

CS 50 Web Design

APCSP Module 2: Internet



Unit 3: JavaScript

LECTURE 13: PROJECT DESIGN USING JAVASCRIPT

DR. ERIC CHOU

IEEE SENIOR MEMBER



Objectives

Develop a JavaScript Project

1. Story line, scene, and scene transitions
2. Scene, sprite, actor, and other graphics design
3. Animation (Game loop)
4. Human-computer interaction
5. Model-View-Controller Design (MVC)

Interactive Programming (Timing and Control)

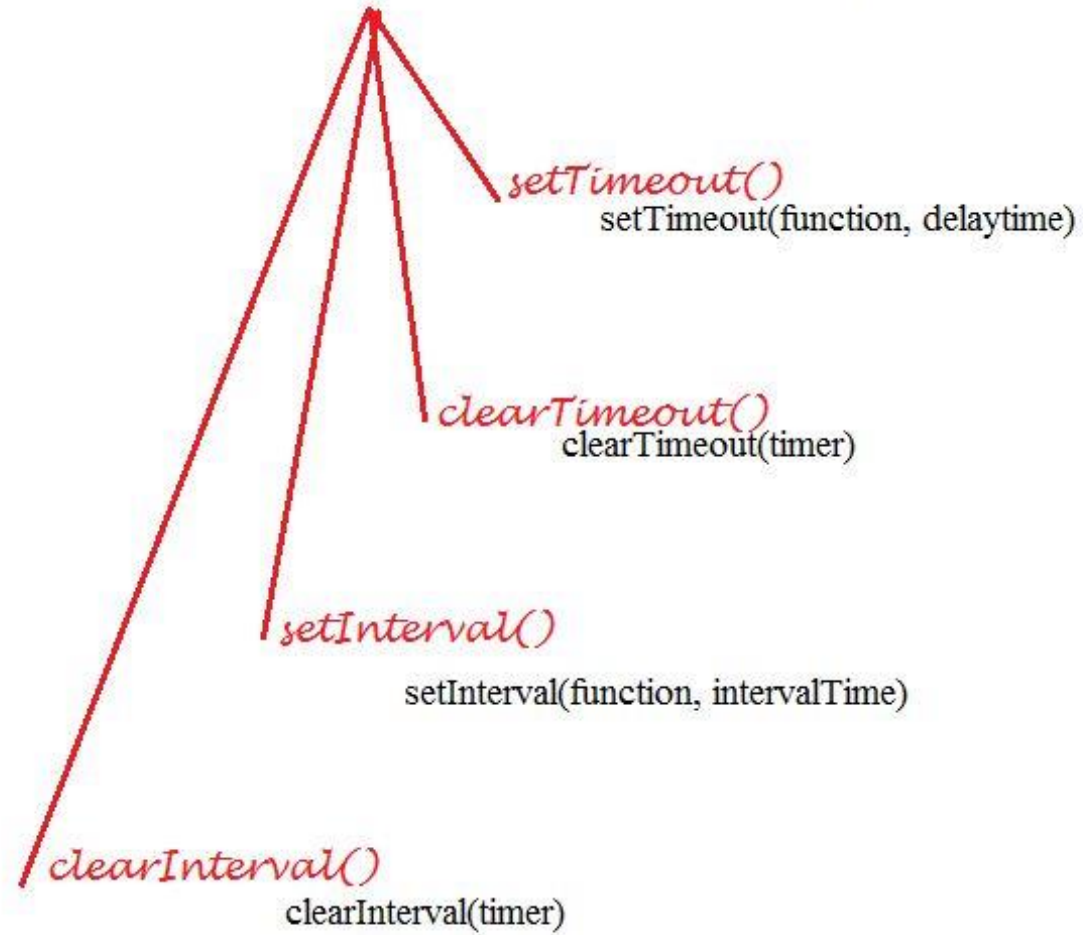
SECTION 1



Overview: Interactive Programming

- Until now, the **JavaScript** code on our web pages has run as soon as the page is loaded, pausing only if we include a call to a function like `alert` or `confirm`. But we don't always necessarily want all of our code to run as soon as the page loads — what if we want some code to run after a delay or in response to something the user does?
- In this chapter, we'll look at different ways of modifying when our code is run. Programming in this way is called **interactive programming**. This will let us create interactive web pages that change over time and respond to actions by the user.

Timer Method in JavaScript



Time Out

SECTION 2

setTimeout() {}

JAVASCRIPT

SET TIMEOUT FUNCTION

Delaying Code with setTimeout

- Instead of having JavaScript execute a function immediately, you can tell it to execute a function after a certain period of time. Delaying a function like this is called setting a timeout. To set a timeout in JavaScript, we use the function **setTimeout**.
- This function takes two arguments (as shown in Figure 10- 1): the function to call after the time has elapsed and the amount of time to wait (in milliseconds).

The function to call after
timeout milliseconds have passed



```
setTimeout(func, timeout)
```



The number of milliseconds to wait
before calling the function

Figure 10-1. The arguments for setTimeout



Delaying Code with setTimeout

- The following listing shows how we could use setTimeout to display an alert dialog.

```
❶ var timeUp = function () {  
  alert("Time's up!");  
};  
❷ setTimeout(timeUp, 3000);  
1
```

- At ❶ we create the function timeUp, which opens an alert dialog that displays the text "Time's up!". At ❷ we call setTimeout with two arguments: the function we want to call (timeUp) and the number of milliseconds (3000) to wait before calling that function. We're essentially saying, "Wait 3 seconds and then call timeUp."
- When setTimeout(timeUp, 3000) is first called, nothing happens, but after 3 seconds timeUp is called and the alert dialog pops up.

