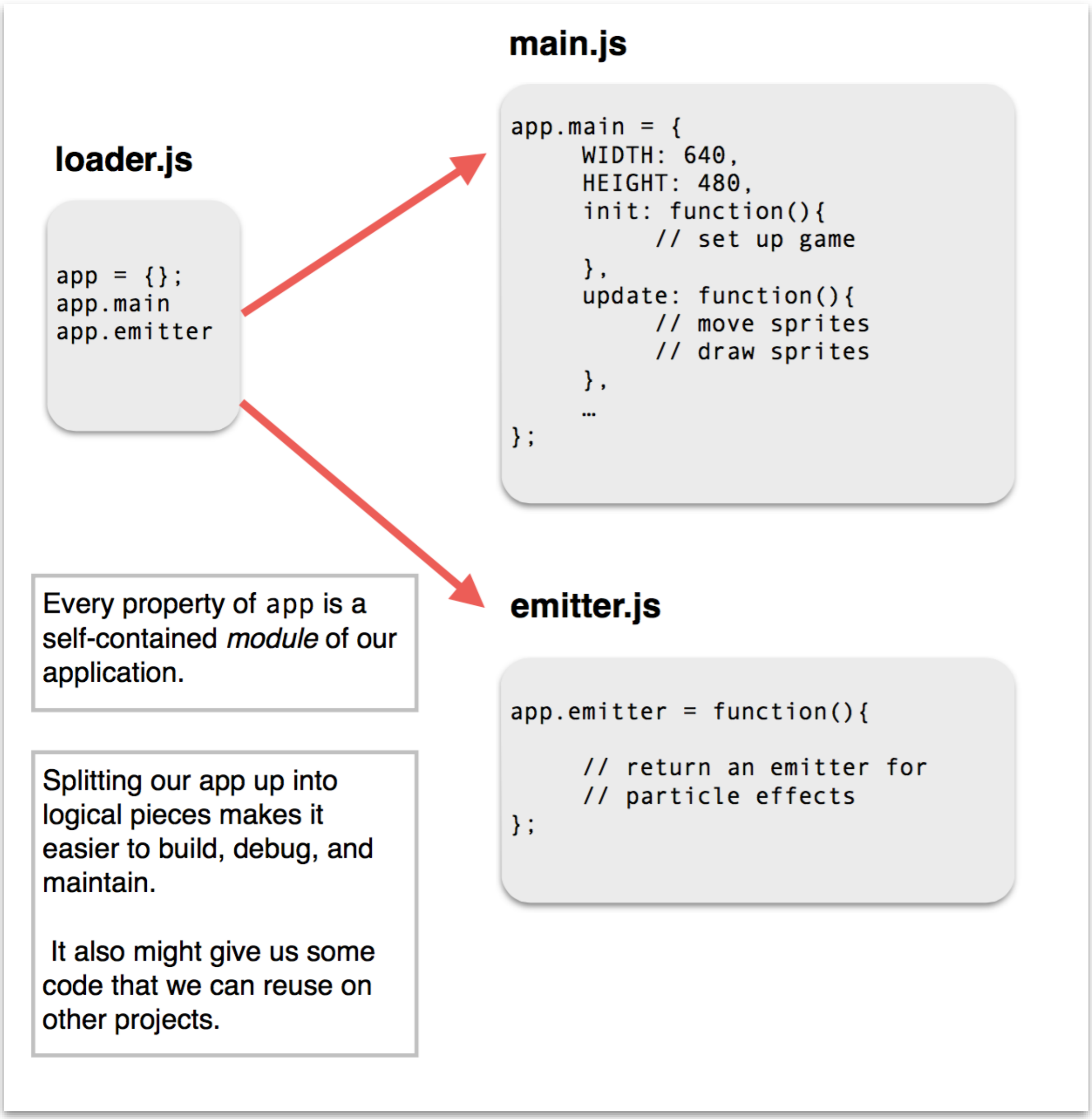
1) Download the start files from my courses and load the HTML file into Chrome. You should see the following:



- We're going to walk through the start files in class and discuss what's going on in them. Going forward, we are going to create distinct *modules* for our game. The first module (and only to start) is `app.main` - this is a *controller object* that will start the game, create circles, track the score, determine the game over conditions and so on.

