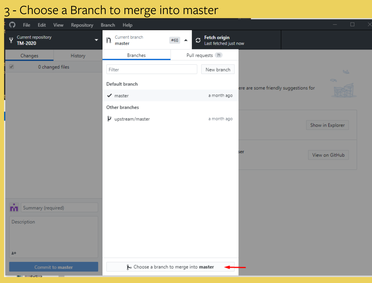
Phaser3

Phaser is an HTML5 game framework which aims to help developers make powerful, cross-browser HTML5 games really quickly. It was created specifically to harness the benefits of modern browsers, both desktop and mobile. The only browser requirement is the support of the canvas tag.

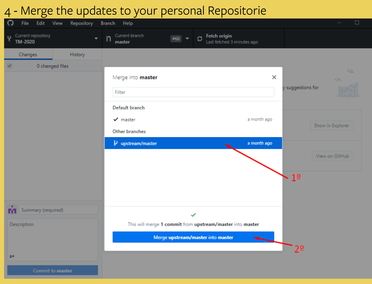
Part 1 - Starting

First Steps

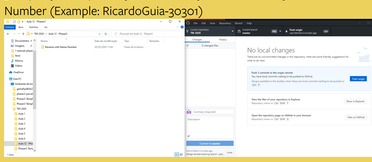
As Usual for the TM Lessons go to your github application:
1 - Fetch Origin
2 - Current Branch
3 - Choose a Branch to merge into master



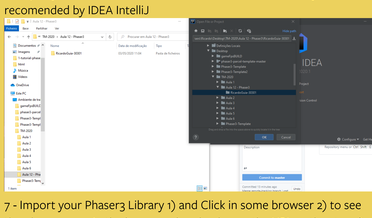
4 - Merge the updates to your personal Repositorie



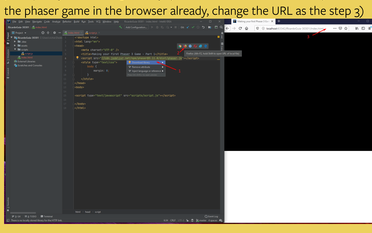
5 - Open your repository and rename your folder with your Name-Number (Example: RicardoGuia-30301)



6 - Open your folder with IntelliJ and add the external Library as recomended by IDEA IntelliJ



7 - Import your Phaser3 Library 1) and Click in some browser 2) to see the phaser game in the browser already, change the URL as the step 3)



Open the `script.js` page in IntelliJ and let's have a closer look at the code.

```
var config = {
    type: Phaser.AUTO,
    width: 800,
    height: 600,
    scene: {
        preload: preload,
        create: create,
        update: update
    }
};

var game = new Phaser.Game(config);

function preload ()
{
}

function create ()
{
}

function update ()
{
}
```

The config object is how you configure your Phaser Game. There are lots of options that can be placed in this object and as you expand on your Phaser knowledge you'll encounter more of them.

An instance of a Phaser.Game object is assigned to a local variable called `game` and the configuration object is passed to it. This will start the process of bringing Phaser to life.

In Phaser 2 the `game` object acted as the gateway to nearly all internal systems and was often accessed from a global variable. In Phaser 3 this is no longer the case and it's no longer useful to store the game instance in a global variable.

The `type` property can be either `Phaser.CANVAS`, `Phaser.WEBGL`, or `Phaser.AUTO`. This is the rendering context that you want to use for your game. The recommended value is `Phaser.AUTO` which automatically tries to use WebGL, but if the browser or device doesn't support it it'll fall back to Canvas. The canvas element that Phaser creates will be simply be appended to the document at the point the script was called, but you can also specify a parent container in the game config should you wish.

The `width` and `height` properties set the size of the canvas element that Phaser will create. In this case 800 x 600 pixels. Your game world can be any size you like, but this is the resolution the game will display in.

Part 2 - Loading Assets

Let's load the assets we need for our game. You do this by putting calls to the Phaser Loader inside of a Scene function called `preload`. Phaser will automatically look for this function when it starts and load anything defined within it.