Delaying Code with setTimeout

- Notice that calling **setTimeout** returns 1. This return value is called the ***timeout ID***. The timeout ID is a number that's used to identify this particular timeout (that is, this particular delayed function call). The actual number returned could be any number, since it's just an identifier. Call **setTimeout** again, and it should return a different timeout ID, as shown here:

```
setTimeout(timeUp, 5000);  
2
```

- You can use this timeout ID with the **clearTimeout** function to cancel that specific timeout. We'll look at that next.

Canceling a Timeout

- Once you've called **setTimeout** to set up a delayed function call, you may find that you don't actually want to call that function after all. For example, if you set an alarm to remind you to do your homework, but you end up doing your homework early, you'd want to cancel that alarm.
- To cancel a timeout, use the function **clearTimeout** on the timeout ID returned by **setTimeout**. For example, say we create a "do your homework" alarm like this:

Canceling a Timeout

```
var doHomeworkAlarm = function () {  
  alert("Hey! You need to do your homework!");  
};  
❶ var timeoutId = setTimeout(doHomeworkAlarm, 60000);
```

The function **doHomeworkAlarm** pops up an alert dialog telling you to do your homework. When we call **setTimeout(doHomeworkAlarm, 60000)** we're telling JavaScript to execute that function after 60,000 milliseconds (or 60 seconds) has passed. At ❶ we make this call to **setTimeout** and save the timeout ID in a new variable called **timeoutId**.

To cancel the timeout, pass the timeout ID to the **clearTimeout** function like this:

```
clearTimeout(timeoutId) ;
```

Now **setTimeout** won't call the **doHomeworkAlarm** function after all.

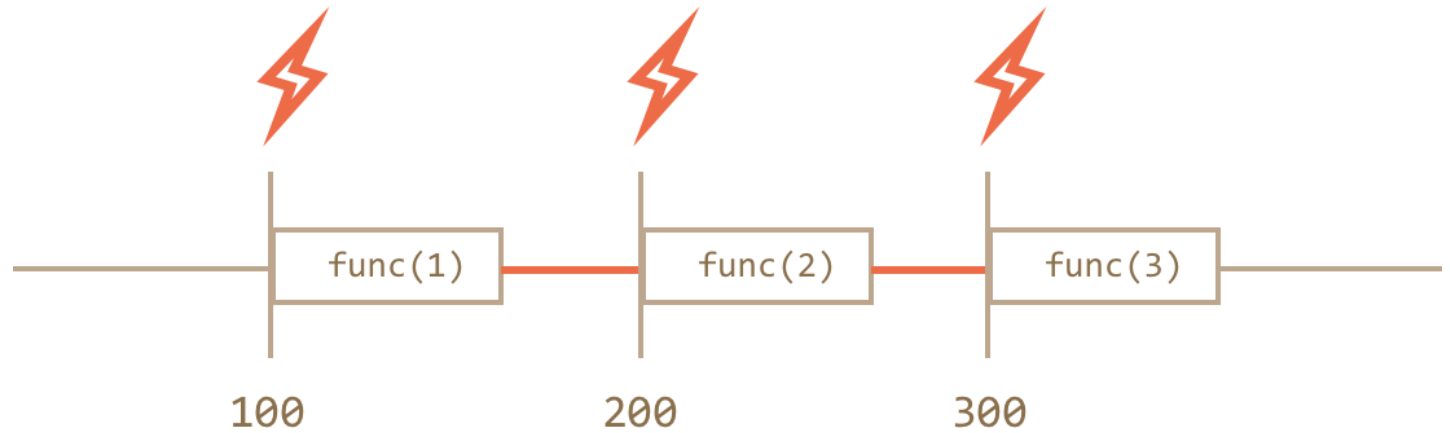


In-Class Demonstration Program

SET LONG TIMER

Flashing Text Demo Program: flashing.html

- The time-out points will fire some function.
- We repeatedly fire `hide()` and `show()` will create flashing effect for text display.




Flashing Text Project 2:

Demo Program: [flashing2.html](#)

- The JavaScript section uses `addEventListener` instead of default event handler attachment by `onclick`.
- When **`addEventListener`** is used, it is safer to put the JavaScript section to the end of `<body>` section. Otherwise, the **`addEventListener`** may not work.
- An Object is return by the `blink` function. The returned object is actually the blinking controller. The blinking object is a blinking controller. There are two functions: `start()` and `stop()`. These two functions are added to the Event Handlers of the buttons.


```
cycle_time = 1000;
function blink(element, time) {
  function loop(){
    element.style.visibility = "hidden";
    setTimeout(function () {
      element.style.visibility = "visible";
    }, time);
    timer = setTimeout(function () {
      loop();
    }, time * 2);
    cleared = false;
  }
}
```



Recursion

```
function blink(element, time) {
```

```
  function loop(){
    element.style.visibility = "hidden";
    setTimeout(function () {
      element.style.visibility = "visible";
    }, time);
    timer = setTimeout(function () {
      loop();
    }, time * 2);
    cleared = false;
  }
```

```
var timer, cleared = true;
```

```
// expose methods
```

```
return {
```

```
  start: function() {
    if (cleared) loop();
  },
```

```
  stop: function() {
    clearTimeout(timer);
    cleared = true;
  }
};
```

```
}
```

Emergency

start blinking

stop blinking

```
<div id="blink" style="color:red; font-size:48px">Emergency</div>
<button id="start">start blinking</button><br />
<button id="stop">stop blinking</button>
```


```
var blinking = blink(document.getElementById("blink"),
  cycle_time/2);
```

```
document.getElementById("start").addEventListener("click",
  function(){
    blinking.start();
  });
```

```
document.getElementById("stop").addEventListener("click",
  function(){
    blinking.stop();
  });
```

Intervals

SECTION 3



Calling Code Multiple Times with `setInterval`

- The `setInterval` function is like **`setTimeout`**, except that it **repeatedly** calls the supplied function after regular pauses, or intervals. For example, if you wanted to update a clock display using JavaScript, you could use **`setInterval`** to call an update function every second. You call **`setInterval`** with two arguments: the function you want to call and the length of the interval (in milliseconds), as shown in Figure 10-2.

