

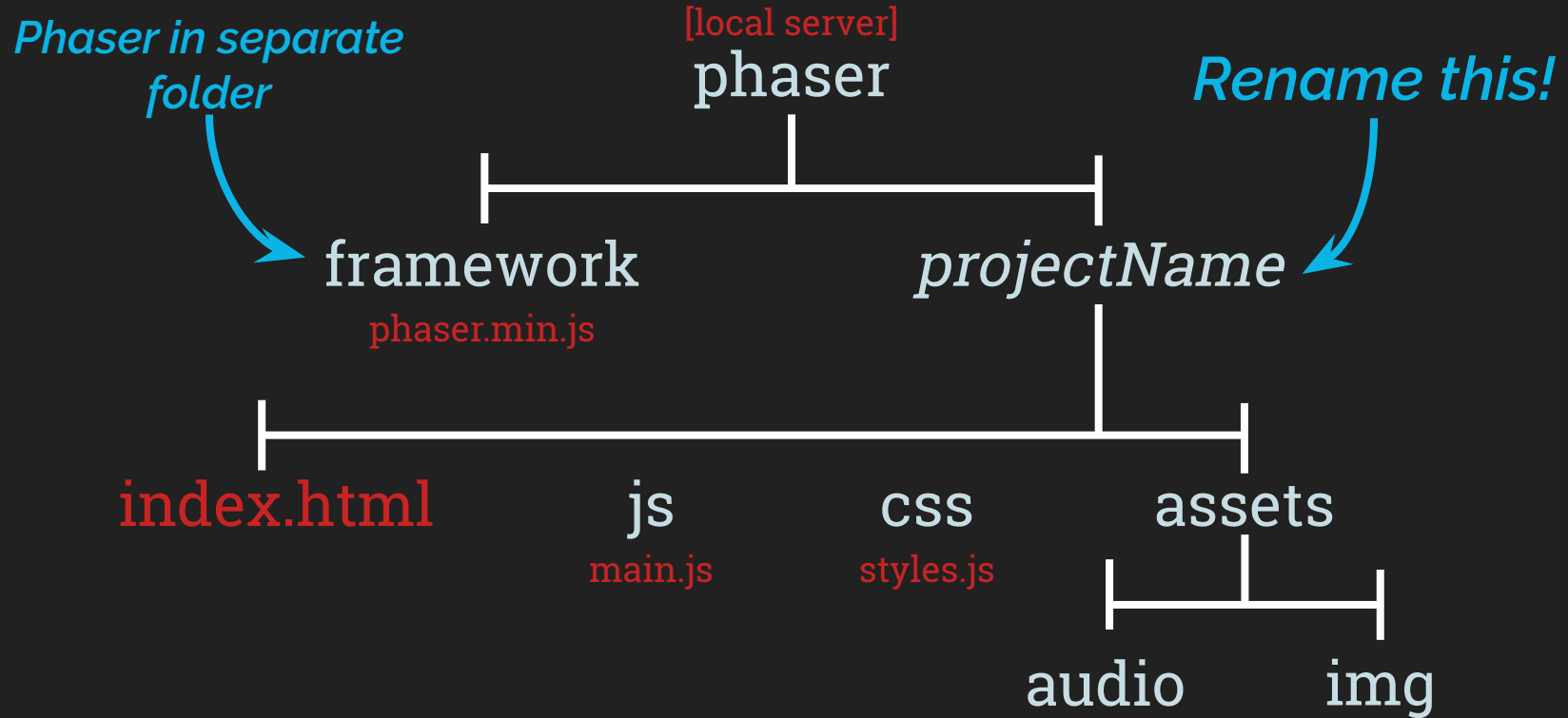
CMPM 120

Prefabs and Prototypes

Reviewing the First Assignment

The best time to turn it in is before the deadline

The second best
time is now



Collaborating and Community



<https://xkcd.com/979/>

"All long help threads should have a sticky globally-editable post at the top saying 'DEAR PEOPLE FROM THE FUTURE: Here's what we've figured out so far ...'"

Please help each other!

Getting help from other people is **good**...

...just **cite** where the code or ideas came from.

...it can also help if you type the code in yourself instead of just pasting it.

This is a good idea because it's important to understand what your code is doing.

Code on Stack Overflow can be **wrong**

Why You Are Here

- Learn the **basic principles** of game programming and **put them into practice**
- Learn how to do the low-level implementation so we can turn **ideas** into **working games**
- Learn how **technology** *and* **teamwork** affect game design (PLO 7 & 8)

Note: Not (directly) testing you on knowing Computer Science concepts! That's what classes like 12B are for!

Document your process

If your workflow involves following a particular set of steps, write that down.

This applies to artists too!

Comments

// Slight change in how comments will be graded going forward

// You're all good at telling me the **how**

// But I also want to know the **why**

// If how something works seems obvious to you, less need for comments -- though it may not be obvious to the rest of your team.

Literate Programming

// The original literate programming paper

<http://www.literateprogramming.com/knuthweb.pdf>

Objects & Prefabs

Learning Objectives

By the end of class you should be able to...

- Paraphrase what a **game prefab** is
- Explain how to use **JavaScript objects** to...
 - ◆ ...extend an existing **prototype**
 - ◆ ...make prefabs and **organize your code** using prototypes

Homework Assignment #2

- Demonstrate **organizing** a game's files
- Implement multiple **game states**
- Practice making your code **modular**

What are some reasons to have more than one file for our games?

- Too much information
- Logical organization
- C++ re-compiles faster
- teamwork!