- Every time we have major functionality to add to the game, we'll create a new module file, and make it a property of `app`. Modules we will be creating soon include **emitter** and **sprite.**

- You can read about the *module pattern* here:
http://addyosmani.com/resources/essentialjsdesignpatterns/book/#modulepatternjavascript



```
app = {};
app.main
app.emitter
```

```
app.main = {
     WIDTH: 640,
     HEIGHT: 480,
     init: function(){
          // set up game
     },
     update: function(){
          // move sprites
          // draw sprites
     },
     …
};
```

```
app.emitter = function(){

     // return an emitter for
     // particle effects
};
```

Every property of app is a self-contained *module* of our application.

Splitting our app up into logical pieces makes it easier to build, debug, and maintain.

It also might give us some code that we can reuse on other projects.

2) Here's a reminder of the key points of the demo:

A) Open the HTML file in a text editor. Note that the 3 *<script>* tags point at external JS files where we're going to be writing code.

```
    <script src='js/utilities.js'></script>
    <script src='js/loader.js'></script>
    <script src='js/main.js'></script>
```

B) Open up **utilities.js** in a text editor. Note that we've got some helper functions we've used before, and we're going to go ahead and keep these in the global scope (because it leads to less cumbersome code later)

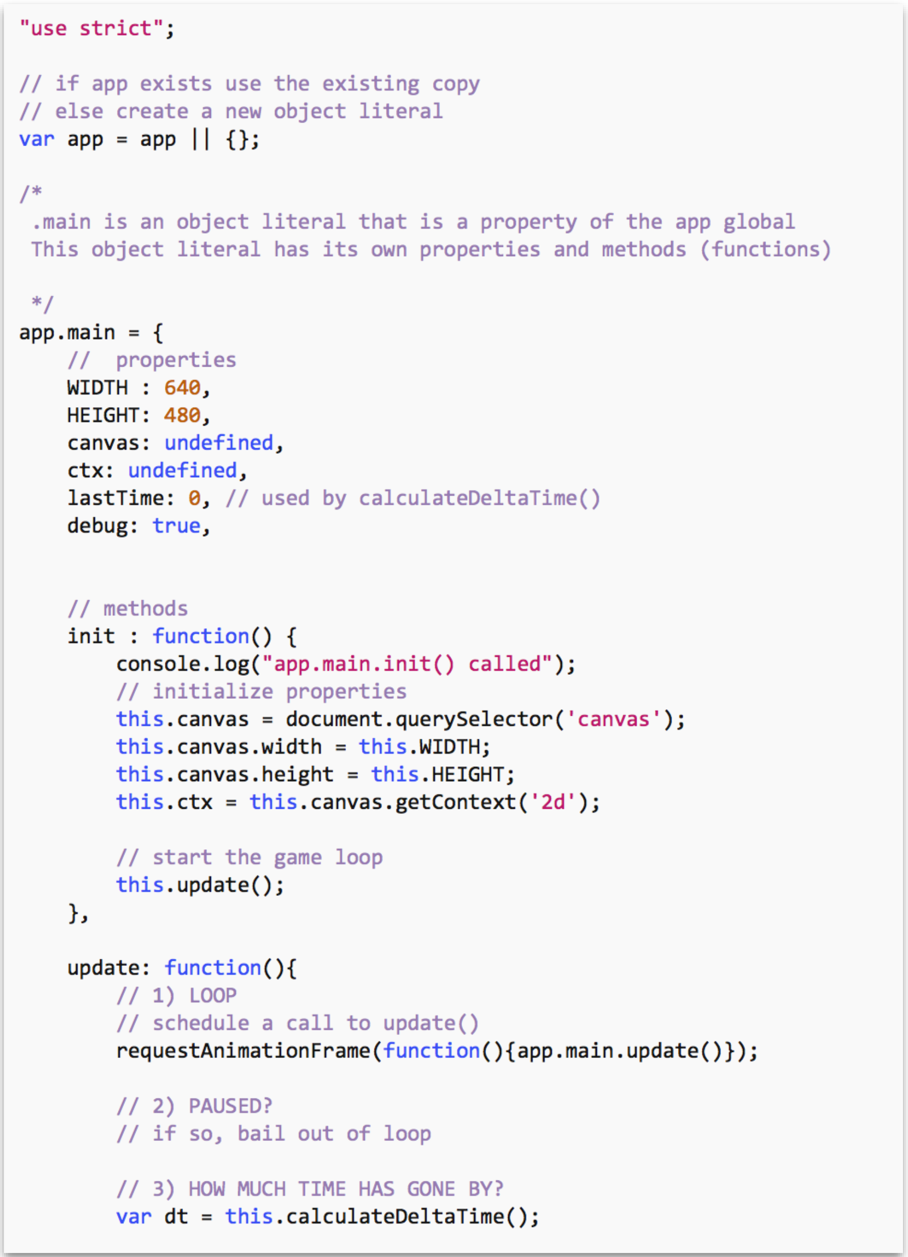
C) Open up **loader.js** in a text editor. Note that we're creating a global variable named *app*. This global variable is an object literal that is going to contain our entire application.

```
/*
loader.js
variable 'app' is in global scope - i.e. a property of window.
app is our single global object literal - all other functions and
properties of
the game will be properties of app.
*/
"use strict";

// if app exists use the existing copy
// else create a new empty object literal
var app = app || {};


window.onload = function(){
	console.log("window.onload called");
	app.main.init();
}
```