Calling Code Multiple Times with setInterval

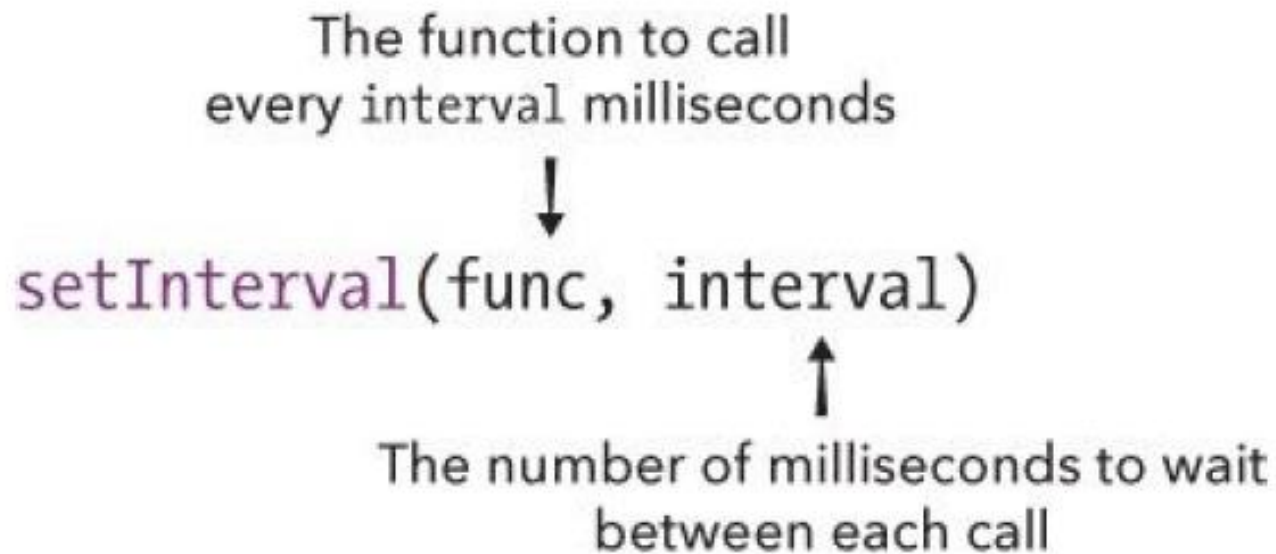


Figure 10-2. The arguments for setInterval

Calling Code Multiple Times with setInterval

Here's how we could write a message to the console every second:

```
❶ var counter = 1;
❷ var printMessage = function () {
  console.log("You have been staring at your console for "
+ counter
+ " seconds");
❸ counter++;
};
❹ var intervalId = setInterval(printMessage, 1000);
You have been staring at your console for 1 seconds
You have been staring at your console for 2 seconds
You have been staring at your console for 3 seconds
You have been staring at your console for 4 seconds
You have been staring at your console for 5 seconds
You have been staring at your console for 6 seconds
❺ clearInterval(intervalId);
```

Calling Code Multiple Times with setInterval

At ❶ we create a new variable called `counter` and set it to 1. We'll be using this variable to keep track of the number of seconds you've been looking at your console.

At ❷ we create a function called `printMessage`. This function does two things. First, it prints out a message telling you how long you have been staring at your console. Then, at ❸, it increments the `counter` variable.

Next, at ❹, we call `setInterval`, passing the `printMessage` function and the number 1000. Calling `setInterval` like this means "call `printMessage` every 1,000 milliseconds." Just as `setTimeout` returns a timeout ID, `setInterval` returns an *interval ID*, which we save in the variable `intervalId`.

We can use this interval ID to tell JavaScript to stop executing the `printMessage` function. This is what we do at ❺, using the `clearInterval` function.

```
1▼ <html>
2▼ <body>
3▼ <script>
4  var counter = 1;
5▼ var printMessage = function () {
6    document.write("You have been staring at your console for " +
7                  counter + " seconds"+"<br>");
8    counter++;
9    if (counter > 20) clearInterval(intervalId);
10 };
11 var intervalId = setInterval(printMessage, 1000);
12
13 </script>
14 </body>
15 </html>
```

Demo Program: multiple.html



In-Class Demonstration Program

MULTIPLE FUNCTION CALL

A decorative graphic on the left side of the slide, featuring a cluster of 3D triangles in various colors (blue, red, pink, green, yellow, orange) arranged in a geometric pattern, creating a sense of depth and movement.

