Phaser.js is a *game framework* built with JavaScript that you can use to make video games on the web! Phaser gives us a set of utilities that we can use so that we don't have to start from scratch when we want to make a game.

<script src="https://cdn.jsdelivr.net/npm/phaser@3.16.2/dist/phaser.min.js"></script>

Draw A Circle

In this exercise, we're going to cover how to make a simple geometric shape in Phaser. A Circle! To do this, we're going to use the method `this.add.circle()`. This creates a `GameObject` that we can use to represent a circle in the game. `GameObject`s can have different animations or interactions that we can add to them, but for now, we're going to focus on drawing them. To make a circle we provide `this.add.circle()` with four arguments in the following order:

- The `x` coordinate for the center of the circle. A larger `x` value means the circle will be further to the right in the game.
- The `y` coordinate for the center of the circle. A larger `y` value means the circle will appear further down in the game.
- The `radius`, a value for how large the circle should be.
- The `fillColor`, a number representing the color for the circle. We will be providing hexadecimal values for all colors in this lesson. Feel free to use an online color picker when choosing yours, most will supply a hexadecimal value. Hexadecimal colors in JavaScript look like `0xFF8030` where `0x` indicates the value is hexadecimal and `FF8030` is the color code for the color we want.

For example:

```
this.add.circle(20, 50, 10, 0xFF0000);
```

This line of code would create a new circle that is `20` pixels to the right and `50` pixels down from the top-left corner (sometimes called "the origin"). This circle will be `10` pixels in radius (i.e., `20` pixels across), and bright red due to the color we supplied `0xFF0000`.

Draw A Sprite

Phaser gives us very similar tools to create a `GameObject` for sprites that we provide. A *sprite* is an image that is intended to represent a character, enemy, or some other object in a game. We use image files for a lot of different things in games, including backgrounds and icons, but sprites represent a person or item inside the game itself.

In order to add a sprite we can call `this.add.sprite()`. However, we need to load in the image we want to use as a sprite in an earlier step. `preload()` is the name of the function that happens before `create()`. Where `create()` is about setting up all the game objects to set the stage for our game, `preload()` is about loading in all *asset* files that we'll be using. "Assets" could refer to lots of things but, for now, our assets will consist of sprites, background images, in-game sounds, and background music.

So the two steps to creating a sprite in your game are: Loading the image in `preload()` with `this.load.image()` which takes two arguments: a `key` that you'll use to refer to it and the `path` to the image itself. The `path` can be a local file or the URL for a resource hosted elsewhere on the web.

Creating the GameObject in `create()` with `this.add.sprite()` which requires three arguments: the `x` value, the `y` value, and the `key` used when you loaded the image.

```
function preload() {
  this.load.image('dragon', 'assets/sprites/dragon.png');
}

function create() {
  this.add.sprite(40, 80, 'dragon');
}
```

In `preload()` we loaded an image into our game and gave it a `key` value of `'dragon'`. The location of the image we loaded was `assets/sprites/dragon.png`.

Then we created the GameObject itself in `create()`. We set it 40 pixels to the right and 80 pixels down and informed Phaser to use the image we loaded in earlier under the `key` "dragon". It's important to note that our `x` and `y` values of 40 and 80 indicate the *center* of the sprite. So calling `this.add.image(0, 0, 'dragon');` would only display the bottom right quarter of the image.

Draw A Background Image

Now that we've covered drawing a sprite, let's talk about drawing a background image. A background image is not a sprite — usually the player will not interact with the background at all. Because of this, we don't need a `sprite` object for our background images. We still need to load in the image in our `preload()` function but we can use `this.add.image()` instead of `this.add.sprite()`:

```
function preload() {
  this.load.image('bg', 'assets/images/bg.png');
}

function create() {
  this.add.image(200, 200, 'bg');
}
```

Above, we load in our background image from the path `assets/images/bg.png` and use the key `bg` to refer to it. In `create()` we add the image, centered at (200, 200) — 200 pixels to the right of the left edge and 200 pixels down from the top edge.

Create A Config

We've been making GameObjects in an existing game, but let's take a step back and think about how to create a Phaser game from the beginning. We can do that by calling `new Phaser.Game()`. `Phaser.Game()` by itself creates a new `<canvas>` element and appends it to your HTML document. This element allows Phaser to draw sprites and shapes to a website.

In order to make games of our own design, we call `Phaser.Game()` with an object that specifies some meta-information about the game. We define that object beforehand and conventionally name the variable `config`. Here are some of the keys we can define in a `config`:

- `height`: the height, in pixels, of the `<canvas>` Phaser creates.
- `width`: the width, in pixels, of the `<canvas>` Phaser creates.
- `parent`: the HTML element that Phaser will place the `<canvas>` element inside of.
- `backgroundColor`: the background color of our game, usually given in hexadecimal.
- `type`: the Renderer Phaser will use: either `Phaser.WEBGL` which offers some additional features while drawing or `Phaser.CANVAS` which can run on more browsers. The default, `Phaser.AUTO`, checks if the browser supports WebGL and, if not, switches to canvas.

There is plenty more data that we could possibly add to configure our games, but these basics will get us pretty far. If you wish to learn more about the fields a `config` supports, check out the Phaser documentation (the docs!).

Start Making A Scene

It's nice to have our games made to be a specific size with a specific background color, but where do we define game logic? Phaser games are composed of *Scenes* that we define and pass to Phaser in the config! A Phaser Scene can have any of the following functions:

`preload()`, where we load in external files (or "assets") to our game. `create()`, where we define the GameObjects that are necessary at the start of our game. `update()` where we define animation and interaction in our game (more on this in later exercises!)

When defining a Scene, it's only necessary to define the functions we need. So we don't need to define a `preload()` function for a game that doesn't use external assets, for instance.

After these functions are defined, they're passed in as an object to the Phaser config! Here's how that looks altogether:

```
function preload() {
  this.load.image('codey', 'assets/images/codey.png');
}

function create() {
  this.add.image('codey', 130, 100);
}

const config = {
  width: 300px,
  height: 600px,

  scene: {
    preload,
    create
  }
};
```

Above, we created `preload()` and `create()` methods for our Scene. The `this` object which we refer to so often in these methods refers to the Scene itself! Relatedly, `this.load` and `this.add` are helper libraries the Scene provides with methods (like `this.load.image()` and `this.add.circle()`) that allow us to load in files and create game objects respectively.