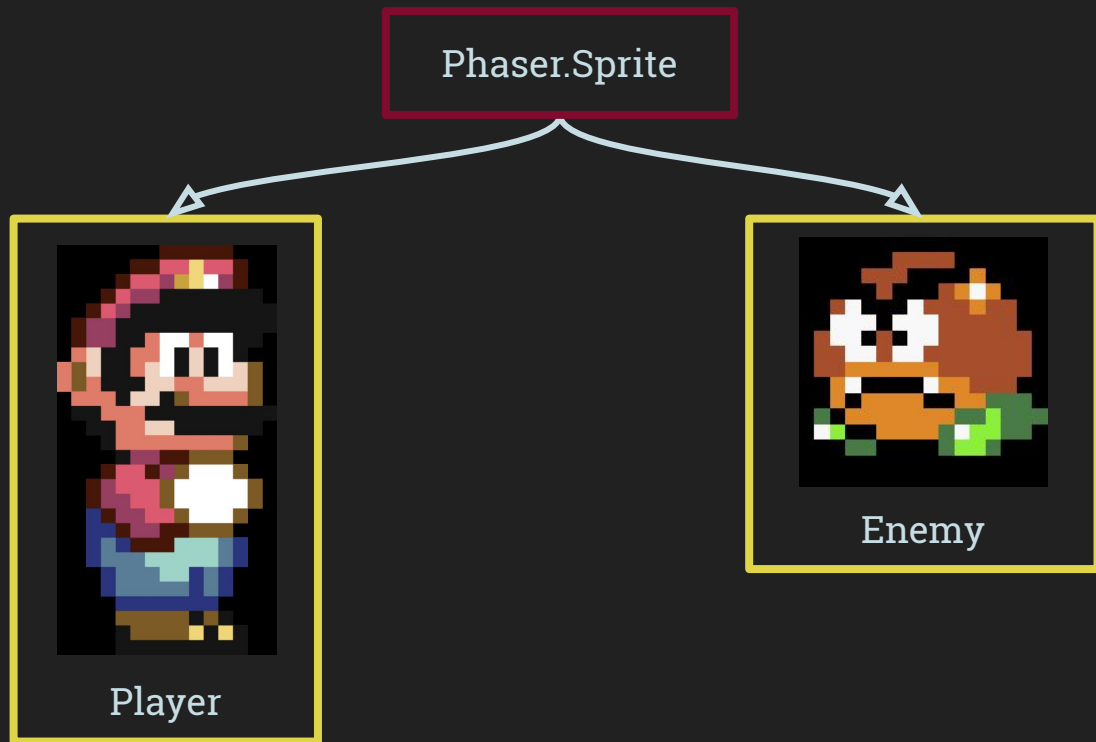
Game Prefabs

Games are complicated systems, we need **organization**.

One way this is done is called a **prefab**.

A *prefabricated object* - same code and data gets used many times

- In Phaser, prefabs are usually:
 - ◆ **in-game elements** that extend `Phaser.Sprite`
 - ◆ or related **user interface elements** that extend `Phaser.Group`.



A **prefab** will add properties and methods that make the **extended Phaser object** unique



Player

this.jumpHeight
this.runSpeed
etc.

Prefab properties and
methods

Phaser.Sprite

Public Properties

alive anchor angle animations autoCull blendMode body bottom
cameraOffset centerX centerY checkWorldBounds children components cropRect
damage data debug deltaX deltaY deltaZ destroyPhase events exists
fixedToCamera frame frameName fresh game heal health height
ignoreChildInput inCamera input inputEnabled inWorld key left lifespan
maxHealth name offsetX offsetY outOfBoundsKill outOfCameraBoundsKill
pendingDestroy physicsType previousPosition previousRotation renderOrderID
right scaleMax scaleMin setHealth shader smoothed texture tint
tintedTexture top transformCallback transformCallbackContext type width
world x y z

Public Methods

addChild addChildAt alignIn alignTo bringToTop contains crop destroy
getBounds getChildAt getChildIndex getLocalBounds kill loadTexture
moveDown moveUp overlap play postUpdate preUpdate removeChild
removeChildAt removeChildren reset setFrame resizeFrame revive
sendToBack setChildIndex setFrame setScaleMinMax setTexture swapChildren
update updateCrop

Inherited properties and methods

If you have experience with other programming languages

JavaScript does
things a bit
differently

functions()

You are probably familiar with grouping code into functions for **organization** and **reuse**.

```
function addFive(parameter) {  
    return parameter + 5;  
}
```

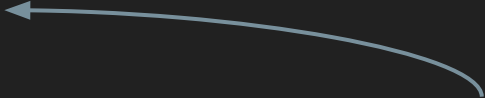
In Javascript, functions are a type of **object**.

```
return addFive;
```

```
1 // Type 1: Function declarations
2 // no parameters or return
3 function useless() {
4     console.log('nihilism');
5 }
6
7 useless();      // :(
8
9 // Note: primitive parameters (like number) are passed *by value*
10 function cube(num) {
11     return num * num * num;
12 }
13
14 cube(10);      // 1000
15 cube(8);       // 512
16 cube('3');     // ???
17 cube('cat');   // NaN
18
19 // Type 2: Function expressions
20 // Really handy for passing a function as an argument to another function
21
22 // anonymous style
23 var cubed = function(num) { return num * num * num; }
24 var x = cubed(4);    // x = 64
25
26 // or named (useful for function recursion)
27 var exp = function power(base, exponent) {
28     console.log('exponent: ' + exponent); // see the recursion happen
29     if(exponent == 0) return 1;
30     else return base * power(base, exponent - 1);
31 }
32 var a = exp(9, 0);   // a = 1
33 var b = exp(8, 3);   // b = 512
```

Functions inside functions

```
function addANumber(a_number) {  
    var adder = function(parameter) {  
        return parameter + a_number;  
    }  
    return adder;  
}
```



By the way, this is called a **closure**.

```
var add_five = addANumber(5);  
add_five(10);
```

Function Scope



```
> var bar = foo;
```

```
✖ ▶ Uncaught ReferenceError: foo is not defined  
   at <anonymous>:1:11
```

Variable bindings are only valid in part of the program.

This region is called the **scope**.

let versus var

```
function exampleFunctionOne() {  
    let first = 7;  
    console.log(first);  
    for(let first = 0; first < 5;  
first++) {  
        console.log(first);  
    }  
    console.log(first);  
}
```

The let statement declares an **enclosing block scope** local variable.

```
function exampleFunctionTwo() {  
    // hoisting: var second;  
    console.log(second);  
    for(var second = 0; second < 5;  
second++) {  
        console.log(second);  
    }  
    console.log(second);  
}
```

The var statement declares a **function scope** variable.

Lexical Scope versus Closures

```
function parent() {  
    var parent_value = 1;  
    function child() {  
        var child_value = 2;  
    }  
}
```

Lexical scope exists in the written code: the `parent_value` **is** accessible in the child function, but the `child_value` **isn't** accessible in the parent function.

```
function makeAdder(x) {  
    return function(y) {  
        return x + y;  
    };  
}  
var add5 = makeAdder(5);  
var add10 = makeAdder(10);  
console.log(add5(2)); // 7  
console.log(add10(2)); // 12
```

Closures use the **run-time context** from when the outer function was called and the inner function was created.

Functions are objects, objects have properties

Therefore, functions can have properties.

You'll remember this later.

```
> var example = function(text) { console.log(text);};  
< undefined  
> example("Hello")  
Hello  
< undefined  
> example.length  
< 1  
> example.toString()  
< "function(text) { console.log(text);}"  
>
```


Arrays

An **array** is an **ordered** set of objects that you can access by **index**.

```
1 // define an array literal (preferred method)
2 var primeNumbers = [2, 3, 5, 7, 11, 13, 17, 19, 23, 29];
3
4 // other valid array definitions
5 var fibonacci = new Array(0, 1, 1, 2, 3, 5, 8, 13, 21, 34);
6
7 var tacoBellSauces = Array('mild', 'hot', 'fire');
8
9 // arrays can hold mixed types (including other objects)
10 var nonsense = [11, 'tacos', 8.88, null, [3, 8], true, {life: 0, death: 1}];
11
12 // an array's first index is [0]
13 var x = primeNumbers[0]; // x = 2
14 var y = tacoBellSauces[3]; // y = undefined
15 var z = nonsense[1]; // z = 'tacos' :p
16
17 // arrays are special objects that have some built-in properties...
18 var len = primeNumbers.length; // len = 10
19
20 // ...and some of those properties are methods
21 primeNumbers.push(31, 37, 41, 43); // add values to the *end* of the array
22 var popped = primeNumbers.pop(); // removes/returns last (popped = 43)
23 var shiftY = primeNumbers.shift(); // removes/returns first (shiftY = 2)
24
25 var fibList = fibonacci.join('|'); // fibList = "0|1|1|2|3|5|8|13|21|34"
26 var revFib = fibonacci.reverse(); // revFib = [34, 21, 13, 8, 5, 3, 2, 1, 1, 0]
27
28 // There are lots of other handy array methods!
29
```