JavaScript Game Development Overview

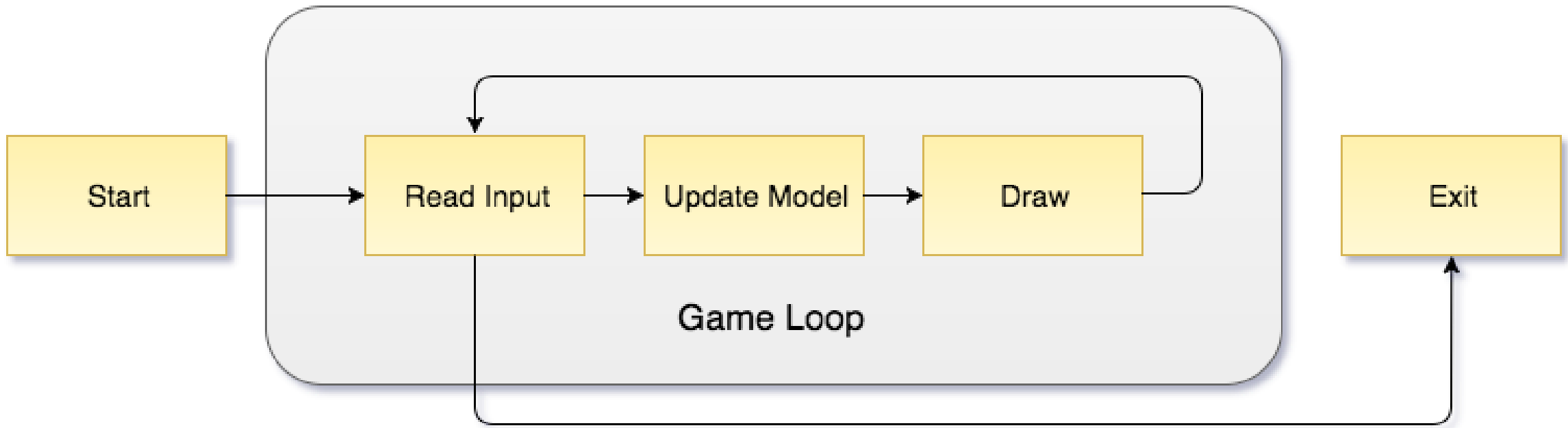
- Web is a convenient platform for a quick start in game development, especially for those of you who are familiar with JavaScript language. This is a feature-rich platform in content rendering and input processing from different resources.
- First steps in game development for Web:
 - Sort out the game loop and rendering definitions.
 - Learn to process user input.
 - Create the main scene prototype of the game.
 - Add the rest of the game scenes.



Game Loop

- Game cycle is a heart of the game, where user input and game logic processing occur along with current game state rendering. Schematically, game loop can be described in the following way:
- As a code, the simplest implementation of the game cycle in JavaScript may look like this:

```
// game loop
setInterval(() => {
    update();
    render();
}, 1000 / 60);
```



Use JavaScript setInterval for Game Loop

Game Loop

- Update() function is responsible for the game process logic and updating of game state, depending on the user input. Wherein, it absolutely doesn't matter, with the help of which technologies the rendering itself is happening (Canvas, DOM, SVG, console etc.).



Game Loop

Please note that browser environment imposes certain restrictions on the work of this code:

- Stable timer interval is not guaranteed, which means that the game process may occur with **different** speed.
- The timer in inactive browser tabs may be stopped or repeatedly launched, when the tab is activated, which may lead to strange behavior of the game.
- `setInterval` is already outdated for such tasks. Today it is recommended to use **`requestAnimationFrame`** method, because it allows to achieve better productivity and decrease energy consumption.
- Please check for our Game Design Chapter.

Animation

SECTION 4

Animating Elements with setInterval

- As it turns out, we can use **setInterval** to animate elements in a browser. Basically, we need to create a function that moves an element by a small amount, and then pass that function to **setInterval** with a short interval time.
- If we make the movements small enough and the interval short enough, the animation will look very smooth.

Animating Elements with setInterval

Let's animate the position of some text in an HTML document by moving the text horizontally in the browser window. Create a document called *interactive.html*, and fill it with this HTML:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Interactive programming</title>
  </head>
  <body>
    <h1 id="heading">Hello world!</h1>
    <script
      src="https://code.jquery.com/jquery-
        2.1.0.js">
    </script>
    <script>
      // We'll fill this in next
    </script>
  </body>
</html>
```

Animating Elements with setInterval

Now let's look at the JavaScript. As always, put your code inside the `<script>` tags of the HTML document.