In our `config` we added in a new key, `scene`, and pass an object to it. The object maps our `preload()` function to the scene and our `create()` function to the scene. Note that we are using JavaScript's property-value shorthand, the code above would be the same if we passed `{ preload: preload, create: create }` to the `scene` instead.

Move Your Bodies

Now we're going to add motion to our Phaser game! You might recall that we can add an `update()` function to our Scene that adds motion, animation, and interaction! We're going to start with something very simple: moving an object across the screen. If we create a GameObject centered at (0, 100), that is, if we create a GameObject with its `x` value equal to 0 and its `y` value equal to 100, we can move it across the screen in the `update()` function like so:

```
let circle;

function create() {
  // create a red circle with radius 20 at (0, 100)
  circle = this.add.circle(0, 100, 20, 0xff0000);
}

function update() {
  // move that circle to the right
  circle.x += 1;
}
```

First, we define a global variable called `circle` because we're going to be referring to the same object in two different functions. There are a few ways to do this, but for small games, it's OK to make a few global variables.

Afterwards, in `create()` we assign a new GameObject to `circle` that we instantiate using `this.add.circle()` with `x`, `y`, `radius`, and `fillColor` arguments.

Then in `update()` we update our `circle`‘s `x` value. `update()` is called every frame the game is in view.

`update()` has two optional parameters: `time` and `delta`. The first is the number of milliseconds that your Phaser game has been running, the second is the difference (in milliseconds) since the last time `update()` was called. Phaser does its best to call `update()` 60 times per second, so `delta`‘s values will normally be around `16.6`. This is good enough for us, so we’ll frequently define `update()` without parameters.

Storing State

Because we handle all of our Scene information in different functions, it becomes important to pass information between our Scene functions. Something we created in `create()` will need to be updated in `update()`, for instance. There are a few ways we can accomplish this:

- Create global variables for everything.
- Attach important variables to the Scene itself by creating a new property for `this` from within a Scene method.
- Create a `gameState` object and keep track of the state there.

Each solution is a valid technique, but we’re going to focus on creating a `gameState` object and manipulating that from now on. Here’s one way that we’ll apply this solution:

```
const gameState = {};

function create() {
  gameState.circle = this.add.circle(20, 20, 5, 0xff0000);
  gameState.rectangle = this.add.rectangle(40, 60, 90, 10, 0x00ff00);
}

function update() {
  gameState.circle.x += 1;
  gameState.rectangle.y += 1;
}
```

In the above code, we started by defining a `gameState` as a new JavaScript object. In `create()` we create a circle and a rectangle and assign them to properties of our `gameState` object. Then in `update()` we can refer to `gameState.circle` and `gameState.rectangle` to manipulate them!

Input

It’s time to make a game that we can actually play! Phaser gives us ways to handle mouse and keyboard input, so that we can trigger events to happen in the game. In this exercise we’re going to look at ways to use the mouse, but we’ll be covering keyboard input in our next exercise.

In order to interact with a GameObject, we need to call the `setInteractive()` method on it. The `setInteractive()` method tells Phaser to listen in for interactive events on the GameObject.

After calling `setInteractive()` we can provide the GameObject with an *event handler*. The event handler is a function that gets called when a specific interaction has happened to the GameObject. We're going to look at four different possible events in this exercise:

- `'pointerdown'`: this event gets called when the mouse button has been pressed (but not released) on the GameObject.
- `'pointerup'`: this event gets called when the mouse button has been released over a GameObject.
- `'pointerover'`: this event gets called when the mouse pointer hovers over the GameObject.
- `'pointerout'`: this event gets called when a mouse pointer that was hovering over a GameObject is moved somewhere else.

It's interesting to consider `'pointerdown'` and `'pointerup'` are completely separate events. "A mouse click is a mouse click, you can't say it's only half" one might say. Not to meander too far into the dark forests of user experience and game *design*, let's just clarify that pressing down the mouse is done much more haphazardly than releasing the mouse. A player may realize they had not intended to click somewhere, drag their cursor away to release. If we capture the `'pointerup'` event, we'll only do things after the mouse click has been finished. If we only use the `'pointerdown'` event, we'll ignore a player who realized they have misclicked.

Now let's say we wanted to make our famous `circle` GameObject change colors at the click of a mouse. We would use the `'pointerup'` event for that, because that means the mouse has been clicked and released. We would add an event listener to change the color on the `'pointerup'` event. Here's how that would look:

```
const gameState = {}

function create() {
  gameState.circle = this.add.circle(50, 50, 20, 0xFF0000);
  gameState.circle.setInteractive();
  gameState.circle.on('pointerup', function() {
    this.fillColor = 0x00FF00;
  });
}

const config = {
  scene: { create }
}

const game = new Phaser.Game(config)
```

The above code creates a red circle in a Scene's `create()` method, then makes it interactive by calling `setInteractive()` on it. Now that it's interactive, we can add an event listener for the `'pointerup'` event. This is done by calling the `.on()` method on the GameObject.

The `.on()` method takes two arguments: the name of the event (`'pointerup'`) and the callback function. We create a new function as the second argument, with one line inside of it: reassigning `this.fillColor`. Updating `this.fillColor` changes the color of the circle (that's what `this` is here).

Notice that we have a change in our game's state without defining an `update()` function. Event listeners can be defined in the `create()` function, because they're part of the definition of the GameObject and the setup of the Game.

Keyboard Events

Only some games exclusively use the mouse to play. Plenty of browser games, especially those with more complex gameplay, require keyboard input. Before, we assigned mouse events to specific GameObjects. This helped us tell if a mouse cursor was hovering over, or clicking on, an object in our game. Where mouse clicks take place at a specific point in our game with `x` and `y` coordinates, a keyboard offers a different interface without that positional information. So our keyboard handlers will handle any keypress that happens while our game is in focus — for this reason they won't be registered as events to particular GameObjects in our game.

The first way of adding a keypress handler is by calling `this.input.keyboard.on()`. This takes two arguments: first, the name of the event — something like `"keyboard-A"` for the `A` keypress. Next, the function to be called when handling the keypress. We can pass a function we've defined elsewhere, but unless you're duplicating functionality (say to avoid replicating the same code for a keyboard and a gamepad) it's fine to define an anonymous function. Here's how that would look:

```
const gameState = {};

function create() {
  gameState.circle = this.add.circle(30, 30, 10, 0xFF0000);
  this.input.keyboard.on('keyboard-W', function() {
    gameState.circle.fillColor = 0xFFFF00;
  });
}
```

This code creates a red circle and then, when a `W` is pressed on the keyboard, the circle turns yellow.