Looping through arrays

```
var ants_of_california = ["argentine ants", "forelius pruinosus", "bicolored pyramid ant", "odorous house ant", "ghost ant", "velvety tree ant"];
```

```
for(let i = 0; i < ants_of_california.length; i++) {  
    console.log(ants_of_california[i]);  
}
```

```
ants_of_california.push("argentine ants");
```

```
for(let ant of ants_in_california) {  
    console.log(ant);  
}
```

Objects

Most things in JavaScript are **objects**

Objects are arbitrary collections of **properties**

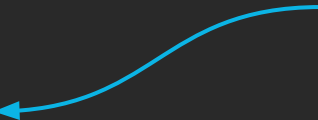
Properties that are bound to functions are called **methods**

We can **access**, **reassign**, and **enumerate** an object's properties

We can use objects to organize things in our game

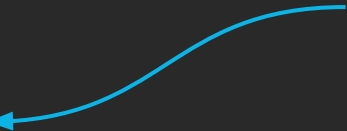
```
1 var game = new Phaser.Game(800, 600, Phaser.AUTO, '', { preload: preload, create: create, update: update });
2
3 // define a creature for our game
4 var creature = {
5   image: "1F42C.png",
6   namekey: "dolphin",
7   velocity: 200,
8   wrap: true,
9   bounce: 1.0
10 };
11
12 // A place to hold the sprite for the creature.
13 // Global for demonstration purposes; you should
14 // organize this better for your game
15 var dolphin;
16
17 // Notice how we're accessing the object's properties
18 // and using them to load an image. The data is no
19 // longer hard-coded in the preload function, it
20 // instead lives elsewhere.
21 function preload() {
22   game.load.path="assets/img/";
23
24   // this is to demonstrate that we can access the properties
25   // of an object with a variable
26   var key_for_creature = "namekey";
27
28   game.load.image(creature[key_for_creature], creature.image);
29 }
30
31 function create() {
32   dolphin = game.add.sprite(50, 50, creature.namekey);
33   game.physics.arcade.enable(dolphin);
34   dolphin.body.velocity.x = creature.velocity;
35   dolphin.body.bounce = creature.bounce;
36 }
37
38 function update() {
39
40   // An example of a ternary operator, which is just a more compressed if-else statement on one line
41   // can you tell me why (800-72) is a bad way to implement this?
42   creature.velocity *= (dolphin.body.x < 0) || (dolphin.body.x > (800-72)) ? -1 : 1;
43
44   dolphin.body.velocity.x = creature.velocity;
45 }
46
47 for (i in creature) {
48   console.log(i);
49 }
50
```

Using the object values instead of writing them explicitly in preload()



```
50
51 // Object properties can have functions
52 var creature = {
53   image: "1F42C.png",
54   namekey: "dolphin",
55   velocity: 200,
56   wrap: true,
57   bounce: 1.0,
58   switch_direction: function() {
59     creature.velocity *= -1;
60     console.log("switched directions");
61   }
62 };
63
64 function update() {
65
66   // An example of a trinary operator, which is just a more compressed if-else statement on one line
67   // can you tell me why (800-72) is a bad way to implement this?
68   if((dolphin.body.x < 0) || (dolphin.body.x > (800-72))) {
69     creature.switch_direction()
70   }
71
72   dolphin.body.velocity.x = creature.velocity;
73 }
```

Properties can
have functions: we
call this a **method**



What if we want multiple, slightly different objects?

```
79 // What if we want multiple objects that are slightly different?
80 var creature1 = {
81   image: "1F42C.png",
82   namekey: "dolphin",
83   velocity: 200,
84   vertical: 50,
85   bounce: 1.0,
86   switch_direction: function() {
87     creature.velocity *= -1;
88     console.log("switched directions");
89   }
90 };
91
92 var creature2 = {
93   image: "1F40D.png",
94   namekey: "snake",
95   velocity: 200,
96   vertical: 250,
97   bounce: 1.0,
98   switch_direction: function() {
99     creature.velocity *= -1;
100     console.log("switched directions");
101   }
102 };
103
104 // This, as you might guess, doesn't scale.
105
```



```

106 // Instead, we can make a constructor function
107 function Creature(key, image, velocity, vertical) {
108     this.image = image;
109     this.namekey = key;
110     this.velocity = velocity;
111     this.vertical = vertical;
112     this.bounce = 1.0;
113     this.switch_direction = function() {
114         this.velocity *= -1;
115         console.log("switched directions");
116     }
117 }
118
119 var dolphin = new Creature('dolphin', "1F42C.png", 200, 50);
120 var snake = new Creature('snake', "1F480.png", 150, 250);
121 var rabbit = new Creature('bunny', "1F407.png", 250, 350);
122 // in fact, let's stick these new objects in an array
123 var game_creatures = [dolphin, snake, rabbit];
124
125
126 function preload() {
127     game.load.path = "assets/img/";
128
129     // this is to demonstrate that we can access the properties
130     // of an object with a variable
131     for (let a_creature of game_creatures) {
132         console.log(a_creature);
133         game.load.image(a_creature.namekey, a_creature.image);
134     }
135 }
136
137 // This is global for demonstration purposes. Organize your code better than this.
138 var creature_sprites = [];
139
140 function create() {
141     // Since we have the creatures in an array, we can create the sprites with a loop
142     for (let a_creature of game_creatures) {
143         var a_creature_sprite = game.add.sprite(50, a_creature.vertical, a_creature.namekey);
144         game.physics.arcade.enable(a_creature_sprite);
145         a_creature_sprite.body.velocity.x = a_creature.velocity;
146         a_creature_sprite.body.bounce = a_creature.bounce;
147         a_creature_sprite["definition"] = a_creature;
148
149         // add to the array of creature sprites so we can access it later
150         creature_sprites.push(a_creature_sprite);
151     }
152 }
153
154 function update() {
155     for (let a_creature_sprite of creature_sprites) {
156         if ((a_creature_sprite.body.x < 0) || (a_creature_sprite.body.x > (800-72))) {
157             a_creature_sprite.definition.switch_direction();
158         }
159         if (a_creature_sprite.body.y > 600) {
160             a_creature_sprite.body.y = 0;
161         }
162         a_creature_sprite.body.velocity.x = a_creature_sprite.definition.velocity;
163         a_creature_sprite.body.acceleration.y = a_creature_sprite.definition.acceleration;
164     }
165 }
166
167 rabbit.move = function() {
168     this["acceleration"] = 5;
169 }
170
171 rabbit.move();

```

Share setup by using a
constructor function!