Currently the `preload` function is empty. Change it to:

```
function preload ()
{
    this.load.image('sky', 'assets/sky.png');
    this.load.image('ground', 'assets/platform.png');
    this.load.image('star', 'assets/star.png');
    this.load.image('bomb', 'assets/bomb.png');
    this.load.spritesheet('dude',
        'assets/dude.png',
        { frameWidth: 32, frameHeight: 48 }
    );
}
```

This will load in 5 assets: 4 images and a sprite sheet. It may appear obvious to some of you, but I would like to point out the first parameter, also known as the asset key (i.e. 'sky', 'bomb'). This string is a link to the loaded asset and is what you'll use in your code when creating Game Objects. You're free to use any valid JavaScript string as the key.

Display an Image

In order to display one of the images we've loaded place the following code inside the `create` function:

```
this.add.image(400, 300, 'sky');
```

You can find this in `part3.html`. If you load it in a browser you should now see a game screen with a blue sky backdrop covering it:

The values `400` and `300` are the x and y coordinates of the image. Why 400 and 300? It's because in Phaser 3 all Game Objects are positioned based on their center by default. The background image is 800 x 600 pixels in size, so if we were to display it centered at 0 x 0 you'd only see the bottom-right corner of it. If we display it at 400 x 300 you see the whole thing.

Hint: You can use `setOrigin` to change this. For example the code: `this.add.image(0, 0, 'sky').setOrigin(0, 0)` would reset the drawing position of the image to the top-left.

The order in which game objects are displayed matches the order in which you create them. So if you wish to place a star sprite above the background, you would need to ensure that it was added as an image second, after the sky image:

```
function create ()
{
    this.add.image(400, 300, 'sky');
    this.add.image(400, 300, 'star');
}
```

If you put the `star` image first it will be covered-up by the sky image.

Part 3 - World Building

Under the hood `this.add.image` is creating a new Image Game Object and adding it to the current Scenes display list. This list is where all of your Game Objects live. You could position the image anywhere and Phaser will not mind. Of course, if it's outside of the region 0x0 to 800x600 then you're not going to visually see it, because it'll be "off screen", but it will still exist within the Scene.