---

createCursorKeys()

Another way of adding a keyboard event listener is by using a shortcut that Phaser offers, `createCursorKeys()`. This creates an object that maps the names of some usual cursor keys (`UP`, `DOWN`, `LEFT`, `RIGHT`, `SHIFT`, and `SPACE`) to a cursor object that we can use to detect when they've been pressed. We can save those as a property in our `gameState` object and then check if they're pressed within our `update()` function.

```
const gameState = {};

function create() {
  gameState.circle = this.add.circle(50, 50, 20, 0xFF0000);
  gameState.cursors = this.input.keyboard.createCursorKeys();
}

function update() {
  if (gameState.cursors.left.isDown) {
    // move the circle left!
    gameState.circle.x -= 3;
  }
}

const config = { scene: { create, update }};
const game = new Phaser.Game(config);
```

Above, we created a `cursors` property for our `gameState` and assigned the result of `.createCursorKeys()` to it. In our `update()` method we checked if the left key is being pressed by checking if `gameState.cursors.left.isDown` is truthy. If the left button is pressed, we move the circle to the left.

Sounds

Games aren't just idle toys. Many games cultivate experiences, blurring the distinction between a momentary diversion and artwork that you can interact with. To this end, there's few things that can prop up the player's immersion like good music and sound design. Since sounds are assets (like sprites and images), we load them in during the `preload()` function. First let's handle playing music for a scene, here's how we'd do that:

```
const gameState = {};

function preload() {
  this.load.audio('theme', 'assets/music/theme.wav');
}

function create() {
  gameState.music = this.sound.add('theme');
  gameState.music.play();
}

const config = { scene: { preload, create }};
const game = new Phaser.Game(config);
```

First we load in the our theme song asset in `preload()`. We provide `this.load.audio()` with the key `'theme'`, similar to how we give keys to our visual assets. After the key, we also need to provide `this.load.audio()` with the path to the asset. Then we put the loaded asset in our scene in our `create()` function using `this.sound.add()`. This creates a sound object that we can `.play()`. We save that sound object as the value for `gameState.music` and immediately call the `.play()` method.

We might also want a sound to play whenever a specific action takes place. Let's say we want a little beep to play when the mouse has been clicked. Here's how we'd do that:

```
const gameState = {}

function preload() {
  this.load.audio('chime', 'assets/sounds/chime.wav);
}

function create() {
  gameState.circle = this.add.circle(100, 100, 30, 0xFF0000);
  gameState.circle.setInteractive();

  gameState.chimeSound = this.sound.add('chime');

  gameState.circle.on('pointerup', function() {
    gameState.chimeSound.play();
  });
}

const config = { scene: { preload, create }};
const game = new Phaser.Game(config);
```

Above, we load in the sound located at `assets/sounds/chime.wav` and save that sound with the key `'chime'`. We then create a circle in our Scene and set it to be interactive. Then we add our `'chime'` sound to our Scene. We then add in a new event listener so that when our circle gets clicked, we play our `chime` sound.

Coloring

We've been given the basic layout of our Game — it's up to us, the game developer, to provide the part of our game that makes it playable. We need to create the *mechanics* of the game, the part that makes it more than a picture of a pegasus.

Updating Color

Now that we can change the color of our pegasus, let's get to work on making that palette selector functional. When we click on a color, we want that to become the selected color.

We need to use a third argument to our event handler method `.on()`, where we give extra context to be used within our event handler. We want each circle in our color palette, `paletteCircle`, to be aware of the color that it indicates so that it can update `gameState.selectedColor`. The syntax for our event context is this:

```
// important number we want to use
const importantNumber = 10;

// pointer event handler where we want
// to use our important number
gameObject.on('pointerup', function() {
  this.gameObject.x = this.importantNumber
}, { importantNumber, gameObject });
```

Above we created `importantNumber` to be some important piece of data that we wanted to use inside our event handler. We call `gameObject.on('pointerup')` to create an event handler for the `gameObject` click event.

Inside the event handler, `this` refers to our `gameObject` unless the third argument is present. If there is a third argument, `this` refers to the object passed in that third argument, the *context*. When our third parameter is `{ importantNumber, gameObject }` we can refer to `this.importantNumber` inside our handler.

Which is what we need! In the code above we update the `x` value of `gameObject` to be whatever value is inside of `importantNumber`.

But since `this` doesn't refer to the `gameObject` anymore, we need to pass that into the context given by that third argument as well.

Indicating Selections

Coloring the pegasus is great, the "playing" aspect of our game is figured out, but we need to consider our presentation. If we want someone playing our game to know where to click and how to actually color in the pegasus, we're going to need to work on the interface.

You may have noticed that clicking on some parts of the pegasus don't perfectly line up with the shapes. This has to do with how the shapes are stacked on top of each other. There's a lot of ways to fix that potentially, but a solution that will work well is giving the person playing the game some insight into what part of the Pegasus will be colored after they click.

We're going to use the screen blend mode to indicate which part of the Pegasus is going to be selected. Blend modes dictate how a filled-in shape interacts with the other shapes visible, somewhat like changing the opacity and color of the shape. The result will be that our highlighted section will turn semi-transparent and white with a white outline.

Indicating Palette Selection

Our game is a lot more playable now that people know what they're clicking on, but how can we keep track of what color is currently active? By adding relevant hover events to our `paletteCircle`s!

Adding a Sprite

Let’s jump right in and make a sprite. We’ve made sprites before using the method `this.add.sprite()`, but this time around, we’re going to use `this.physics.add.sprite()` and have our sprite affected by physics! In this case, our sprite will feel the effects of gravity.

The method `this.physics.add.sprite()` takes 3 arguments:

- The first argument sets the x-coordinate of the sprite’s center.
- The second argument sets the y-coordinate of the sprite’s center.
- The third argument is the key of the image loaded in the `preload()` function.

Here’s the actual syntax:

```
function create() {
  this.physics.add.sprite(320, 300, 'player');
}
```

In the example above, we create a `'player'` sprite with its center at the coordinate (320, 300) that will follow the physics of our game. The sprite created from the example above will continue to fall indefinitely. Let’s see this in action for ourselves!

Implementing Physics

In the previous exercise, we saw our sprite carried away by our game’s gravity. The reason we were able to implement gravity was through the use of a *physics plugin* which decide how GameObjects interact with each other. The plugin we’re going to use is Phaser’s Arcade physics plugin — this plugin is great for adding gravity to our game and detecting collisions.