Note the use
of the **new**
keyword

Where have we seen the
new keyword before?

We can add
methods to
existing objects!

```

73 Phaser.Game = function (width, height, renderer, parent, state, transparent, antialias, physicsConfig) {
74
75     /**
76      * @property {number} id - Phaser Game ID
77      * @readonly
78      */
79     this.id = Phaser.GAMES.push(this) - 1;
80
81     /**
82      * @property {object} config - The Phaser.Game configuration object.
83      */
84     this.config = null;
85
86     /**
87      * @property {object} physicsConfig - The Phaser.Physics.World configuration object.
88      */
89     this.physicsConfig = physicsConfig;
90
91     /**
92      * @property {string|HTMLElement} parent - The Game's DOM parent (or name thereof), if any, as set when the game was created. The actual
93      * @readonly
94      * @default
95      */
96     this.parent = '';
97
98     /**
99      * The current Game Width in pixels.
100     *
101     * _Do not modify this property directly:_ use {@link Phaser.ScaleManager#setGameSize} - e.g. `game.scale.setGameSize(width, height)` -
102     *
103     * @property {integer} width
104     * @readonly
105     * @default
106     */
107     this.width = 800;

```

You've already used
constructor functions!

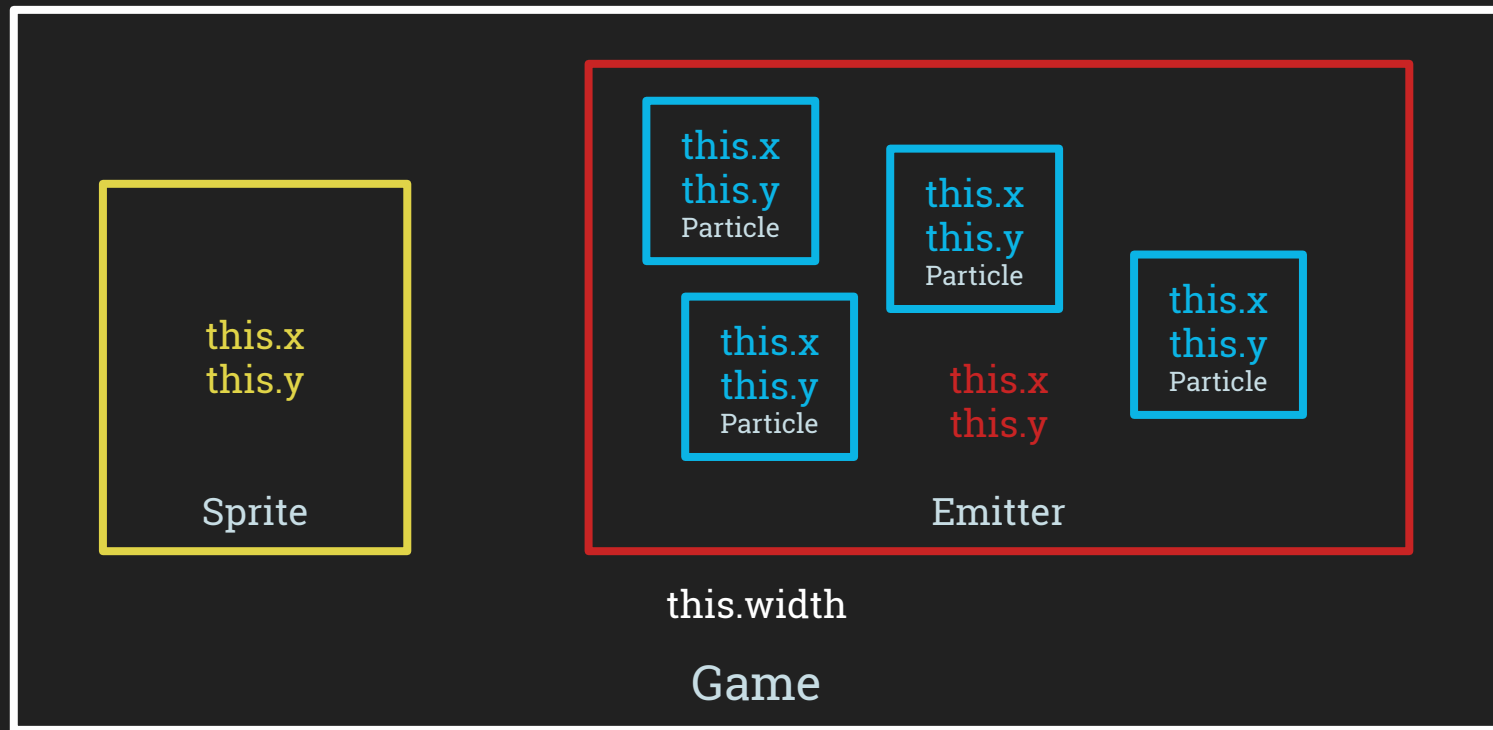
Phaser.Game() is a
constructor function

new & this

Calling a function with the **new** keyword causes it to be treated as a **constructor**.

The constructor will have its **this** variable bound to a fresh object.

this refers to the object the code is inside



Q: Will this one-line program
throw a browser error?

```
this.greeting = "Hello World";
```

Prototypes

Prototypes

You might have noticed something unexplained last week:

In our Phaser states example what is `MainMenu.prototype`?

```
// define MainMenu state and methods
var MainMenu = function(game) {};
MainMenu.prototype = {
  init: function() {
    this.level = 1;
  },
  preload: function() {
    console.log('MainMenu: preload');
  },
  create: function() {
    console.log('MainMenu: create');
    game.stage.backgroundColor = "#facade";
    console.log('level: ' + this.level);
  },
  update: function() {
    // main menu logic
    if(game.input.keyboard.isDown(Phaser.Keyboard.SPACEBAR)) {
      // pass this.level to next state
      // .start(key, clearWorld, clearCache, parameter)
      game.state.start('GamePlay', true, false, this.level);
    }
  }
}
```

“Every JavaScript object has a second JavaScript object associated with it. This second object is known as a **prototype**, and the first object inherits properties from the prototype.”

JavaScript: The Definitive Guide (6E), p. 118

Two prototype concepts

Every JavaScript **object** has a **prototype attribute** that points to its “parent,” i.e., the object from which it inherited its properties. This attribute is normally referred to as the ***prototype object***.

The **prototype object** is a property of each **instance**.

```
Object.getPrototypeOf(my_object);
```

Every JavaScript **function** has a **prototype property** that is empty by default. You implement inheritance by attaching properties and methods to this property.