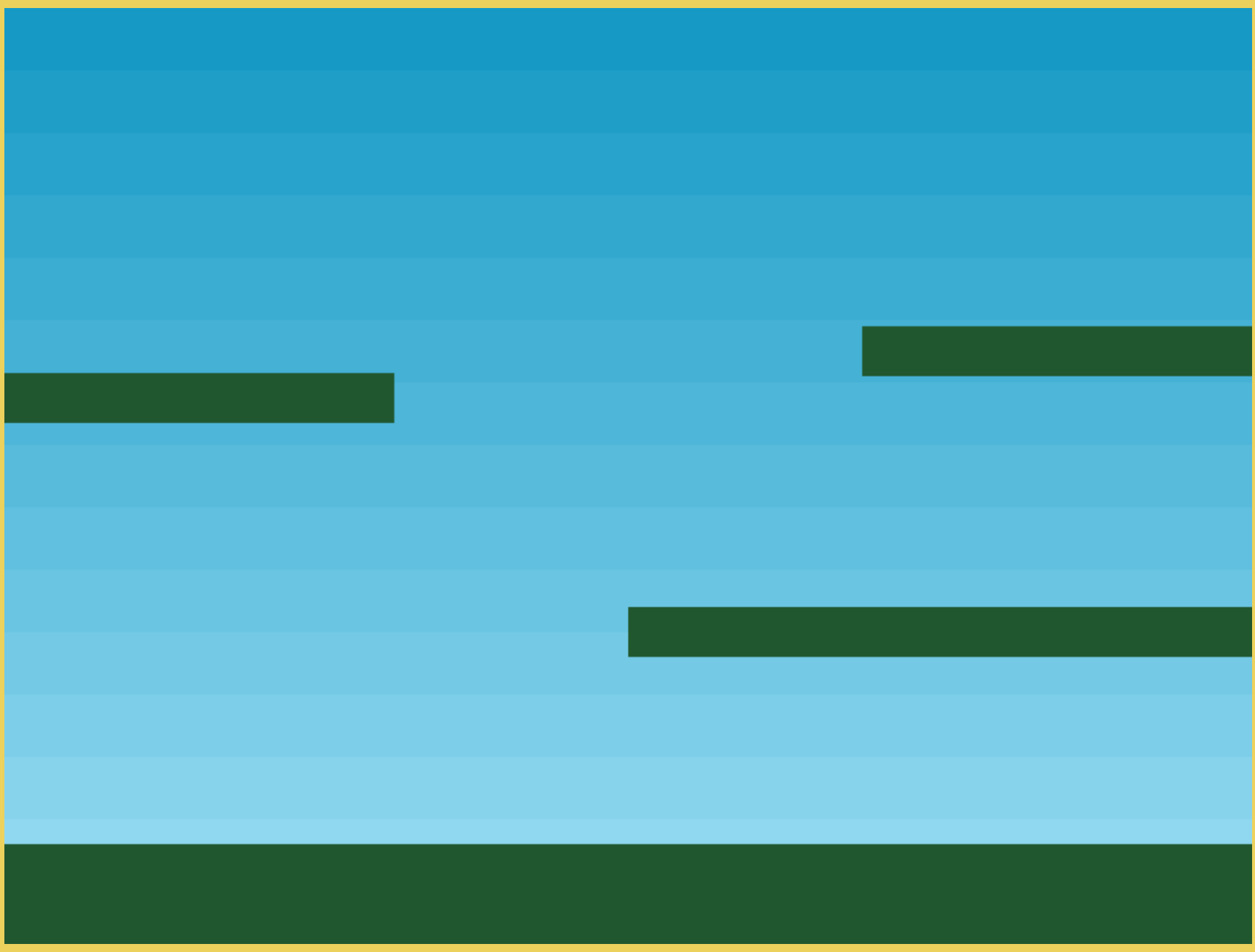
For now let's build up the Scene by adding a background image and some platforms. Here is the updated `create` function:

```
var platforms;

function create ()
{
    this.add.image(400, 300, 'sky');

    platforms = this.physics.add.staticGroup();

    platforms.create(400, 568, 'ground').setScale(2).refreshBody();

    platforms.create(600, 400, 'ground');
    platforms.create(50, 250, 'ground');
    platforms.create(750, 220, 'ground');
}
```

Glancing quickly at the code you'll see a call to `this.physics`. This means we're using the Arcade Physics system, but before we can do that we need to add it to our Game Config to tell Phaser our game requires it. So let's update that to include physics support. Here is the revised game config:

```
var config = {
    type: Phaser.AUTO,
    width: 800,
    height: 600,
    physics: {
        default: 'arcade',
        arcade: {
            gravity: { y: 300 },
            debug: false
        }
    },
    scene: {
        preload: preload,
        create: create,
        update: update
    }
};
```

The new addition is the `physics` property, now you should see a much more game-like scene:



We've got a backdrop and some platforms, but how exactly are those platforms working?

Part 4 - The Platforms

We just added a bunch of code to our `create` function that deserves a more detailed explanation. First, this part:

```
platforms = this.physics.add.staticGroup();
```

This creates a new Static Physics Group and assigns it to the local variable `platforms`. In **Arcade Physics there are two types of physics bodies: Dynamic and Static.**
A dynamic body is one that can move around via forces such as velocity or acceleration. It can bounce and collide with other objects and that collision is influenced by the mass of the body and other elements.

A Static Body simply has a position and a size. It isn't touched by gravity, you cannot set velocity on it and when something collides with it, it never moves. Static by name, static by nature. And perfect for the ground and platforms that we're going to let the player run around on.

But what is a Group? As their name implies they are ways for you to group together similar objects and control them all as one single unit. You can also check for collision between Groups and other game objects. Groups are capable of creating their own Game Objects via handy helper functions like `create`. A Physics Group will automatically create physics enabled children, saving you some leg-work in the process.