```
❶ var leftOffset = 0;
❷ var moveHeading = function () {
❸     $("#heading").offset({ left: leftOffset });
❹     leftOffset++;
❺     if (leftOffset > 200) {
        leftOffset = 0;
    }
};
❻ setInterval(moveHeading, 30);
```

Animating Elements with setInterval

- When you open this page, you should see the heading element gradually move across the screen until it travels 200 pixels; at that point, it will jump back to the beginning and start again. Let's see how this works.
- At ❶ we create the variable **leftOffset**, which we'll use later to position our Hello world! heading. It starts with a value of 0, which means the heading will start on the far left side of the page.
- Next, at ❷, we create the function **moveHeading**, which we'll call later with setInterval. Inside the **moveHeading** function, at ❸, we use `$("#heading")` to select the element with the id of "heading" (our h1 element) and use the offset method to set the left offset of the heading — that is, how far it is from the left side of the screen.



Animating Elements with `setInterval`

- The `offset` method takes an object that can contain a `left` property, which sets the left offset of the element, or a `top` property, which sets the top offset of the element. In this example we use the `left` property and set it to our `leftOffset` variable. If we wanted a static offset (that is, an offset that doesn't change), we could set the property to a numeric value. For example, calling `$("#heading").offset({ left: 100 })` would place the heading element 100 pixels from the left side of the page.
- At ④ we increment the `leftOffset` variable by 1. To make sure the heading doesn't move too far, at ⑤ we check to see if `leftOffset` is greater than 200, and if it is, we reset it to 0. Finally, at ⑥ we call `setInterval`, and for its arguments we pass in the function `moveHeading` and the number 30 (for 30 milliseconds).



Animating Elements with setInterval

- This code calls the **moveHeading** function every 30 milliseconds, or about 33 times every second. Each time **moveHeading** is called, the **leftOffset** variable is incremented, and the value of this variable is used to set the position of the heading element.
- Because the function is constantly being called and **leftOffset** is incremented by 1 each time, the heading gradually moves across the screen by 1 pixel every 30 milliseconds.



Demonstration Program

ANIMATION BE SETINTERVAL
(ANIMATION0.HTML)

Interaction

SECTION 5

Responding to User Actions

- As you've seen, one way to control when code is run is with the **functions** **setTimeout** and **setInterval**, which run a function once a fixed amount of time has passed.
- Another way is to run code only when a user performs certain actions, such as clicking, typing, or even just moving the mouse.
- This will let users interact with your web page so that your page responds according to what they do.

Responding to User Actions

- In a browser, every time you perform an action such as clicking, typing, or moving your mouse, something called an *event* is triggered. An event is the browser's way of saying, "This thing happened!"
- You can listen to these events by adding an *event handler* to the element where the event happened.
- Adding an event handler is your way of telling JavaScript, "If this event happens on this element, call this function." For example, if you want a function to be called when the user clicks a heading element, you could add a click event handler to the heading element. We'll look at how to do that next.

Responding to Clicks

- When a user clicks an element in the browser, this triggers a *click event*. jQuery makes it easy to add a handler for a click event. Open the *interactive.html* document you created earlier, use **File ▶ Save As** to save it as *clicks.html*, and replace its second script element with this code:

```
❶ var clickHandler = function (event) {  
  ❷   console.log("Click! " + event.pageX + " " + event.pageY);  
    };  
❸ $("h1").click(clickHandler);
```

Responding to Clicks

- At ❶ we create the function **clickHandler** with the single argument event. When this function is called, the event argument will be an object holding information about the click event, such as the location of the click. At ❷, inside the handler function, we use `console.log` to output the properties **pageX** and **pageY** from the event object. These properties tell us the event's x- and y-coordinates — in other words, they say where on the page the click occurred.
- Finally, at ❸ we activate the click handler. The **code** `$("#h1")` selects the h1 element, and calling `$("#h1").click(clickHandler)` means “When there is a click on the h1 element, call the **clickHandler** function and pass it the event object.” In this case, the click handler retrieves information from the event object to output the x- and y-coordinates of the click location.

Responding to Clicks

- Reload your modified page in your browser and click the heading element. Each time you click the heading, a new line should be output to the console, as shown in the following listing. Each line shows two numbers: the x- and y-coordinates of the clicked location.

```
Click! 88 43  
Click! 63 53  
Click! 24 53  
Click! 121 46  
Click! 93 55  
Click! 103 48
```



Demonstration Program

CLICKS.HTML

Browser Coordinates

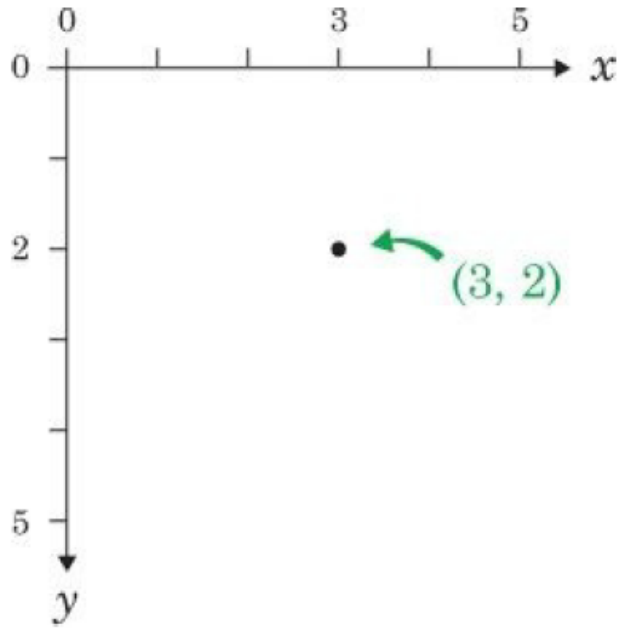


Figure 10-3. Coordinates in the browser, showing a click at the coordinate (3, 2)

- In the web browser and in most programming and graphics environments, the 0 position of the x- and y-coordinates is at the top-left corner of the screen. As the x-coordinate increases, you move right across the page, and as the y-coordinate increases, you move down the page (see Figure 10-3).

The mousemove Event

- The `mousemove` event is triggered every time the mouse moves. To try it out, create a file called *mousemove.html* and enter this code:

```
<!DOCTYPE html>
<html>
<head>
<title>Mousemove</title>
</head>
<body>
  <h1 id="heading">Hello world!</h1>
  <script src="https://code.jquery.com/jquery-2.1.0.js"></script>
  <script>
    ❶ $("html").mousemove(function (event) {
    ❷ $("#heading").offset({
      left: event.pageX,
      top: event.pageY
    });
    });
  </script>
</body>
</html>
```

The mousemove Event

At ❶ we add a handler for the `mousemove` event using `$("#html").mousemove(handler)`. In this case, the *handler* is the entire function that appears after `mousemove` and before `</script>`. We use `$("#html")` to select the `html` element so that the handler is triggered by mouse movements that occur anywhere on the page. The function that we pass into the parentheses after `mousemove` will be called every time the user moves the mouse.

The mousemove Event

- In this example, instead of creating the event handler separately and passing the function name to the `mousemove` method (as we did with our `clickHandler` function earlier), we're passing the handler function directly to the `mousemove` method.
- This is a very common way of writing event handlers, so it's good to be familiar with this type of syntax.



The mousemove Event

- At ❷, inside the event handler function, we select the heading element and call the offset method on it. As I mentioned before, the object passed to offset can have left and top properties. In this case, we set the left property to event.pageX and the top property to event.pageY.
- Now, every time the mouse moves, the heading will move to that location. In other words, wherever you move the mouse, the heading follows it!



Demonstration Program

MOUSEMOVE.HTML

```
1  <!DOCTYPE html>
2  ▼ <html>
3  ▼   <head>
4      <title>Mousemove</title>
5      </head>
6  ▼   <body>
7      <h1 id="heading">Hello world!</h1>
8      <script src="https://code.jquery.com/jquery-3.4.1.js"></script>
9  ▼   <script>
10 ▼       $("html").mousemove(function (event) {
11 ▼           $("#heading").offset({
12               left: event.pageX,
13               top: event.pageY
14           });
15       });
16   </script>
17   </body>
18 </html>
```

A Game Project Development

SECTION 6

A close-up photograph of two dice, one yellow and one blue, stacked on top of each other. They are resting on a surface that appears to be a treasure map with various colored squares and patterns. The background is blurred, showing more of the map and some other objects.

Chapter 11. Find the Buried Treasure!

- Let's put what we've learned so far to good use and make a **game**! The aim of this game is to find the hidden treasure.
- In this game, the web page will display a treasure map. Inside that map, the program will pick a single pixel location, which represents where the hidden treasure is buried. Every time the player clicks the map, the web page will tell them how close to the treasure they are.
- When they click the location of the treasure (or very close to it), the game congratulates them on finding the treasure and says how many clicks it took to find it. Figure 11-1 shows what the game will look like after a player clicks the map.

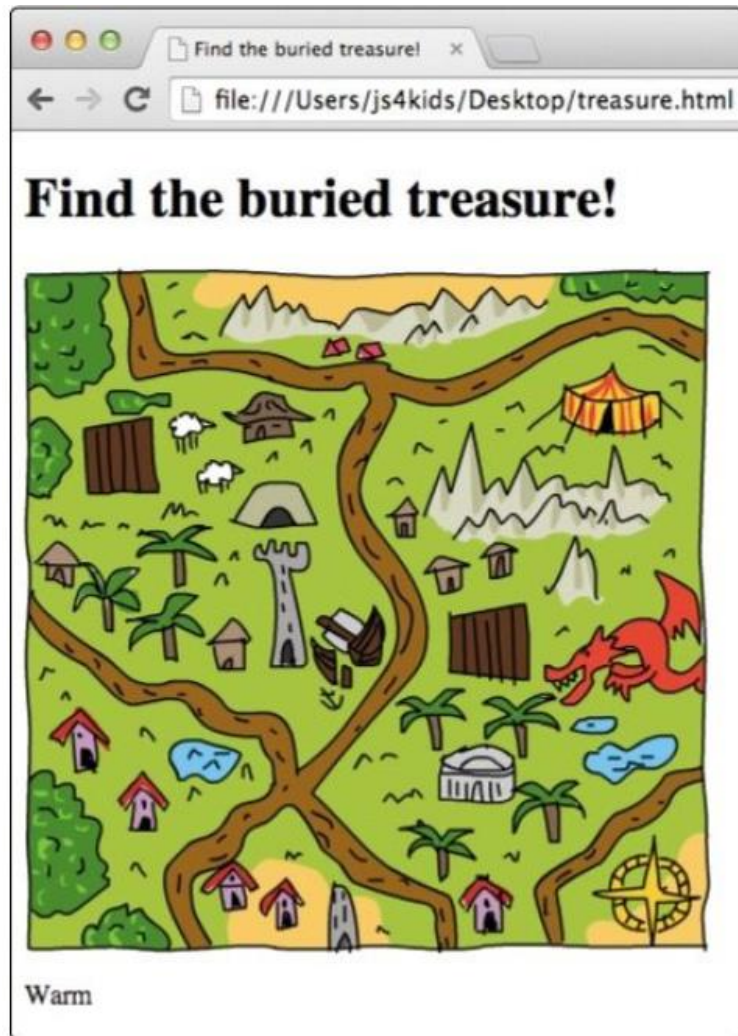


Figure 11-1. The buried treasure game

Game Design

SECTION 7

Designing the Game

Before we start writing the code, let's break down the overall structure of this game. Here is a list of steps we need to take to set up the game so it can respond accordingly when a player clicks the treasure map.

1. Create a **web page** with an image (the treasure map) and a place to display messages to the player.
2. Pick a random spot on the map picture to hide the treasure.
3. Create a click handler. Each time the player clicks the map, the click handler will do the following:
 - a. Add 1 to a click counter.
 - b. Calculate how far the click location is from the treasure location.
 - c. Display a message on the web page to tell the player whether they're hot or cold.
 - d. Congratulate the player if they click on the treasure or very close to it, and say how many clicks it took to find the treasure.

I'll show you how to implement each of these features in the game, and then we'll go through the full code.

Web Page

SECTION 8

Creating the Web Page with HTML

Let's look at the HTML for the game. We'll use a new element called `img` for the treasure map and add a `p` element where we can display messages to the player. Enter the following code into a new file called ***treasure.html***.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Find the buried treasure!</title>
  </head>
  <body>
    <h1 id="heading">Find the buried treasure!</h1>
    ① 
    ③ <p id="distance"></p>
    <script src="https://code.jquery.com/jquery-
      3.4.1.js"></script>

    <script>
      // Game code goes here
    </script>
  </body>
</html>
```