To add the Arcade physics plugin to our `config` object, we need to add a `physics` property and provide additional specifications in its value:

```
const config = {
  // ...
  physics: {
    default: 'arcade',
    arcade: {
      gravity: { y: 300 },
      debug: true
    }
  }
};
```

Notice that in the value of our `physics` property, our object has two keys, `default` and `arcade`.

- `default` has a value of `'arcade'` which tells Phaser to use the Arcade physics plugin.
- `arcade` is another object that contains details about how we want the Arcade physics to work. The `arcade` object has two keys:
    - `gravity` which is set to `{ y: 300 }` to assign a downward gravity. (The value 300 is based on personal preference, the higher the number, the stronger the effect of gravity is)
    - `debug` with a value of `true` to see the outline of objects in our game and their velocity.

When we call `this.physics.add.sprite()`, we're actually telling Phaser's physics plugin to create our sprite for us and have this sprite follow the physics the game.

Adding Static Groups

We're gonna get that player sprite to land on some steady ground. But first, we need to create a ground platform! This ground platform will not be affected by gravity but we want it to interact with the player sprite. Therefore, we can't use `this.add.image()` like we do for our background image since images aren't affected by physics and don't interact with GameObjects.

What we need is `this.physics.add.staticGroup()` to create a *Group object* which keeps track of our platforms. Group objects are used to create and maintain sprites in a group. In this case, we can use this Group object to create additional platforms:

```
function create() {
  const platforms = this.physics.add.staticGroup();
  platforms.create(320, 350, 'platform');
}
```

From the example above, we've created a static Group Object and saved it to the `platforms` variable. Our `platforms` won't be affected by the game's gravity. Then, we call `platforms.create()` with these 3 arguments:

- The first argument is the x-coordinate of the sprite's center.
- The second argument is the y-coordinate of the sprite's center.
- The last argument is the `key` of the sprite's image.

Let's add this platform for Codey!

Instructions

1.

In `create()`, call `this.physics.add.staticGroup()` and assign it to a variable `platforms`.

2.

In a line under defining the `platforms` variable, call `platforms.create(225, 510, 'platform')`.

You will see a teal platform appear at the bottom of the screen, but Codey should still be passing through. We'll fix this issue in our next exercise.

Detecting Collisions

We have a platform sprite, we have a player sprite, but they're not interacting with each other! What's missing? Remember, our physics plugin determines how GameObjects interact. The way to decide these interactions is to create a *Collider* object that checks if two GameObjects bump into each other.

To set a Collider object we need to call `this.physics.add.collider()`. The `.collider()` method takes three arguments (the last one argument is optional). The first two arguments are the GameObjects (or Group objects) that collide. Here's an example of a collider with two arguments:

```
function create() {
  const player = this.physics.add.sprite(100, 100, 'player');
  const platform = this.physics.add.sprite(100, 500, 'platform');
  this.physics.add.collider(player, platform);
}
```

In the example above, we created a Collider object by calling `this.physics.add.collider(player, platform)`. Now, the `player` and `platform` objects don't overlap when they bump into each other.

While we're on the topic of collisions, we can call the `.setCollideWorldBounds(true)` for GameObjects that we want to stay inside the screen of the game. For example, calling `player.setCollideWorldBounds(true)` will make it so the player sprite can't fall off the screen!

Adding Controls and Velocity

Until we can telepathically instruct characters in games, we need physical controls to direct our player characters on how to move.

One way we can implement controls is by using Phaser's `this.input.keyboard.createCursorKeys()` method to create a useful object for keyboard inputs. The created object has properties `up`, `down`, `left`, `right`, `space`, and `shift` that are directly related to the keyboard keys. We can also access the `.isDown` property to the key to see if it is pressed. If the key is pressed, we can change the horizontal velocity of a GameObject by using the `.setVelocityX()` method. For instance:

```
function create() {
  // Previous code ...
  gameState.cursors = this.input.keyboard.createCursorKeys();
}

function update() {
  if (gameState.cursors.left.isDown) {
    heroSprite.setVelocityX(-160);
  } else if (gameState.cursors.right.isDown) {
    heroSprite.setVelocityX(160);
  } else {
    heroSprite.setVelocityX(0);
  }
}
```

In our `update()` function, we created the `cursors` object using `this.input.keyboard.createCursorKeys()`. Then, we added conditionals to check if the left arrow key is being pressed (`cursors.left.isDown`) or if the right arrow key is being pressed (`cursors.right.isDown`). If the left arrow key is pressed we set the horizontal velocity using `setVelocityX()` with an argument of `-160` to move the sprite left. The opposite happens when the right arrow key is pressed, we call `.setVelocityX(160)` on `heroSprite` and it moves to the right. If neither keys are pressed, we set the velocity to zero and the sprite does not move.

Adding Enemies

At this point, we have a controllable player sprite, a platform, and a collider set up. Let's make some enemies!

Unlike the player sprite, we probably want multiple enemies to triumph over. And unlike the platform, we should have physics affect them. Once again, Phaser has a handy method for us, `this.physics.add.group()` which returns a Group object that we can use to organize multiple enemy sprites. For instance:

```
function create() {
  const enemies = this.physics.add.group();
  enemies.create(320, 10, 'enemy');
}
```

From the example above we called `this.physics.add.group()` and assigned it to a variable, `enemies`. Then we call `enemies.create(320, 10, 'enemy')` to create one sprite centered at the coordinates (320, 10) using the `'enemy'` key.

But with our current code, we know that our enemy will always appear at the coordinates (320, 10). That's not much of challenge. Instead, we can randomize where this enemy's x-coordinate using `Math.random()` and multiply that value by the width of the screen. The logic being:

- `Math.random()` returns a value from 0 to 1, e.g. `0.12418156798374347`.

- When we multiply the value from `Math.random()` with the width of the screen (in pixels), we get a value that is between from 0 (left-hand side) to the width of the game(right-hand side).
- The product is a random x-coordinate that is always on the screen.

Our updated code:

```
function create() {
  const enemies = this.physics.add.group();
  const xCoordinate = Math.random() * 450;
  enemies.create(xCoordinate, 10, 'enemy');
}
```

Since we want to create multiple enemy sprites, we can use a function to house the logic for enemy creation:

```
function create() {
  const enemies = this.physics.add.group();
  function generateEnemy () {
    const xCoordinate = Math.random() * 450;
    enemies.create(xCoordinate, 10, 'enemy');
  }
}
```

With our updated code, we can create an enemy sprite every time we call `generateEnemy()`. Let’s bring this logic into our game!