With our platform Group made we can now use it to create the platforms:

```
platforms.create(400, 568, 'ground').setScale(2).refreshBody();

platforms.create(600, 400, 'ground');
platforms.create(50, 250, 'ground');
platforms.create(750, 220, 'ground');
```

As we saw previously it creates this scene:

During our preload we imported a 'ground' image. It's a simple green rectangle, 400 x 32 pixels in size and will serve for our basic plaform needs:

The first line of code above adds a new ground image at 400 x 568 (remember, images are positioned based on their center) - the problem is that we need this platform to span the full width of our game, otherwise the player will just drop off the sides. To do that we scale it x2 with the function `setScale(2)`. It's now 800 x 64 in size, which is perfect for our needs. The call to `refreshBody()` is required because we have scaled a *static* physics body, so we have to tell the physics world about the changes we made.

The ground is scaled and in place, so it's time for the other platforms:

```
platforms.create(600, 400, 'ground');
platforms.create(50, 250, 'ground');
platforms.create(750, 220, 'ground');
```

The process is exactly the same as before, only we don't need to scale these platforms as they're the right size already.

3 platforms are placed around the screen, the right distances apart to allow the player to leap up to them.

So let's add our player.

Part 5 - Ready Player One

Create a new variable called `player` and add the following code to the `create` function.

```
player = this.physics.add.sprite(100, 450, 'dude');

player.setBounce(0.2);
player.setCollideWorldBounds(true);

this.anims.create({
    key: 'left',
    frames: this.anims.generateFrameNumbers('dude', { start: 0, end: 3 }),
    frameRate: 10,
    repeat: -1
});

this.anims.create({
    key: 'turn',
    frames: [ { key: 'dude', frame: 4 } ],
    frameRate: 20
});

this.anims.create({
    key: 'right',
    frames: this.anims.generateFrameNumbers('dude', { start: 5, end: 8 }),
    frameRate: 10,
    repeat: -1
});
```

There are two separate things going on here: the creation of a Physics Sprite and the creation of some animations that it can use.

Physics Sprite

The first part of the code creates the sprite:

```
player = this.physics.add.sprite(100, 450, 'dude');

player.setBounce(0.2);
player.setCollideWorldBounds(true);
```

This creates a new sprite called `player`, positioned at 100 x 450 pixels from the bottom of the game. The sprite was created via the Physics Game Object Factory (`this.physics.add`) which means it has a Dynamic Physics body by default.

After creating the sprite it is given a slight bounce value of 0.2. This means when it lands after jumping it will bounce ever so slightly. The sprite is then set to collide with the world bounds. The bounds, by default, are on the outside of the game dimensions. As we set the game to be 800 x 600 then the player won't be able to run outside of this area. It will stop the player from being able to run off the edges of the screen or jump through the top.

Animations

If you glance back to the `preload` function you'll see that 'dude' was loaded as a sprite sheet, not an image. That is because it contains animation frames. This is what the full sprite sheet looks like:

There are 9 frames in total, 4 for running left, 1 for facing the camera and 4 for running right. Note: Phaser supports flipping sprites to save on animation frames, but for the sake of this tutorial we'll keep it old school.

We define two animations called 'left' and 'right'. Here is the left animation:

```
this.anims.create({
    key: 'left',
    frames: this.anims.generateFrameNumbers('dude', { start: 0, end: 3 }),
    frameRate: 10,
    repeat: -1
});
```

The 'left' animation uses frames 0, 1, 2 and 3 and runs at 10 frames per second. The 'repeat -1' value tells the animation to loop.

This is our standard run-cycle and we repeat it for running in the opposite direction, using the key 'right' and a final one for 'turn'.

Part 6 - Body Velocity: A world of physics

Phaser has support for a variety of different physics systems, each acting as a plugin available to any Phaser Scene. At the time of writing it ships with Arcade Physics, Impact Physics and Matter.js Physics. We will be using the Arcade Physics system for our game, which is simple and light-weight, perfect for mobile browsers.

When a Physics Sprite is created it is given a **body** property, which is a reference to its Arcade Physics Body. This represents the sprite as a physical body in Phasers Arcade Physics engine. The body object has a lot of properties and methods that we can play with.