D) Open up **main.js** in a text editor. Here’s the first part:



```
"use strict";

// if app exists use the existing copy
// else create a new object literal
var app = app || {};

/*
 .main is an object literal that is a property of the app global
 This object literal has its own properties and methods (functions)

 */
app.main = {
    //  properties
    WIDTH : 640,
    HEIGHT: 480,
    canvas: undefined,
    ctx: undefined,
    lastTime: 0, // used by calculateDeltaTime()
    debug: true,


    // methods
    init : function() {
        console.log("app.main.init() called");
        // initialize properties
        this.canvas = document.querySelector('canvas');
        this.canvas.width = this.WIDTH;
        this.canvas.height = this.HEIGHT;
        this.ctx = this.canvas.getContext('2d');

        // start the game loop
        this.update();
    },

    update: function(){
        // 1) LOOP
        // schedule a call to update()
        requestAnimationFrame(function(){app.main.update()});

        // 2) PAUSED?
        // if so, bail out of loop

        // 3) HOW MUCH TIME HAS GONE BY?
        var dt = this.calculateDeltaTime();
```

i. note how we check to see if the `app` object literal exists yet - if not we make it.
ii. either way, we add a new property named `main` to the `app` object literal.
iii. and `app.main` has its own properties and “methods” like `WIDTH`, `ctx`, `init`, and `update`.
iv. note how the `this` keyword is used here to refer to a property of the current object.
v. note the odd way we are calling `app.main.update()` with `requestAnimationFrame()` - why don’t we just write `this.update()`? Try it and see what goes wrong - `this` gets weird in JS, we’ll talk about that later.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this

Part II - Drawing a Circle

** Download the start files, and rename the folder to *Boomshine-A* **

1) To draw our circle, we'll first initialize some new properties of `.main`. Add the following properties to `app.main`, right after the `debug` property:

```
// init circle properties
x: 100,
y: 100,
radius: 40,
xSpeed: 200, // pixels per second
ySpeed: 160, // pixels per second
fillStyle: "red",
```

2) Go ahead and draw the circle with this code. Add it to the appropriate place in `app.main.update()`:

```
// ii) draw circles
this.ctx.beginPath();
this.ctx.arc(this.x, this.y, this.radius, 0, Math.PI*2, false);
this.ctx.closePath();
this.ctx.fillStyle = this.fillStyle;
this.ctx.fill();
```

Reload the page in a browser and you should see the circle on the screen. If you don't, first check the console for errors, then be sure that you put the above code in the correct places.

Part III - Animate the Circle

1) To move the circle, add the following to the appropriate place in `update()`:

```
// move circles
 this.x += this.xSpeed * dt; // pixels per second * approx. 1/60
 this.y += this.ySpeed * dt; // pixels per second * approx. 1/60
```

Reload the page. The circle will now move across the page and leave off of the right side of the screen.