The **prototype property** is a property of the **constructor**.

```
Object.getPrototypeOf(new  
ConstructMyObject()) ==  
ConstructMyObject.prototype;
```

https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/Object_prototypes

```
hasOwnProperty()  
isPrototypeOf()  
propertyIsEnumerable()  
toString()  
valueOf()
```

Object.prototype



```
var playerSprite = {  
  x: 200,  
  Y: 200,  
  src: "dolphin.png"  
}
```

object literal

Object literals all have the same prototype object

All objects created with { } have

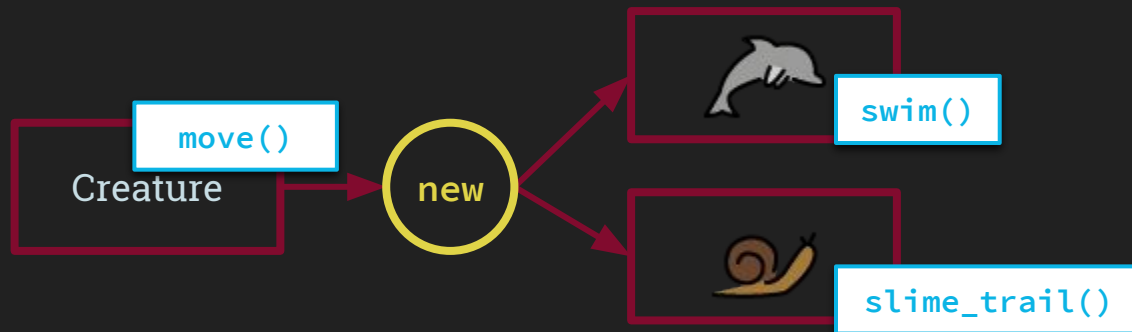
Object.prototype as their **prototype object**

Editing the Constructor

We know how to add new properties to existing objects.

But what if we want to add a shared property to the constructor itself?

```
167 rabbit.move = function() {  
168     this["acceleration"] = 5;  
169 }  
170
```



```

L74
L75 function Creature(key, image, velocity, vertical) {
L76   this.image = image;
L77   this.namekey = key;
L78   this.velocity = velocity;
L79   this.vertical = vertical;
L80   this.bounce = 1.0;
L81 }
L82
L83 Creature.prototype = {
L84   switch_direction: function() {
L85     this.velocity *= -1;
L86     console.log("switched directions");
L87   },
L88   move: function() {
L89     this["acceleration"] = 5;
L90   }
L91 }
L92

```

We add to the
prototype with
.prototype



```
L74
L75 function Creature(key, image, velocity,
L76   this.image = image;
L77   this.namekey = key;
L78   this.velocity = velocity;
L79   this.vertical = vertical;
L80   this.bounce = 1.0;
L81 }
L82
L83 Creature.prototype = {
L84   switch_direction: function() {
L85     this.velocity *= -1;
L86     console.log("switched directions");
L87   },
L88   move: function() {
L89     this["acceleration"] = 5;
L90   }
L91 }
L92
```

You can think of it like
how some cards in
Magic: the Gathering
alter how cards of a
particular type work



Image credit: @roborosewater
<https://twitter.com/RoboRosewater/status/972577767576489984>

.prototype and Phaser

We use prototypes with
our Phaser states.

How does it work?

```
5 // define game
6 var game = new Phaser.Game(800, 600, Phaser.AUTO);
7
8 // define MainMenu state and methods
9 var MainMenu = function(game) {};
10 MainMenu.prototype = {
11     preload: function() {
12         console.log('MainMenu: preload');
13     },
14     create: function() {
15         console.log('MainMenu: create');
16         game.stage.backgroundColor = "#facade";
17     },
18     update: function() {
19         // main menu logic
20         if(game.input.keyboard.isDown(Phaser.Keyboard.SPACEBAR)) {
21             game.state.start('GamePlay');
22         }
23     }
24 }
25
26 // define GamePlay state and methods
27 var GamePlay = function(game) {};
28 GamePlay.prototype = {
29     preload: function() {
30         console.log('GamePlay: preload');
31     },
32     create: function() {
33         console.log('GamePlay: create');
34         game.stage.backgroundColor = "#ccddaa";
35     },
36     update: function() {
37         // game play logic
38     }
39 }
```

Adding States

Tell Phaser we want to add an object that enacts a state, with the key of **'MainMenu'**

```
61
62 // add states to StateManager and start MainMenu
63 game.state.add('MainMenu', MainMenu);
64 game.state.add('GamePlay', GamePlay);
65 game.state.add('GameOver', GameOver);
66 game.state.start('MainMenu');
```

Say that we want to start with the state identified with the key of **'MainMenu'**

Adding States

Phaser looks up the **'MainMenu'** key and sees that it points to a function called **MainMenu()**

```
5 // define game
6 var game = new Phaser.Game(800, 600, Phaser.AUTO);
7
8 // define MainMenu state and methods
9 var MainMenu = function(game) {};
10 MainMenu.prototype = {
11     preload: function() {
12         console.log('MainMenu: preload');
13     },
14     create: function() {
15         console.log('MainMenu: create');
16         game.stage.backgroundColor = "#ffcc00";
17     },
18     update: function() {
19         // main menu logic
20         if(game.input.keyboard.isDown(Phaser.Keyboard.SPACEBAR)) {
21             game.state.start('GamePlay');
22         }
23     }
24 }
25
26 // define GamePlay state and methods
27 var GamePlay = function(game) {}
28 GamePlay.prototype = {
29     preload: function() {
30         console.log('GamePlay: preload');
31     },
32     create: function() {
33         console.log('GamePlay: create');
34         game.stage.backgroundColor = "#ccddaa";
35     },
36     update: function() {
37         // game play logic
38     }
39 }
```

Note that the **game** object has been passed to the function so we can reference it within the state

Adding States

Normally, these would be adding new properties

```

5 // define game
6 var game = new Phaser.Game(800, 600, Phaser.AUTO);
7
8 // define MainMenu state and methods
9 var MainMenu = function(game) {};
10 MainMenu.prototype = {
11   preload: function() {
12     console.log('MainMenu: preload');
13   },
14   create: function() {
15     console.log('MainMenu: create');
16     game.stage.backgroundColor = "#facade";
17   },
18   update: function() {
19     // main menu logic
20     if(game.input.keyboard.isDown(Phaser.Keyboard.SPACEBAR)) {
21       game.state.start('GamePlay');
22     }
23   }
24 }

```

But because we added MainMenu to Phaser's StateManager, Phaser made MainMenu an instance of its State object.

```

61
62 // add states to StateManager and start
63 game.state.add('MainMenu', MainMenu);
64 game.state.add('GamePlay', GamePlay);
65 game.state.add('GameOver', GameOver);
66 game.state.start('MainMenu');