For example, to simulate the effects of gravity on a sprite, it's as simple as writing:

```
player.body.setGravityY(300)
```

This is an arbitrary value, but logically, the higher the value, the heavier your object feels and the quicker it falls. If you add this to your code you will see that the player falls down without stopping, completely ignoring the ground we created earlier:

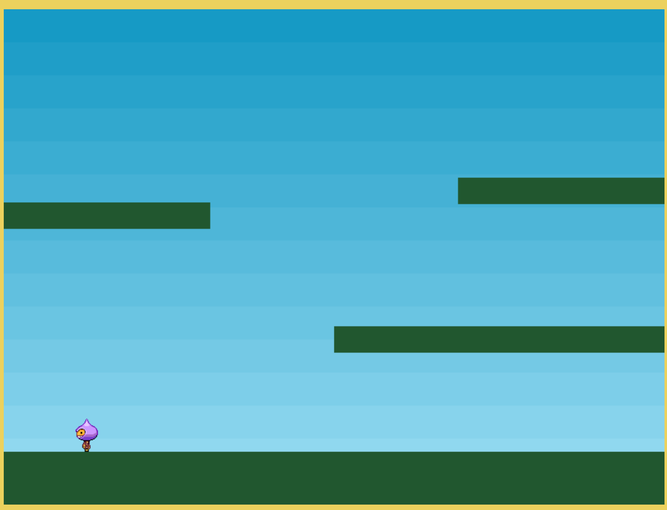
The reason for this is that we're not yet testing for collision between the ground and the player.

We already told Phaser that our ground and platforms would be static bodies. Had we not done that, and created dynamic ones instead, then when the player collided with them it would stop for a moment and then everything would have collapsed. This is because unless told otherwise, the ground sprite is a moving physical object and when the player hits it, the resulting force of the collision is applied to the ground, therefore, the two bodies exchange their velocities and ground starts falling as well.

In order to allow the player to collide with the platforms we can create a Collider object. This object monitors two physics objects (which can include Groups) and checks for collisions or overlap between them.

```
this.physics.add.collider(player, platforms);
```

The Collider is the one that performs the magic. It takes two objects and tests for collision and performs separation against them. In this case we're giving it the player sprite and the platforms Group. It's clever enough to run collision against all Group members, so this one call will collide against the ground and all platforms. The result is a firm platform that doesn't collapse:



Part 7 - Controlling the player with the keyboard

Colliding is all good and well, but we really need the player to move. You would probably think of heading to the documentation and searching about how to add an event listener, but that is not necessary here. Phaser has a built-in Keyboard manager and one of the benefits of using that is this handy little function:

```
cursors = this.input.keyboard.createCursorKeys();
```

This populates the cursors object with four properties: up, down, left, right, that are all instances of Key objects. Then all we need to do is poll these in our `update` loop:

```
if (cursors.left.isDown)
{
    player.setVelocityX(-160);

    player.anims.play('left', true);
}
else if (cursors.right.isDown)
{
    player.setVelocityX(160);

    player.anims.play('right', true);
}
else
{
    player.setVelocityX(0);

    player.anims.play('turn');
}

if (cursors.up.isDown && player.body.touching.down)
{
    player.setVelocityY(-330);
}
```

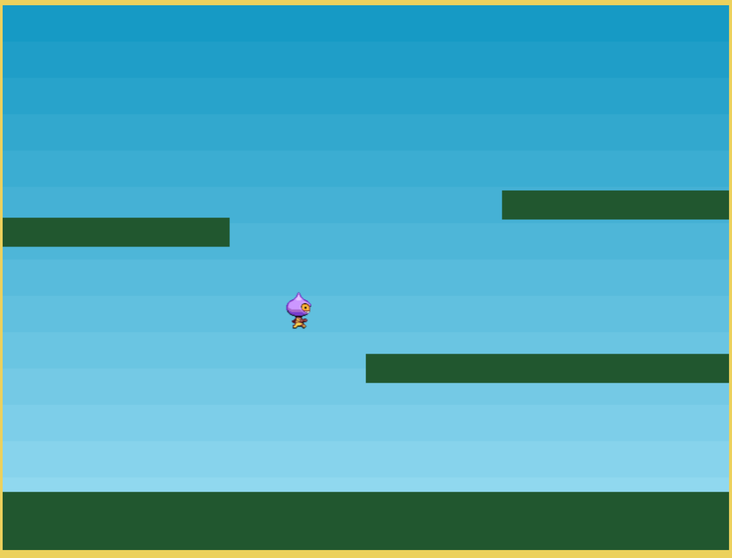
The first thing it does is check to see if the left key is being held down. If it is we apply a negative horizontal velocity and start the 'left' running animation. If they are holding down 'right' instead we literally do the opposite. By clearing the velocity and setting it in this manner, every frame, it creates a 'stop-start' style of movement.