Part IV - Bouncing

1) To make the circle bounce off of the sides of the canvas, add the following code to `update()`, right after where we moved the circle:

```
// did circle leave screen? Then bounce
if(this.circleHitLeftRight(this.x, this.y, this.radius)){
     this.xSpeed *= -1;
}

if(this.circleHitTopBottom(this.x, this.y, this.radius)){
     this.ySpeed *= -1;
}
```

2) Now write the helper methods of `app.main` we called above. These are new properties of `app.main` (remember that if a property of a JS object points at a function, we usually call it a *method*):

```
circleHitLeftRight: function (x, y, radius){
     if (x < radius || x > this.WIDTH - radius){
          return true;
     }
},

circleHitTopBottom: function (x, y, radius){
     if (y < radius || y > this.HEIGHT - radius){
          return true;
     }
}
```

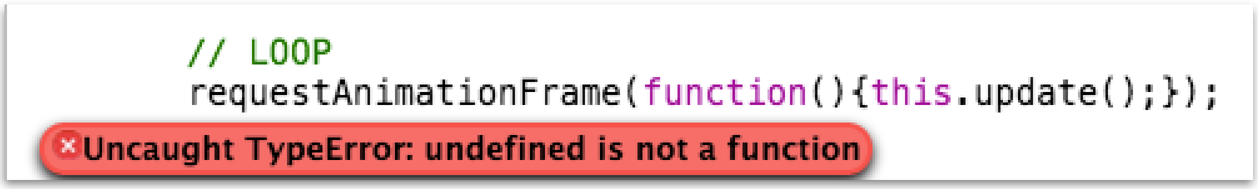
Part IV - Break this app!

1) Change this line:

```
requestAnimationFrame(function(){app.main.update()});
```

to this:

```
requestAnimationFrame(function(){this.update();}); //looks good to me!
```

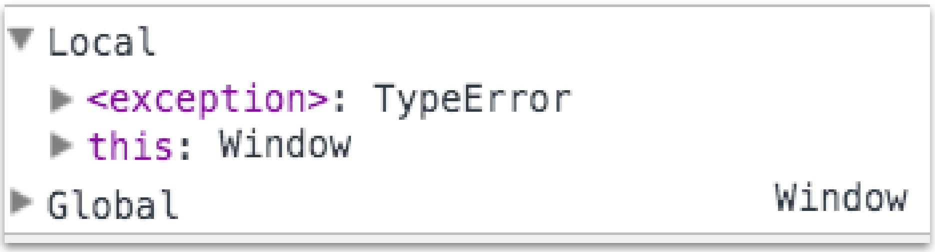


```
// LOOP
requestAnimationFrame(function(){this.update();});
Uncaught TypeError: undefined is not a function
```

2) Run the app. Error in console!

So the problem is in calling *this.update()* and that presumably the *update* function is *undefined*.

A little more investigation in the debugger reveals a hint, that the value of *this* is the global *Window* object:



```
▼ Local
  ▶ <exception>: TypeError
  ▶ this: Window
▶ Global                        Window
```

Therefore the JS runtime was looking for a *update()* function in the *global scope*, not in the *app.main* scope. No such function exists, so we get a console error.

Why was the value of `this` is the global `Window` object, instead of `app.main` like we would expect?

3) The answer: because in JavaScript, the value of *this* is determined by how the function was ***called***. Here are four common cases (there are more, but we'll start here):

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this

A) **In code in the global context (outside of any function), *this* refers to the window object.**

for example:

```
this.a = 37;

console.log(window.a); // 37
```

B) **Inside of a function, the value of this depends on how it was called.** For a simple call (in strict mode) the the value of `this` remains at whatever it's set to when entering the execution context. If it's not defined, it remains *undefined*. For example:

```
function doStuff(){
      // 'this' is undefined
}
doStuff()
```

C) **When a function is used as an event handler, its `this` is set to the element the event fired from.** For example:

```
document.querySelector("#tintRedCheckbox").onchange = function(e){
      tintRed = this.checked; // this is the checkbox that called the method
};
```

D) (This one applies to our issue in Boomshine) - **When a function is called as a method to an object, *this* is set to the object the method was called on.** For example:

```
requestAnimationFrame(function(){this.update();});
```

is equivalent to:

```
window.requestAnimationFrame(function(){this.update();});
```

therefore `this = window`

Because *this* referred to *window*, and *window* does not have an *update* method, we got a JS error.