Timed Events

Currently, our game creates a few enemy sprites and stops. But we could create a loop to consistently create more enemies to make the game more challenging. To implement this loop we can call `this.time.addEvent()` and pass in an object with specifications for how we want this loop to run. For instance:

```
function create() {
  function generateEnemy () {
    // Code to create an enemy sprite
  }
  const enemyGenLoop = this.time.addEvent({
    callback: generateEnemy,
    delay: 100,
    callbackScope: this,
    loop: true,
  })
}
```

In our example, we accessed the Scene’s `time` property and called `.addEvent()` with an object representing the event we want called. That object has 4 keys that each provide different specifications:

- `callback` has a value of `generateEnemy` which means this event will call `generateEnemy()` function.

- `delay` has a value of `100`, which determines, in milliseconds, how long is the delay before executing the callback again.
- `callbackScope` has a value of `this`, which selects the Scene this event is used in.
- `loop` has a value of `true`, which means that this event will continue to execute until we remove it.

Removing Enemies

Our enemy sprites threaten to take over our game if we don't remove them! We're going to need to use Phaser's `.destroy()` method to remove enemy sprites from our game.

Let's first take a second to consider when we want our enemies removed. For our game, bugs should disappear when they hit the ground. Therefore, we need a Collider for `enemies` and `platforms`. Unlike the one we previously created, this Collider takes a third argument of a callback function.

```
function create() {
  // …
  this.physics.add.collider(enemies, platforms, function(singleEnemy) {
    singleEnemy.destroy();
  });
}
```

In our example, we called `this.physics.add.collider()` with three arguments:

- The first two arguments are Group objects, `enemies` and `platforms`.
- The third argument is a callback function that has a parameter, `singleEnemy`:
  - The ordering of the callback function's parameter corresponds to the ordering of `.collider()`'s first two arguments.
  - We call `singleEnemy.destroy()` which will remove the enemy's sprite when it collides with a platform.

Adding a score

One thing we're still missing from our game is a scoring system to motivate our players. We can store our score as property in the `gameState` object and increase it as the game progresses. To display the score we can call `this.add.text()`. We should also consider when we want to increase the score, e.g. increase the score every 10 seconds, or each time an enemy is generated, or how much the player sprite moves, etc…

For our game, we could increase the score each time the player sprite dodges a bug. When a player dodges a bug, it collides with the platform, so we can add our logic in that Collider object. Let's do that now!

When a game starts we should set a new property with a key of `score` and value of `0` in the `gameState` object like so:

```
const gameState = { score: 0 };
```

We’ll also need `gameState` to display the score on screen. In `create()` we would add:

```
gameState.scoreText = this.add.text(16, 16, 'Score: 0', { fontSize: '15px',
fill: '#000000' })
```

Now we can access `gameState.scoreText` and change the text to display the increased score. Since we want to increment `gameState.score` when a bug collides with the ground platform, we need to add that logic to our ColliderObject:

```
this.physics.add.collider(bugs, platforms, function (bug){
  bug.destroy();

  gameState.score += 10;
  gameState.scoreText.setText(`Score: ${gameState.score}`)

})
```

Losing Condition

It’s great to see our score increase, but it doesn’t mean much if there’s no way to lose. One common losing condition is when a player sprite collides with an enemy sprite. This means we need to include another Collider object with a callback in `create()`:

```
this.physics.add.collider(player, enemies, () => {
  // Logic to end the game
});
```

For our game, it ends when the player sprite a bug collide. When this event happens we also want certain things to stop. For instance, we don’t want our game to continue creating bugs. We could call `.destroy()` on the loop that creates our bugs. Another thing we would want to stop is the physics of our game. To put a pause on physics, we call `this.physics.pause()`.

In the code example above, notice the use of an arrow function (`() => {}`) for a callback instead of anonymous function (`function() {}`). The reason for using an arrow function is that it implicitly binds `this`. We know that we want to call `this.physics.pause()` and we need `this` to reference the Scene object. Therefore, we use an arrow function to bind `this` as the Scene object. To read more about arrow functions and `this`, check out MDN’s arrow function documentation

Resetting A Scene

Once a game ends, we should allow players to restart and try again! Phaser has a convenient method to solve this issue: `this.scene.restart()`.

Like the method’s name implies, `this.scene.restart()`, restarts the Scene.

We should also consider when we want to restart the scene, each game has different criteria. For our game, we want this option to be available after the player sprite has collided with a bug. We should also give our players an option to restart when they choose to.

One way to implement this logic is to add on to the callback function of the player and bugs Collider object. We'll add an event listener for a mouse click, or `'pointerup'` event. When this event occurs, then we'll restart the Scene. Let's add this feature in.

Building a Multi-Scened Phaser Game

Phaser games are defined by Scenes. In a simple game, we might only use one Scene. In more elaborate games, we can use multiple Scenes to build out start screens, end screens, or even different levels.

In this lesson, we're going to *refactor* (rewrite the code differently but still achieve the same outcome) an existing game's code to include multiple Scenes. Along the way, we'll discuss how to organize our code so that it is easier to maintain and debug.

With these techniques under our belts, we'll be well on our way to build out multi-Scened games with Phaser!

Using Classes in Phaser (ES6)

In this exercise, we're going to refactor the code from the Learn Phaser: Physics Lesson.

One major change that we're going to implement is the inclusion of JavaScript's `class` syntax. Each `class` will represent a Scene and contain the familiar `preload()`, `create()`, and `update()` functions. We're going to refer to these functions as *methods* since they're functions that belong to a class. Below is a sample template:

```
class ExampleScene extends Phaser.Scene {
  constructor(){
    super({ key: 'ExampleScene' });
  }
  preload() {
    // ...
  }
  create() {
    // ...
  }
  update() {
    // ...
  }
}
```

Our `ExampleScene` class is a subclass of `Phaser.Scene`:

- It has a `constructor()` method that is used to create an initialize the Scene object.
- Inside the `constructor()` method, we call `super()` which calls the `constructor()` of the `Phaser.Scene`.
    - We provide `super()` with the object `{ key: 'ExampleScene' }` so that we can refer to this `class` in our `config` object.
- `preload()`, `create()`, and `update()` are included but are now methods, notice the lack of the `function` keyword.