The player sprite will move only when a key is held down and stop immediately they are not. Phaser also allows you to create more complex motions, with momentum and acceleration, but this gives us the effect we need for this game. The final part of the key check sets the animation to 'turn' and zero the horizontal velocity if no key is held down.

Jump to it

The final part of the code adds the ability to jump. The up cursor is our jump key and we test if that is down. However we also test if the player is touching the floor, otherwise they could jump while in mid-air.

If both of these conditions are met we apply a vertical velocity of 330 px/sec sq. The player will fall to the ground automatically because of gravity. With the controls in place we now have a game world we can explore. Load up part7.html and have a play. Try tweaking values like the 330 for the jump to lower and higher values to see the effect it will have.



Part 8 - Stardust

It's time to give our little game a purpose. Let's drop a sprinkling of stars into the scene and allow the player to collect them. To achieve this we'll create a new Group called 'stars' and populate it. In our create function we add the following code:

```
stars = this.physics.add.group({
    key: 'star',
    repeat: 11,
    setXY: { x: 12, y: 0, stepX: 70 }
});

stars.children.iterate(function (child) {

    child.setBounceY(Phaser.Math.FloatBetween(0.4, 0.8));

});
```

The process is similar to when we created the platforms Group. As we need the stars to move and bounce we create a dynamic physics group instead of a static one.

Groups are able to take configuration objects to aid in their setup. In this case the group configuration object has 3 parts: First it sets the texture key to be the star image. This means that any children created as a result of the config object will all be given the star texture by default. Then it sets the repeat value to be 11. Because it creates 1 child automatically, repeating 11 times means we'll get 12 in total, which is just what we need for our game.

The final part is `setXY` - this is used to set the position of the 12 children the Group creates. Each child will be placed starting at x: 12, y: 0 and with an x step of 70. This means that the first child will be positioned at 12 x 0, the second one is 70 pixels on from that at 82 x 0, the third one is at 152 x 0, and so on. The 'step' values are a really handy way of spacing out a Groups children during creation. The value of 70 is chosen because it means all 12 children will be perfectly spaced out across the screen.

The next piece of code iterates all children in the Group and gives them a random Y bounce value between 0.4 and 0.8. The bounce range is between 0, no bounce at all, and 1, a full bounce. Because the stars are all spawned at y 0 gravity is going to pull them down until they collide with the platforms or ground. The bounce value means they'll randomly bounce back up again until finally settling to rest.

If we were to run the code like it is now the stars would fall through the bottom of the game and out of sight. To stop that we need to check for their collision against the platforms. We can use another Collider object to do this:

```
this.physics.add.collider(stars, platforms);
```

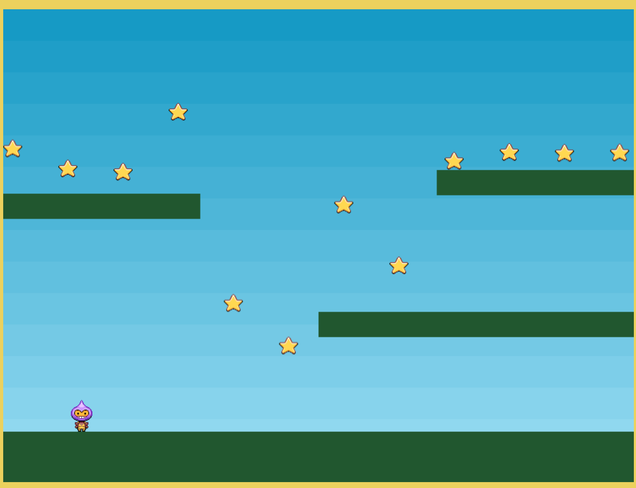
As well as doing this we will also check to see if the player overlaps with a star or not:

```
this.physics.add.overlap(player, stars, collectStar, null, this);
```

This tells Phaser to check for an overlap between the player and any star in the stars Group. If found then they are passed to the 'collectStar' function:

```
function collectStar (player, star)
{
    star.disableBody(true, true);
}
```

Quite simply the star has its physics body disabled and its parent Game Object is made inactive and invisible, which removes it from display. Running the game now gives us a player that can dash about, jump, bounce off the platforms and collecting the stars that fall from above.