```

Adding States

So now these are
overriding inherited
properties

```

5  // define game
6  var game = new Phaser.Game(800, 600, Phaser.AUTO);
7
8  // define MainMenu state and methods
9  var MainMenu = function(game) {};
10 MainMenu.prototype = {
11   preload: function() {
12     console.log('MainMenu: preload');
13   },
14   create: function() {
15     console.log('MainMenu: create');
16     game.stage.backgroundColor = "#facade";
17   },
18   update: function() {
19     // main menu logic
20     if(game.input.keyboard.isDown(Phaser.Keyboard.SPACEBAR)) {
21       game.state.start('GamePlay');
22     }
23   }
24 }
25
61
62 // add states to StateManager and start
63 game.state.add('MainMenu', MainMenu);
64 game.state.add('GamePlay', GamePlay);
65 game.state.add('GameOver', GameOver);
66 game.state.start('MainMenu');

```

```

// define GamePlay state and methods
var GamePlay = function(game) {};
GamePlay.prototype = {
  preload: function() {
    console.log('GamePlay: preload');
  },
  create: function() {
    console.log('GamePlay: create');
    game.stage.backgroundColor = "#ccddaa";
  },
  update: function() {
    // game play logic
  }
}

```



```

7  /**
8   * This is a base State class which can be extended if you are creating your own game.
9   * It provides quick access to common functions such as the camera, cache, input, match, sound and more.
10  *
11  * ##### Callbacks
12  *
13  * | start | preload | loaded | paused | stop |
14  * |-----|-----|-----|-----|-----|
15  * | init |         |         |         |         |
16  * |         | preload | create | paused |         |
17  * |         | loadUpdate* | update* | pauseUpdate* |         |
18  * |         |         | preRender* |         |         |
19  * |         | loadRender* | render* | render* |         |
20  * |         |         |         | resumed |         |
21  * |         |         |         |         | shutdown |
22  *

```

For **State** methods that you don't override, JavaScript moves up the **prototype chain** until it finds them.

Phaser defines all of the **State** methods you see above, but *none of them have any default behaviors*—it's up to you to provide them. 😊

Gloom

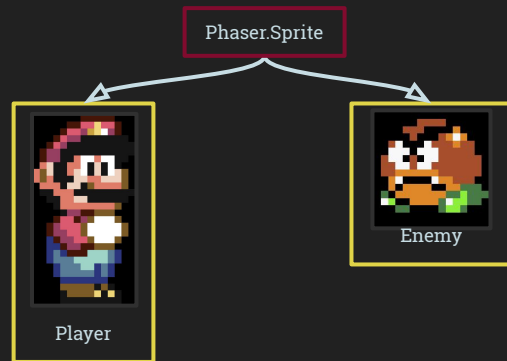
If it helps, you can think of this like the transparent cards in the game Gloom, where cards on top override the cards underneath.



Image credit: Atlas Games
<https://www.atlas-games.com/gloom/>

Constructing a prefab

Step by step



```
function Player(game, key, frame, scale, rotation) {
  // call to Phaser.Sprite // new Sprite(game, x, y, key, frame)
  Phaser.Sprite.call(this, game, game.rnd.integerInRange(64, game.width-64),
    game.rnd.integerInRange(64, game.height-61), key, frame);

  // add properties
  this.anchor.set(0.5);
  this.scale.x = scale;
  this.scale.y = scale;
  this.rotation = rotation;

  game.physics.enable(this);
  this.body.collideWorldBounds = true;
  this.body.angularVelocity = game.rnd.integerInRange(-180, 180);
}

Player.prototype = Object.create(Phaser.Sprite.prototype);
Player.prototype.constructor = Player;

Player.prototype.update = function() {
  if(game.input.keyboard.isDown(Phaser.Keyboard.UP)) {
    this.body.angularVelocity += 5;
  }
  if(game.input.keyboard.isDown(Phaser.Keyboard.DOWN)) {
    this.body.angularVelocity -= 5;
  }
}
```

The prototype should get
its own file



```
function Player(game, key, frame, scale, rotation) {  
    // call to Phaser.Sprite // new Sprite(game, x, y, key, frame)  
    Phaser.Sprite.call(this, game, game.rnd.integerInRange(64, game.width - 64),  
        game.rnd.integerInRange(64, game.height - 61), key, frame);  
  
    // add properties  
    this.anchor.set(0.5);  
    this.scale.x = scale;  
    this.scale.y = scale;  
    this.rotation = rotation;  
  
    game.physics.enable(this);  
    this.body.collideWorldBounds = true;  
    this.body.angularVelocity = game.rnd.integerInRange(-180, 180);  
}
```


```
Player.prototype = Object.create(Phaser.Sprite.prototype);  
Player.prototype.constructor = Player;
```

```
Player.prototype.update = function() {  
    if(game.input.keyboard.isDown(Phaser.Keyboard.UP)) {  
        this.body.angularVelocity += 5;  
    }  
    if(game.input.keyboard.isDown(Phaser.Keyboard.DOWN)) {  
        this.body.angularVelocity -= 5;  
    }  
}
```



The prefab
constructor
function

```
function Player(game, key, frame, scale, rotation) {  
  // call to Phaser.Sprite // new Sprite(game, x, y, key, frame)  
  Phaser.Sprite.call(this, game, game.rnd.integerInRange(64, game.width-64),  
    game.rnd.integerInRange(64, game.height-61), key, frame);  
  
  // add properties  
  this.anchor.set(0.5);  
  this.scale.x = scale;  
  this.scale.y = scale;  
  this.rotation = rotation;  
  
  game.physics.enable(this);  
  this.body.collideWorldBounds = true;  
  this.body.angularVelocity = game.rnd.integerInRange(-180, 180);  
}
```



`.call()`
Call `Phaser.Sprite` as
if it were a method of
this object

```
Player.prototype = Object.create(Phaser.Sprite.prototype);  
Player.prototype.constructor = Player;  
  
Player.prototype.update = function() {  
  if(game.input.keyboard.isDown(Phaser.Keyboard.UP)) {  
    this.body.angularVelocity += 5;  
  }  
  if(game.input.keyboard.isDown(Phaser.Keyboard.DOWN)) {  
    this.body.angularVelocity -= 5;  
  }  
}
```


call()

“JavaScript functions are **objects** and like all JavaScript **objects**, they have **methods**.”

“**call()** allows you to **indirectly** invoke a function as if it were **a method of some other object**. The first argument is the object on which the function is to be invoked; this argument becomes the value of the **this** keyword within the body of the function.”

“Any arguments to **call()** after the first invocation context argument are the values that are passed to the function that is invoked.”