Demonstration Program

TREASUREO.HTML

Creating the Web Page with HTML

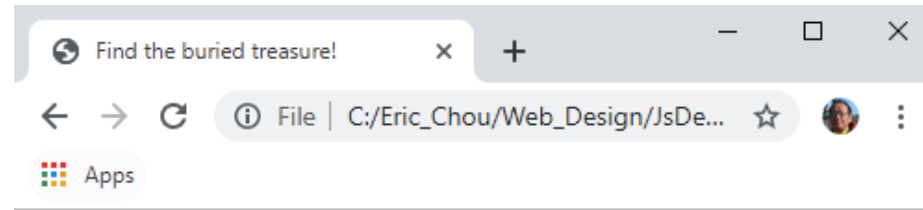
- The `img` element is used to include images in HTML documents. Unlike the other HTML elements we've looked at, `img` doesn't use a closing tag.
- All you need is an opening tag, which, like other HTML tags, can contain various attributes. At ❶ we've added an `img` element with an `id` of "map".
- We set the width and height of this element using the `width` and `height` attributes, which are both set to 400. This means our image will be 400 pixels tall and 400 pixels wide.

Creating the Web Page with HTML

- To tell the document which image we want to display, we use the `src` attribute to include the web address of the image at ❷. In this case, we're linking to an image called `treasuremap.png` on the No Starch Press website.
- Following the **`img`** element is an empty `p` element at ❸, which we give an id of `"distance"`. We'll add text to this element by using JavaScript to tell the player how close they are to the treasure.

```
1 ▼ <html>
2 ▼ <head>
3   <title>Find the buried treasure!</title>
4   </head>
5 ▼ <body>
6     <h1 id="heading">Find the buried treasure!</h1>
7     
9     <p id="distance"></p>
10    <script src="https://code.jquery.com/jquery-3.4.1.js"></script>
11 ▼ <script>
12     // Game code goes here
13 </script>
14 </body>
15 </html>
```

Paragraph <p> as message panel.



Find the buried treasure!



Random Treasure

SECTION 9

Picking a Random Treasure Location

- Now let's build the JavaScript for our game. First we need to pick a random location for the hidden treasure inside the treasure map image.
- Since the dimensions of the map are 400 by 400 pixels, the coordinates of the top-left pixel will be { x: 0, y: 0 }, and the bottom-right pixel will be { x: 399, y: 399 }.



Picking Random Numbers

- To set a random coordinate point within the treasure map, we pick a random number between 0 and 399 for the x value and a random number between 0 and 399 for the y value.
- To generate these random values, we'll write a function that takes a size argument as input and picks a random number from 0 up to (but not including) size:

```
var getRandomNumber = function (size) {  
    return Math.floor(Math.random() * size);  
};
```

Picking Random Numbers

- This code is similar to the code we've used to pick random words in earlier chapters. We generate a random number between 0 and 1 using `Math.random`, multiply that by the size argument, and then use `Math.floor` to round that number down to a whole number.
- Then we output the result as the return value of the function. Calling `getRandomNumber(400)` will return a random number from 0 to 399, which is just what we need!

Setting the Treasure Coordinates

- Now let's use the getRandomNumber function to set the treasure coordinates:

```
❶ var width = 400;  
var height = 400;  
❷ var target = {  
  x: getRandomNumber(width),  
  y: getRandomNumber(height)  
};
```

- The section of code at ❶ sets the width and height variables, which represent the width and height of the img element that we're using as a treasure map. At ❷ we create an object called target, which has two properties, x and y, that represent the coordinates of the buried treasure.
- The x and y properties are both set by getRandomNumber. Each time we run this code, we get a new random location on the map, and the chosen coordinates will be saved in the x and y properties of the target variable.

The Click Handler

SECTION 10

The Click Handler

- The click handler is the function that will be called when the player clicks the treasure map. Start building this function with this code:

```
$("#map").click(function (event) {  
    // Click handler code goes here  
});
```

The Click Handler

- First we use `$("#map")` to select the treasure map area (because the `img` element has an id of "map"), and then we go into the click handler function. Each time the player clicks the map, the function body between the curly brackets will be executed. Information about the click is passed into that function body as an object through the event argument.
- This click handler function needs to do quite a bit of work: it has to increment the click counter, calculate how far each click is from the treasure, and display messages. Before we fill in the code for the click handler function, we'll define some variables and create some other functions that will help execute all these steps.

Counting Clicks

- The first thing our click handler needs to do is track the total number of clicks. To set this up, we create a variable called `clicks` at the beginning of the program (outside the click handler) and initialize it to zero:

```
var clicks = 0;
```

- Inside the click handler, we'll include `clicks++` so that we increment `clicks` by 1 each time the player clicks the map.