Part 9 - A score to settle

There are two final touches we're going to add to our game: an enemy to avoid that can kill the player, and a score when you collect the stars. First, the score.

To do this we'll make use of a Text Game Object. Here we create two new variables, one to hold the actual score and the text object itself:

```
var score = 0;
var scoreText;
```

The `scoreText` is set-up in the `create` function:

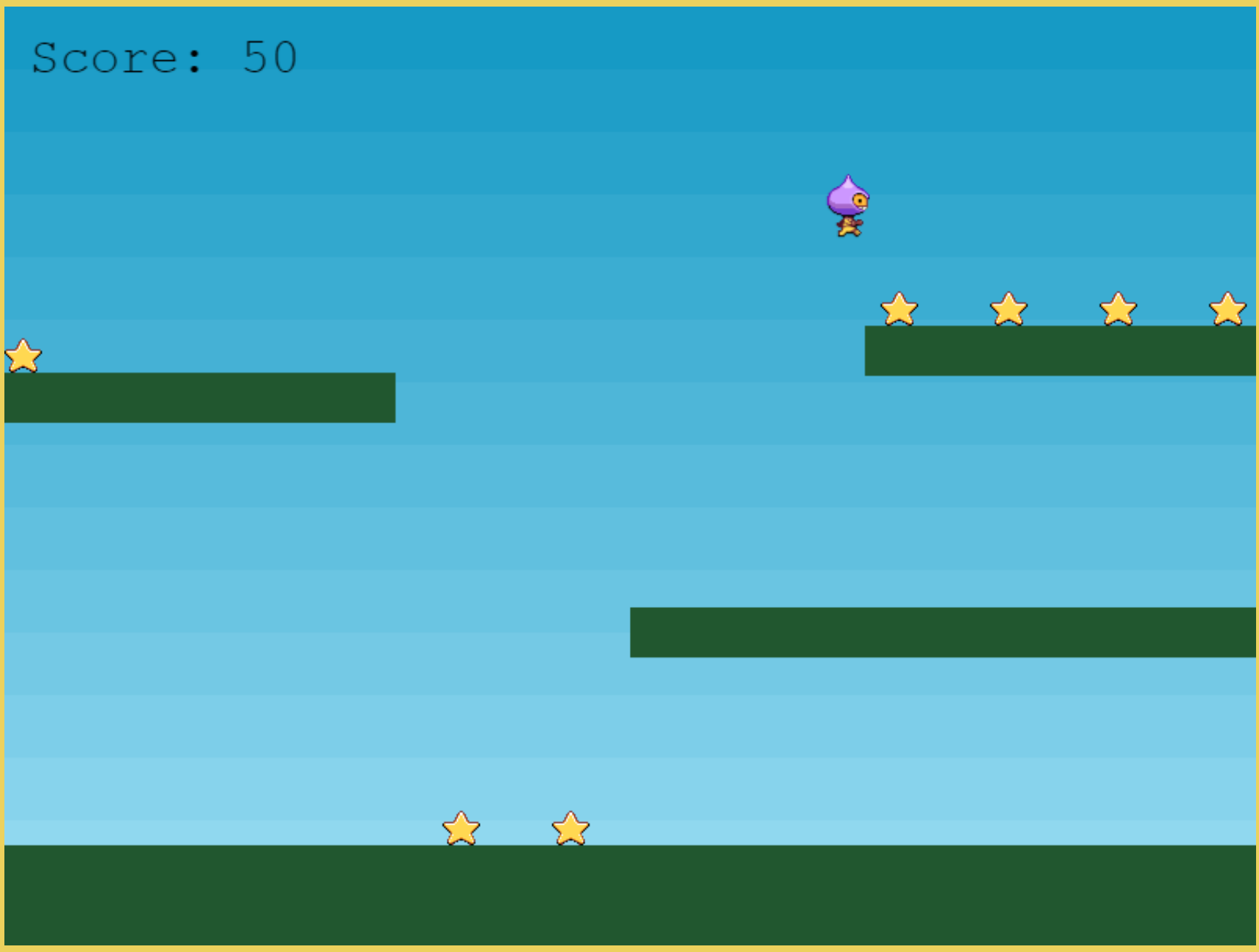
```
scoreText = this.add.text(16, 16, 'score: 0', {
fontSize: '32px', fill: '#000' });
```