JavaScript: The Definitive Guide, p. 170, 187

```
function Player(game, key, frame, scale, rotation) {
  // call to Phaser.Sprite // new Sprite(game, x, y, key, frame)
  Phaser.Sprite.call(this, game, game.rnd.integerInRange(64, game.width-64),
    game.rnd.integerInRange(64, game.height-61), key, frame);

  // add properties
  this.anchor.set(0.5);
  this.scale.x = scale;
  this.scale.y = scale;
  this.rotation = rotation;

  game.physics.enable(this);
  this.body.collideWorldBounds = true;
  this.body.angularVelocity = 0;
}

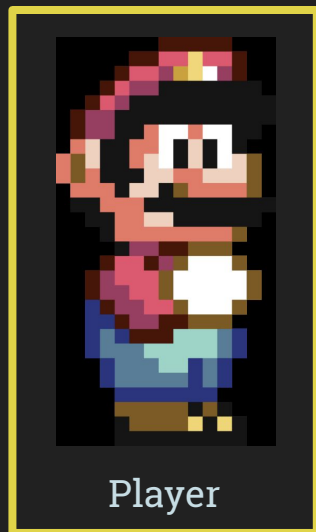
// create a new instance of the Player class
var player = new Player(game, 'player', 0, 1, 0);
```

First argument
this - i.e.
the Player
object

Other arguments
Player.Sprite()
parameters

```
Player.prototype = Object.create(Phaser.Sprite.prototype);
Player.prototype.constructor = Player;
```

```
Player.prototype.update = function() {
  if(game.input.keyboard.isDown(Phaser.Keyboard.UP)) {
    this.body.angularVelocity += 5;
  }
  if(game.input.keyboard.isDown(Phaser.Keyboard.DOWN)) {
    this.body.angularVelocity -= 5;
  }
}
```

First
argument
this - i.e.
the Player
object

Phaser.Sprite

Public Properties

alive anchor angle animations autoCull blendMode body bottom
 bounds children components cropRect
 destroyPhase events exists
 heal health height
 inWorld key left lifespan
 outOfCameraBoundsKill
 previousRotation renderOrderID
 smoothed texture tint
 tintedTexture top transformCallback transformCallbackContext type width
 world x y z

Public Methods

addChild addChildAt alignIn alignTo bringToTop contains crop destroy
 getBounds getChildAt getChildIndex getLocalBounds kill loadTexture
 moveDown moveUp overlap play postUpdate preUpdate removeChild
 removeChildAt removeChildren reset setFrame resizeFrame revive
 sendToBack setChildIndex setFrame setScaleMinMax setTexture swapChildren
 update updateCrop

Inherited properties and methods

this.jumpHeight
 this.runSpeed
 etc.

Prefab properties and
 methods

```
function Player(game, key, frame, scale, rotation) {  
  // call to Phaser.Sprite // new Sprite(game, x, y, key, frame)  
  Phaser.Sprite.call(this, game, game.rnd.integerInRange(64, game.width-64),  
    game.rnd.integerInRange(64, game.height-61), key, frame);
```

```
  // add properties  
  this.anchor.set(0.5);  
  this.scale.x = scale;  
  this.scale.y = scale;  
  this.rotation = rotation;
```

```
  game.physics.enable(this);  
  this.body.collideWorldBounds = true;  
  this.body.angularVelocity = game.rnd.integerInRange(-180, 180);  
}
```

```
Player.prototype = Object.create(Phaser.Sprite.prototype);  
Player.prototype.constructor = Player;
```

```
Player.prototype.update = function() {  
  if(game.input.keyboard.isDown(Phaser.Keyboard.UP)) {  
    this.body.angularVelocity += 5;  
  }  
  if(game.input.keyboard.isDown(Phaser.Keyboard.DOWN)) {  
    this.body.angularVelocity -= 5;  
  }  
}
```

Now we can extend the default Phaser.Sprite by adding our own properties!

Note the use of the `this` keyword to refer to our own object!

```
function Player(game, key, frame, scale, rotation) {  
  // call to Phaser.Sprite // new Sprite(game, x, y, key, frame)  
  Phaser.Sprite.call(this, game, game.rnd.integerInRange(64, game.width-64),  
    game.rnd.integerInRange(64, game.height-61), key, frame);  
  
  // add properties  
  this.anchor.set(0.5);  
  this.scale.x = scale;  
  this.scale.y = scale;  
  this.rotation = rotation;  
  
  game.physics.enable(this);  
  this.body.collideWorldBounds = true;  
  this.body.angularVelocity = game.rnd.integerInRange(-180, 180);  
}
```

```
Player.prototype = Object.create(Phaser.Sprite.prototype);  
Player.prototype.constructor = Player;
```

```
Player.prototype.update = function() {  
  if(game.input.keyboard.isDown(Phaser.Keyboard.UP)) {  
    this.body.angularVelocity += 5;  
  }  
  if(game.input.keyboard.isDown(Phaser.Keyboard.DOWN)) {  
    this.body.angularVelocity -= 5;  
  }  
}
```

Here we explicitly
specify the prefab's
**prototype &
constructor**

```
function Player(game, key, frame, scale, rotation) {  
  // call to Phaser.Sprite // new Sprite(game, x, y, key, frame)  
  Phaser.Sprite.call(this, game, game.rnd.integerInRange(64, game.width-64),  
    game.rnd.integerInRange(64, game.height-61), key, frame);  
  
  // add properties  
  this.anchor.set(0.5);  
  this.scale.x = scale;  
  this.scale.y = scale;  
  this.rotation = rotation;  
  
  game.physics.enable(this);  
  this.body.collideWorldBounds = true;  
  this.body.angularVelocity = game.rnd.integerInRange(-180, 180);  
}
```

```
Player.prototype = Object.create(Phaser.Sprite.prototype);  
Player.prototype.constructor = Player;
```

```
Player.prototype.update = function() {  
  if(game.input.keyboard.isDown(Phaser.Keyboard.UP)) {  
    this.body.angularVelocity += 5;  
  }  
  if(game.input.keyboard.isDown(Phaser.Keyboard.DOWN)) {  
    this.body.angularVelocity -= 5;  
  }  
}
```

And we override the
inherited `update()`
method to add our own
behaviors


```
6 var game;
7
8 // wait for DOM to load before we start up Phaser
9 window.onload = function() {
10     game = new Phaser.Game(500,500, Phaser.AUTO);
11     game.state.add('Play', Play);
12     game.state.start('Play');
13 }
14
15 var Play = function(game) {
16     this.diamond, this.star, this.firstaid;
17 };
18 Play.prototype = {
19     preload: function() {
20         game.load.path = '../assets/img/';
21         game.load.atlas('atlas', 'atlas.png', 'atlas.json');
22     },
23     create: function() {
24         // Player(game, key, frame, scale, rotation)
25         this.diamond = new Player(game, 'atlas', 'diamond', 3, Math.PI);
26         this.star = new Player(game, 'atlas', 'star', 2, 0);
27         this.firstaid = new Player(game, 'atlas', 'firstaid', 0.5, Math.PI/2);
28         game.add.existing(this.diamond);
29         game.add.existing(this.star);
30         game.add.existing(this.firstaid);
31     },
32     update: function() {
33         // note how this is empty b/c our objects update in the prefab!
34     },
35     render: function() {
36         game.debug.text('Press up/down to change angular velocity', 20, 20, 'white');
37     }
38 };
```

Back in main.js, we use
our Prefab constructor
to create three new
Player objects

[main.js]