In order for our Phaser game to know about this class, we'll also need to change the `config` object:

```
const config = {
  // …
  scene: [ExampleScene]
};
```

In the code above, our `config` object's `scene` property has a value of an array. While the only element currently inside the array is `ExampleScene`, we can later add more scenes to this array.

By refactoring our code, we're laying the groundwork to include more scenes in this existing game. In later exercises, we can add scenes for starting and ending the game!

New Scenes

Since we're now using the `class` syntax in our Phaser game, we can add another `class` that will render a start screen for our players. This start screen will give our players a chance to ready themselves before actually playing the game.

Recall the template for creating a Scene:

```
class FirstScene extends Phaser.Scene {
  constructor(){
    super({ key: 'FirstScene' });
  }
  // ...
}
```

Notice we supplied super() with the key of `'FirstScene'`. We'll then reference this key in our `config`:

```
const config = {
  // …
  scene: [FirstScene, AnotherScene]
};
```

This time around, we have two elements inside `scene`'s array. The ordering is important in determining which Scene the game plays first. For the example above, the game will play `FirstScene` and not play `AnotherScene` until prompted.

Scene Transitions

Phaser has built-in methods that make it easy for us to transition from one Scene to another. For us to transition between Scenes we have to `.stop()` the playing of a Scene and `.start()` the next Scene.

To stop a Scene, call `this.scene.stop('keyOfScene').`

Then, to start a Scene, call `this.scene.start('keyOfAnotherScene').`

The `.stop()` and `.start()` methods take in a string that has the value of a Scene's key. Like the name of the methods imply, `.stop()` will stop the Scene and `start()` will start playing the Scene.

Separate Files

Having two Scenes, a `gameState` object, and our `config` object in the same file can make our code feel cluttered. As we add more Scenes, this file will continue to grow and it can become difficult to maintain.

One way to remedy our growing code is to reorganize our code into different files and import them into the same game using `<script>` elements inside our **index.html** file.

Currently, the `<body>` of **index.html** looks like:

```
<body>
  <script
src="https://cdn.jsdelivr.net/npm/phaser@3.16.2/dist/phaser.min.js"></scrip
t>
  <script src="game.js"></script>
</body>
```

We can break up the code in **game.js** into files, each one containing code for a specific scene so that our `<body>` looks more like:

```
<body>
  <script
src="https://cdn.jsdelivr.net/npm/phaser@3.16.2/dist/phaser.min.js"></scrip
t>

  <script src="firstScene.js"></script>
  <script src="secondScene.js"></script>
  <script src="thirdScene.js"></script>
  <script src="game.js"></script>
</body>
```

We'll keep each Scene's code inside separate files to make it easier to maintain our growing code and avoid the need to scroll through a single gigantic game file.

Introduction to Animations and Tweens in Phaser

A static image can only convey so much information. Imagine using a static image for a Sprite and moving it around the game. Will players really get a sense of when it's walking? or running? Does it stay the same when it's jumping? Maybe just standing idly by waiting for the player to take control? Instead we can opt to use animations to distinguish between the many actions our Sprites can do.

In this lesson we'll learn how to take a *sprite sheet* (a single image file that has a Sprite in different poses) and animate it using Phaser. We'll also introduce another visual tool, *tweens*, into our games. We'll take this newfound knowledge and add another dimension to how sprites look, move, and interact inside our Phaser games.

The game we're developing here is a simplistic platformer which will help us visualize animations and tweens. As the game develops, you'll notice we provide for you additional code that builds out the game. When you see our provided code, take some time to read through the comments and refresh yourself on what code is doing — afterall, it's all concepts we'll gone over before!

Now, let's breathe some life into our sprites!

Sprite Sheets

One common tool used to create animations is a sprite sheet that contains all the images that depict how a sprite can move. Take for instance:

As we move through the *frames* (individual images) of the sprite sheet, Codey starts walking!

To implement an animation in our game we need to:

1. Load in the sprite sheet.
2. Create the sprite object.
3. Create the animation by selecting specific frames from the sprite sheet.
4. Play the animation.

Let's focus first on loading in our sprite sheet:

```
class ExampleScene extends Phaser.Scene {
  preload() {
    this.load.spritesheet( 'spriteKey' , 'spriteSheet.png', { frameWidth:
100, frameHeight: 100 });
  }
}
```

`this.load.spritesheet()` takes three arguments: the sprite's key as a string, the location of the sprite sheet, and an object with the properties `frameWidth` and `frameHeight` these properties indicate how many pixels wide and tall the individual frames are. Be sure to use accurate values since inaccurate `frameWidth` or `frameHeight` values will result in a misshaped sprite or a nonfunctioning animation!

Now, let's load in Codey's sprite sheet!

Creating the Animation

With our sprite sheet loaded in, we can now create our sprite object and the animation sequence.

This logic goes inside our `create()` method:

```
class ExampleScene extends Phaser.Scene {
  // … Previous code
  create() {
    gameState.exampleSprite = this.physics.add.sprite(100, 600,
'spriteKey');
  }
}
```

The code above should look familiar, it's the same way we create a sprite object from a single image. When our sprite loads in game, it'll show the first frame of our sprite sheet.

We can now use `this.anims.create()` to create our animations:

```
class ExampleScene extends Phaser.Scene {
  create() {
    gameState.exampleSprite = this.physics.add.sprite(100, 600,
'spriteKey');

    this.anims.create({
      key: 'movement',
      frames: this.anims.generateFrameNumbers('spriteKey', { start: 0, end:
5 }),
      frameRate: 10,
      repeat: -1
    });
  }
}
```

`this.anims.create()` takes an object as an argument that has several properties. In the example above we included:

- `key` - how this animation will be referenced.
- `frames` - which frames of the sprite sheet we're using

    - `this.anims.generateFrameNumbers('spriteKey', { start: 0, end: 5 })` is a Phaser method that returns an array of a sprite sheet's frames from `start` up to (and including) `end`.
- `frameRate` - how many frames play per second (it will default to `24` if `frameRate` is not provided).
- `repeat` - how many times the animation repeats, use `-1` to continuously repeat the animation.

To see what other properties we could include check out RexRainbow's Phaser animation documentation

Animating

✓ Load in sprite sheet
✓ Create the sprite obect
✓ Create animation from sprite sheet
☐ Animate sprite

We have everything else set up, now let's bring our sprite to life!