16 x 16 is the coordinate to display the text at. 'score: 0' is the default string to display and the object that follows contains a font size and fill color. By not specifying which font we'll actually use the Phaser default, which is Courier.

Next we need to modify the `collectStar` function so that when the player picks-up a star their score increases and the text is updated to reflect this:

```
function collectStar (player, star)
{
    star.disableBody(true, true);
    score += 10;
    scoreText.setText('Score: ' + score);
}
```

So 10 points are added for every star and the `scoreText` is updated to show this new total. If you reload the page in browser you will see the stars fall and the score increase as you collect them.



In the final part we'll add some baddies.

Part 10 - Bouncing Bombs

The idea is this: When you collect all the stars the first time it will release a bouncing bomb. The bomb will just randomly bounce around the level and if you collide with it, you die. All of the stars will respawn so you can collect them again, and if you do, it will release another bomb. This will give the player a challenge: get as high a score as possible without dying.

The first thing we need is a Group for the bombs and a couple of Colliders:

```
bombs = this.physics.add.group();

this.physics.add.collider(bombs, platforms);

this.physics.add.collider(player, bombs, hitBomb, null, this);
```

The bombs will of course bounce off the platforms, and if the player hits them we'll call the **`hitBomb`** function. All that will do is stop the game and turn the player red:

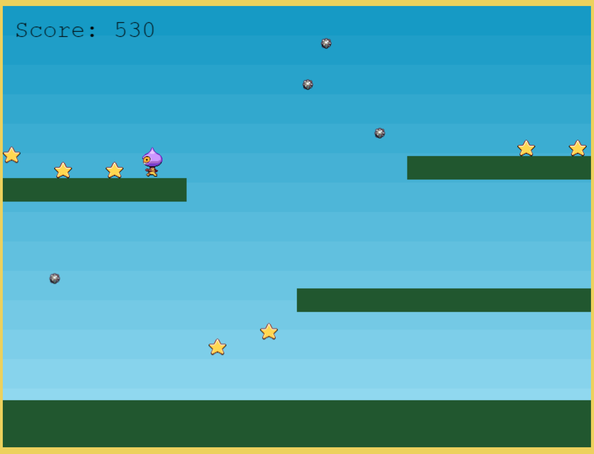
```
function hitBomb (player, bomb)
{
    this.physics.pause();

    player.setTint(0xff0000);

    player.anims.play('turn');

    gameOver = true;
}
```

So far, so good, but we need to release a bomb. To do that we modify the **`collectStar`** function:

```
function collectStar (player, star)
{
    star.disableBody(true, true);

    score += 10;
    scoreText.setText('Score: ' + score);

    if (stars.countActive(true) === 0)
    {
        stars.children.iterate(function (child) {

            child.enableBody(true, child.x, 0, true, true);

        });

        var x = (player.x < 400) ? Phaser.Math.Between(400, 800) :
Phaser.Math.Between(0, 400);

        var bomb = bombs.create(x, 16, 'bomb');
        bomb.setBounce(1);
        bomb.setCollideWorldBounds(true);
        bomb.setVelocity(Phaser.Math.Between(-200, 200), 20);

    }
}
```

We use a Group method called **`countActive`** to see how many stars are left alive. If it's none then the player has collected them all, so we use the iterate function to re-enable all of the stars and reset their y position to zero. This will make all of the stars drop from the top of the screen again.

The next part of the code creates a bomb. First we pick a random x coordinate for it, always on the opposite side of the screen to the player, just to give them a chance. Then the bomb is created, it's set to collide with the world, bounce and have a random velocity.



Check if everything is ok, and if so, commit and push it!