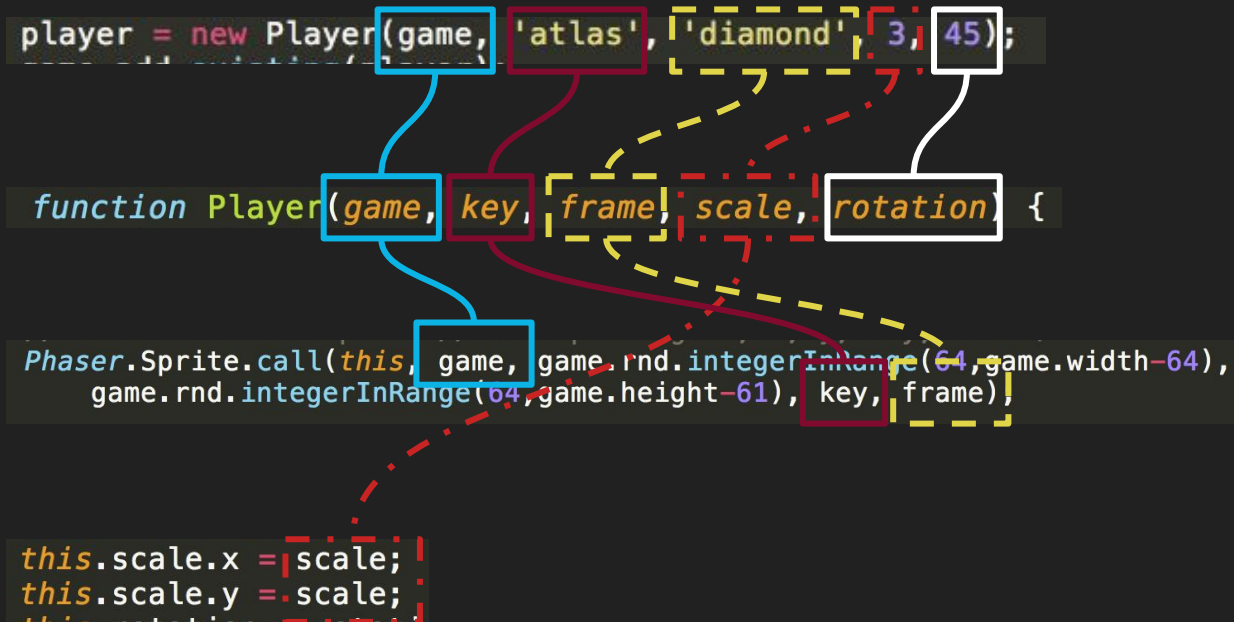
```

6 var game;
7
8 // wait for DOM to load before we start up Phaser
9 window.onload = function() {
10     game = new Phaser.Game(500,500, Phaser.AUTO);
11     game.state.add('Play', Play);
12     game.state.start('Play');
13 }
14
15 var Play = function(game) {
16     this.diamond, this.star, this.firstaid;
17 };
18 Play.prototype = {
19     preload: function() {
20         game.load.path = '../assets/img/';
21         game.load.atlas('atlas', 'atlas.png', 'atlas.json');
22     },
23     create: function() {
24         // Player(game, key, frame, scale, rotation)
25         this.diamond = new Player(game, 'atlas', 'diamond', 3, Math.PI);
26         this.star = new Player(game, 'atlas', 'star', 2, 0);
27         this.firstaid = new Player(game, 'atlas', 'firstaid', 0.5, Math.PI/2);
28         game.add.existing(this.diamond);
29         game.add.existing(this.star);
30         game.add.existing(this.firstaid);
31     },
32     update: function() {
33         // note how this is empty b/c our objects update in the prefab!
34     },
35     render: function() {
36         game.debug.text('Press up/down to change angular velocity', 20, 20, 'white');
37     }
38 };

```

Note that we have to manually add our prefab objects to Phaser's display list

How the parameters flow

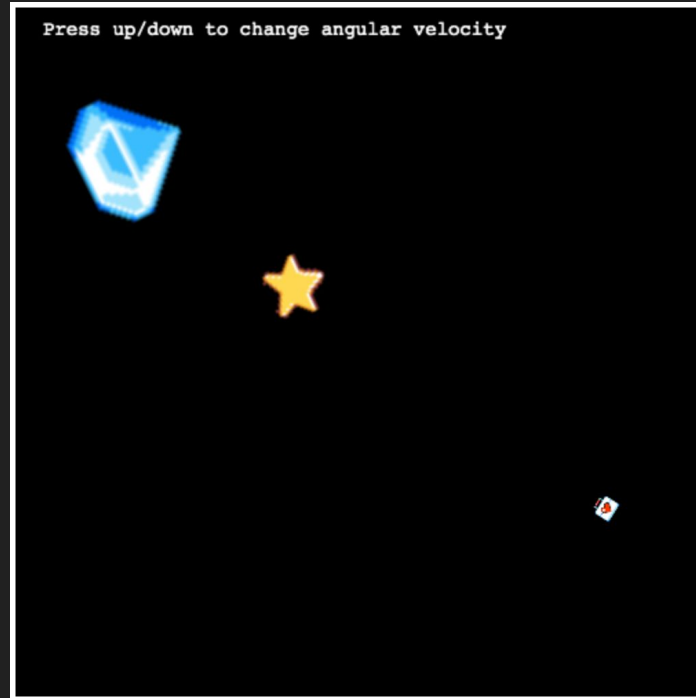


```
<!DOCTYPE html>
<html>
<head>
  <title>Phaser</title>
  <script type="text/javascript" src="../framework/phaser.min.js"></script>
  <script type="text/javascript" src="main.js"></script>
  <script type="text/javascript" src="Player.js"></script>
</head>
<body>
  <h1>I Make Bad Games, LLC.</h1>
</body>
</html>
```

Need to include the
<script> file in
index.html

The order is important!

Note which
file this is

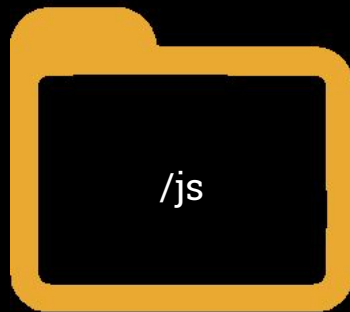


inheritance02.js / Player.js

Project organization

Keeps things manageable

Helps with cooperation



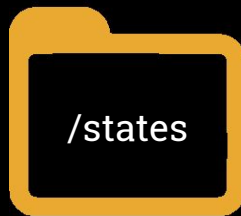
`main.js`



`player.js`



`enemy.js`



`load.js`



`play.js`

Programming Homework #2

Snowy States

Snowy States

- Organization
 - ◆ Comments
 - ◆ File Structure
- States and Conditions
 - ◆ Your game should have three states: MainMenu, Play, and GameOver
 - ◆ Use the state object's .prototype
 - ◆ Add text and additional behaviors, as described in the assignment
- Prefabs
 - ◆ Construct a Snowstorm prefab in a separate file
 - ◆ Add 100 snowflake objects to the scene
 - ◆ Override the prefab's update method to allow the player to reverse all of the snowflakes