Inside `update()`, we can include our logic for controlling our sprite and animating it however we want.

```
class ExampleScene extends Phaser.Scene {
  // … Previous code
  update() {
    if (gameState.cursors.right.isDown) {
      gameState.exampleSprite.setVelocityX(100);
      gameState.exampleSprite.anims.play('movement', true);
    }
  }
}
```

In the code above, if the right arrow key is pressed `gameState.exampleSprite` moves to the right. Then we play the animation by calling `gameState.exampleSprite.anims.play('movement', true)`:

- `gameState.exampleSprite.anims` allows us to access all the animations we created.
- `anims.play()` will play all the animations, or a single animation if passed an argument.
- We provide `.anims.play()` with two arguments:
    - the first is an animation key, in this case it's the `movement` animation.
    - the second is a boolean, which won't play the animation from the start, if it's already in progress.

Flipping an Animation

Codey's animation looks great moving to the right. But if we apply the exact same logic to move left, Codey looks like they're being blown back by some wind or doing the moonwalk. (While it's cool, it isn't exactly what we're going for).

As a fix, we need to set the `.flipX` property of the animation to be `true` or `false` depending on which direction we want our sprite to turn.

```
class ExampleScene extends Phaser.Scene {
  // … Previous code
  update() {
    if (gameState.cursors.right.isDown) {
      gameState.exampleSprite.setVelocityX(100);
      gameState.exampleSprite.anims.play('movement', true);
      // The sprite is facing its original direction
      gameState.exampleSprite.flipX = false;
    } else if ( gameState.cursors.left.isDown) {
      gameState.exampleSprite.setVelocityX(-100);
      gameState.exampleSprite.anims.play('movement', true);
      // The sprite is facing its flipped direction
      gameState.exampleSprite.flipX = true;
    }
  }
}
```

When our Sprite is moving to the right, we assign `.flipX` to false, since our sprite sheet original has the sprite facing the right. Otherwise, when moving left, we assign `.flipX` to true so that it's flipped on its x-axis and facing left.

By manipulating `.flipX` we cam animate our sprite with a sprite sheet that faces one direction!

Let's now add this into our game!

Pausing Animations

Learning how to start an animation is one thing, pausing it is another. When it's game over, or if we decide to include a pause option, we might opt to pause one or all of our animations.

In Phaser, we can call `exampleSprite.anims.pause()` on the sprite to put a pause on its animation.

We can also call `this.anims.pauseAll()` to pause all animations in a Scene.

Let's try out these convenient methods in our own game.

Tweens

While animations allow us to play through the frames of a sprite sheet, *tweens* help refine the transition from frame to frame. By creating *in-between* frames, sprites undergoing changes like their size and positions appear smoother.

One common usage of tweens is to convey movement, for instance:

```
class ExampleScene extends Phaser.Scene {
  // … Previous code
  create () {
    gameState.exampleSprite = this.physics.sprite.add(0, 100, 'example');

    gameState.moveTween = this.tweens.add({
      targets: gameState.exampleSprite,
      x: 300,
      ease: 'Linear',
      duration: 3000,
      repeat: -1,
      yoyo: true
    });

    gameState.moveTween.play();

    // Later on …
    gameState.moveTween.stop();
  }
}
```

In the code above we called `this.tweens.add()` to create a tween saved to `gameState.moveTween`. The object that we provided as an argument:

- `targets` determines which Sprites are affected (we could've also used an array)
- `x` determines the final x-coordinate of `gameState.exampleSprite`.
- `ease` describes how the tween plays.
  - We provided a value of `Linear` which means it plays at a constant speed. But if we wanted some variation, we could have provided another easing function.
- `duration` determines how long the tween lasts (in milliseconds).
- `repeat` is how many times the tween runs (use `-1` to continuously play).
- `yoyo` is a `true` or `false` value, if it's `true`, the tween plays in reverse for the Sprite to return back to its original state (size, position, angle, etc.) before the tween started. If it's `false`, then the Sprite will remain as is after the tween finishes.
- To play the tween, we call `.play()` on `gameState.moveTween`.
- To stop playing the tween, we call `.stop()` on `gameState.moveTween`.

Tween Callbacks

What if we wanted something to happen after a tween finished playing? Or while it's looping? How about before it starts playing? Conveniently, Phaser allows us to provide the tween with callback functions.

To use these callbacks, inside the object we passed to `this.anims.add()`, we assign the following keys callback functions:

- `onStart` - if we want a function to execute when the tween starts.
- `onYoyo` - if we want a function to execute when the tween starts going back to the original position.
- `onRepeat` - if we want a function to execute each time the tween plays.
- `onComplete` - if we want a function to execute when the tween finishes.

If we wanted to remove our sprite after its tween finished playing, we could call:

```
class ExampleScene extends Phaser.Scene {
  // … Previous code
  create () {
    gameState.moveTween = this.tweens.add({
      target: gameState.exampleSprite,
      x: 300,
      ease: 'Linear',
      duration: 3000,
      repeat: 1,
      yoyo: true,
      // Will execute after the tween finishes playing:
      onComplete: function() {
        gameState.exampleSprite.destroy();
      }
    });

    gameState.moveTween.play();
  }
}
```

With our tween in place, after our tween plays, the callback for `onComplete` executes, and destroys `gameState.exampleSprite`!

Cameras

In order to grant the joy of exploration and discovery it becomes important for the game world to be larger than what a player immediately sees upon entering the game. A useful way to convince the player of your game of this is by employing the use of cameras. A *camera* is a view of the game world. Some games (those with mini-maps, for instance) employ multiple cameras, but we will be focusing on just using one.

Phaser creates a camera by default, which we have so far been using without any customization. Its bounds are set to be the same as the dimensions for your Game given in `config`. It also does not move at all, but we will be changing both of those things for our platformer.

```
create() {
  gameState.player = this.physics.add.sprite(100, 100, 'codey');
  this.cameras.main.setBounds(0, 0, 2000, 600);
  this.cameras.main.startFollow(gameState.player);
}
```

Above we modified the default camera, `this.cameras.main`, to make it aware that the world is larger than our current window. We do this by calling `.setBounds()` and giving bounds larger than our `config` width and height. We also tell the camera to follow along our main player by calling `.startFollow()` and passing it the sprite we want to follow.

The `.startFollow()` method also has several additional optional arguments you can pass which we can use to customize how fast the camera locks-on to its target and whether the target should be in the center of the screen exactly or offset somewhat. Here are the arguments for `.startFollow()`: `(target, roundPixels, lerpX, lerpY, offsetX, offsetY)`.