4) So what's the solution? What we need to do is to preserve the value of *this* to be what we want it to be (where `this` equals `app.main`):

The most elegant solution uses the `bind()` method of Function, which was introduced in ECMAScript5.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/bind

`bind()` will create a new function that permanently binds the value of *this* to whatever the first argument of `bind()` is. Here's our solution - go ahead and use this and get rid of the error:

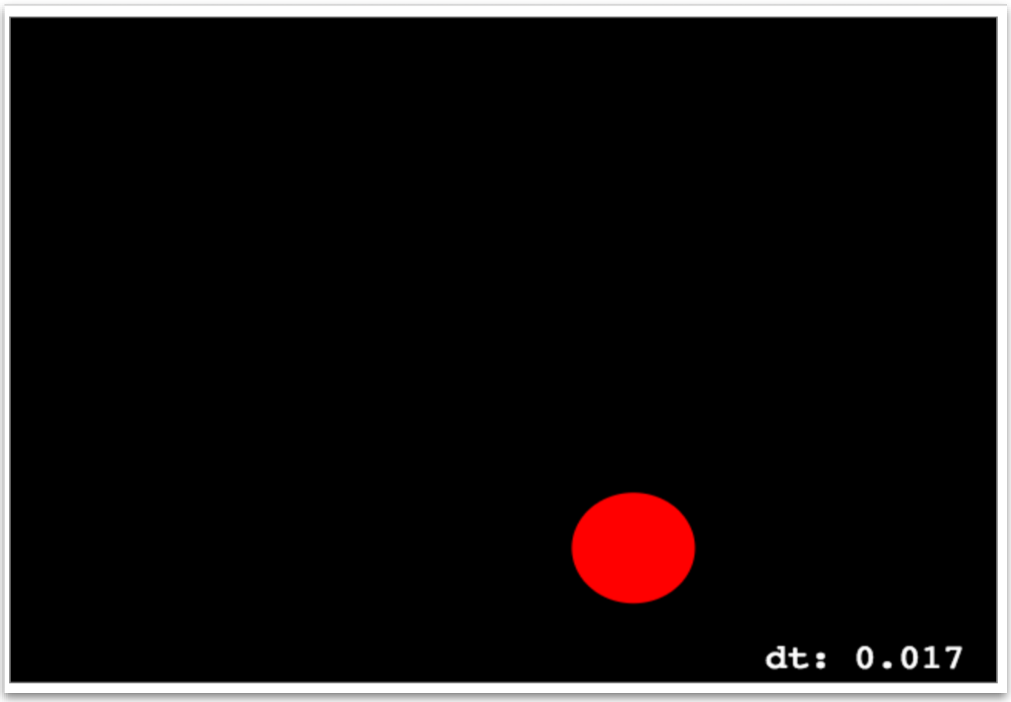
```
requestAnimationFrame(this.update.bind(this));
```

Thus when `update()` is called, the value of `this` will always be `app.main`

PS - The `bind()` method is not using any magic - you can see the JS source code for `bind()` and an explanation here:

http://stackoverflow.com/questions/22103354/javascripts-bind-implementation

5) Think about whether having the individual properties of circle as properties of `app.main` is a good thing. What happens when you have more than 1 circle? Here's a screenshot of the finished version:



```
dt: 0.017
```

Part VI - Finish it up

Boomshine A Rubric

DESCRIPTION	SCORE	VALUE %
Circle Draws – Circle draws correctly to the screen.		**20**
Circle Moves – Circle moves around the screen correctly		**40**
Circle Bounces – Circle correctly bounces off the sides.		**40**
Errors Thrown – Any errors thrown in the console.		**-20% (this time)**
Additional Penalties – These are point deductions for missing submission links, poorly written code, etc. There are no set values for penalties. The more penalties you make the more points you will lose.		
TOTAL		**100%**

ZIP the folder and post it to mycourses. **Include a link to the work on Banjo in the submission comments.**