- `target` is the sprite for the camera to follow
- `roundPixels` is a boolean that effects rendering, set it to `true` if you're experiencing camera jitter.
- `lerpX` and `lerpY` are the speed (between 0 and 1, defaults to 1) with which the camera locks on to the target. A lower lerp speed will have the effect that your character is moving much faster than the camera.
- `offsetX` and `offsetY` is the offset for the camera. Set `offsetX` to something like `-200` will keep Codey on the left side of the screen (so that more of the level ahead is on the screen).

Levels

Now that we have a system for specifying where platforms should be in each level, we can create multiple levels! Each level is going to inherit from our customized `Level` class and then inject information about the level inside the constructor. By doing this we're able to separate each of our Levels from one-another!

Shaking the Camera

Camera shake is an indispensible effect in the modern video game. It hints to the player that something jarring and surprising is happening. When a player falls down we're going to start the level over again, but shake the camera a little bit as if to say "let's try that a little differently next time."

Camera shake is an easy effect to add in Phaser, just call the `.shake()` method on the camera. `.shake()` can take the following arguments:

- `duration` the length of the shake
- `intensity` how strong the camera shakes
- `force` whether or not to start the effect from the beginning if it has already started
- `callback` the function to call each frame while the shake is happening. Accepts two arguments: a reference to the camera and then a duration from 0-1
- `context` the context for the previous function, defaults to the Scene

This command:

```
this.cameras.main.shake(100, .8)
```

Will shake the main camera fairly vigorously for 100 milliseconds. Whereas this command:

```
this.cameras.main.shake(200, .3, true, function(camera, progress) {
  if (progress > .5) {
    gameState.player.setTint(0xff0000);
  }
});
```

Will lighlty shake the camera for 200 milliseconds. Halfway through the shake's completion it will turn `gameState.player` red.

- `context` the context for the previous function, defaults to the Scene

Fading Out

Fading out of a Scene seems like a fitting transition. A cue taken from the film industry, fade-out offers a much softer effect than the shake but is as concise in Phaser. `.fade()` is a camera method that takes the following arguments:

- `duration`, the length of the fade in milliseconds.
- `red`, the 0-255 integer value of how red the fadeout color is.
- `green`, the 0-255 integer value of how green the fadeout color is.
- `blue`, the 0-255 integer value of how blue the fadeout color is.
- `force`, starts the fadeout over if it's already been started.
- `callback`, the callback to use during the fadeout effect.
- `context`, the context to use for the callback function (defaults to the Scene the camera is in).

We can call:

```
this.cameras.main.fade(100, 255, 255, 255, false, function(camera,
progress) {
  if (progress > .5) {
    gameState.player.x = 5;
  }
});
```

In the above code the camera fades to white very quickly. Halfway through the fadeout, the `gameState.player` gets moved 5 pixels from the left edge of the game.

Parallax Scrolling

We're going to create the illusion of depth in our frozen tundra world using something called a *parallax effect*. Parallax motion refers to an observable real-world phenomenon that things closer to us move faster than things further from us. A full moon or a faraway mountain may not appear to move much at all as you drive down a highway on a clear night, but it's simply so far away that the movement is occurring very slowly. By comparison, a hedge of trees against the road or a neon light for a roadside shop will whizz by.

In order to simulate this motion, we can to create three different background layers of different sizes. Why do they need to be different sizes? Since we'll be scrolling some of our "nearer" layers very fast and some of our "farther" layers very slow, we will need our "nearer" layers to be longer or else they will scroll off the screen.

Determining The Scroll Factor

In order to accomplish this effect, we'll update the *scroll factor* of each of our background layers. The scroll factor is how fast an object scrolls (with respect to our camera). By default, all GameObjects have a scroll factor of 1 (scrolls as fast as everything else). A static object that we always want on-screen should have a scroll factor of 0. We can set the scroll factor with the GameObject method `.setScrollFactor()`. But what should we set the scroll factor to?

The scroll factor is the rate we want our background to move. There are three numbers that we'll use to determine it: the width of the game, the width of the background, and the width of the window itself. How do we determine the rate that a smaller background should move so that the entire background only shifts as fast as the player moves across the level? With a formula!

```
const windowWidth = config.width;
const gameWidth = 2000; // the size of the largest background
const someBackgroundWidth = gameState.someBackground.getBounds().width

const someBackgroundScrollFactor = (someBackgroundWidth - windowWidth) /
(gameWidth - windowWidth)
gameState.someBackground.setScrollFactor(someBackgroundScrollFactor
```

Changing the Weather

If a long journey goes on for several days, how do we communicate that length to the folks playing our game? We're going to use a few strategies to add in concepts of ambience, lighting, time of day, and weather to our game to make these same assets feel like the world they're apart of turns and changes like our own.

The sky's color is the easiest thing for us to change, by changing the background color of the game. Since we actually want to update the color of the sky multiple times within our single game, we can make a new "background", a rectangle with the same dimensions as our canvas. We will be able to update the color of our new background with the appropriate color of the sky.

Changing the Lighting

Different light colors things differently, so it will be a stronger effect if we color our world for each time of day. In order to convey this effect we are going to use the `.setTint()` method on each of our GameObjects. `.setTint()` performs a color multiplication effect that changes each pixel in your image consistently, in a way similar to having a colored light cast on it. It's a lot like looking at the same thing through a pair of sunglasses. When we do this, we can contrast the effect of a strong overhead sun during the afternoon, with, say, a lavender light present in the early morning. At night, we can just make everything a little bit darker.

Making It Snow

Where does all of the snow in our game world come from? We assume it falls from the sky, but how? In order to enhance the realism of our game we want the snow to fall as the player crosses through the tundra. In order to do that we'll need to use a *Particle Emitter*. A Particle Emitter creates a pool of particles. A particle is a small, repeated sprite. A pool is a collection of these particles collected for reuse. Instead of constantly dropping new snowflakes, we will "move" the snowflakes that have dropped already to the top of the screen and drop them down again. We can add our Particle Emitter in `create()` like so:

```
create() {
  gameState.particles = this.add.particles('marble');

  gameState.emitter = gameState.particles.createEmitter({
    x: { min: 0, max: 200 },
    y: 0,
    lifespan: 2000,
    speedY: { min:10, max: 200},
    scale: .2,
    quantity: 10,
    blendMode: 'ADD'
  })
}
```

This creates a particle emitter that uses a `'marble'` asset we've preloaded. It creates the particles at the top of the screen and pushes them down screen. It creates these "marbles" at x-coordinates between `0` and `200` and gives them variable speeds as they fall down the screen.