Calculating the Distance Between the Click and the Treasure

- To figure out whether the player is hot or cold (close to the treasure or far away), we need to measure the distance between where the player clicked and the location of the hidden treasure. To do this, we'll write a function called `getDistance`, like so:

```
var getDistance = function (event, target) {  
    var diffX = event.offsetX - target.x;  
    var diffY = event.offsetY - target.y;  
    return Math.sqrt((diffX * diffX) + (diffY *  
        diffY));  
};
```

Calculating the Distance Between the Click and the Treasure

- The `getDistance` function takes two objects as arguments: `event` and `target`. The `event` object is the object passed to the click handler, and it comes with lots of built-in information about the player's click.
- In particular, it contains two properties called `offsetX` and `offsetY`, which tell us the `x`- and `y`coordinates of the click, and that's exactly the information we need.

Calculating the Distance Between the Click and the Treasure

- Inside the function, the variable `diffX` stores the horizontal distance between the clicked location and the target, which we calculate by subtracting `target.x` (the x-coordinate of the treasure) from `event.offsetX` (the x-coordinate of the click).
- We calculate the vertical distance between the points in the same way, and store the result as `diffY`. Figure 11-2 shows how we would calculate `diffX` and `diffY` for two points.

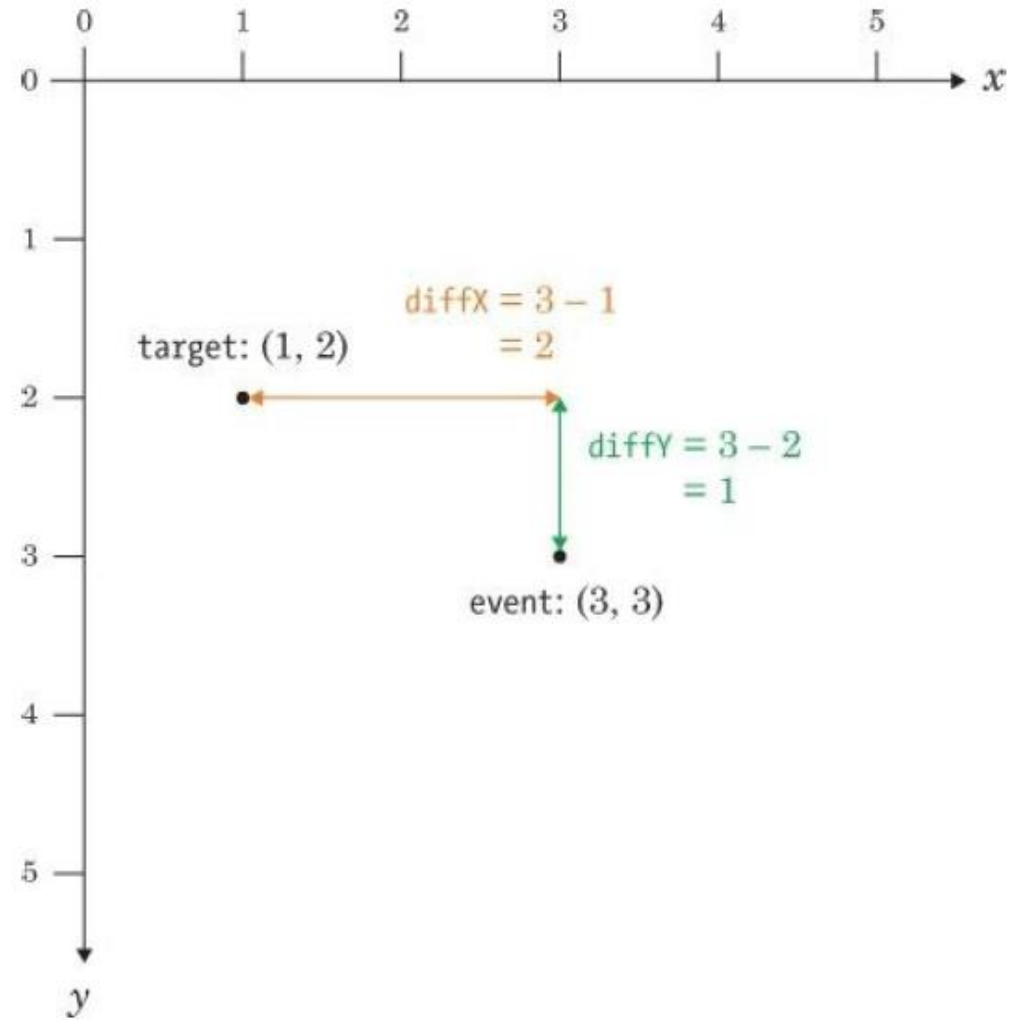


Figure 11-2. Calculating the horizontal and vertical distances between event and target

Using the Pythagorean Theorem

- Next, the `getDistance` function uses the *Pythagorean theorem* to calculate the distance between two points. The Pythagorean theorem says that for a right triangle, where a and b represent the lengths of the two sides bordering the right angle and c represents the length of the diagonal side (the *hypotenuse*), $a^2 + b^2 = c^2$.
- Given the lengths of a and b , we can calculate the length of the hypotenuse by calculating the square root of $a^2 + b^2$.

Using the Pythagorean Theorem

- To calculate the distance between the event and the target, we treat the two points as if they're part of a right triangle, as shown in Figure 11-3. In the `getDistance` function, `diffX` is the length of the horizontal edge of the triangle, and `diffY` is the length of the vertical edge.
- To calculate the distance between the click and the treasure, we need to calculate the length of the hypotenuse, based on the lengths `diffX` and `diffY`. A sample calculation is shown in Figure 11-3.

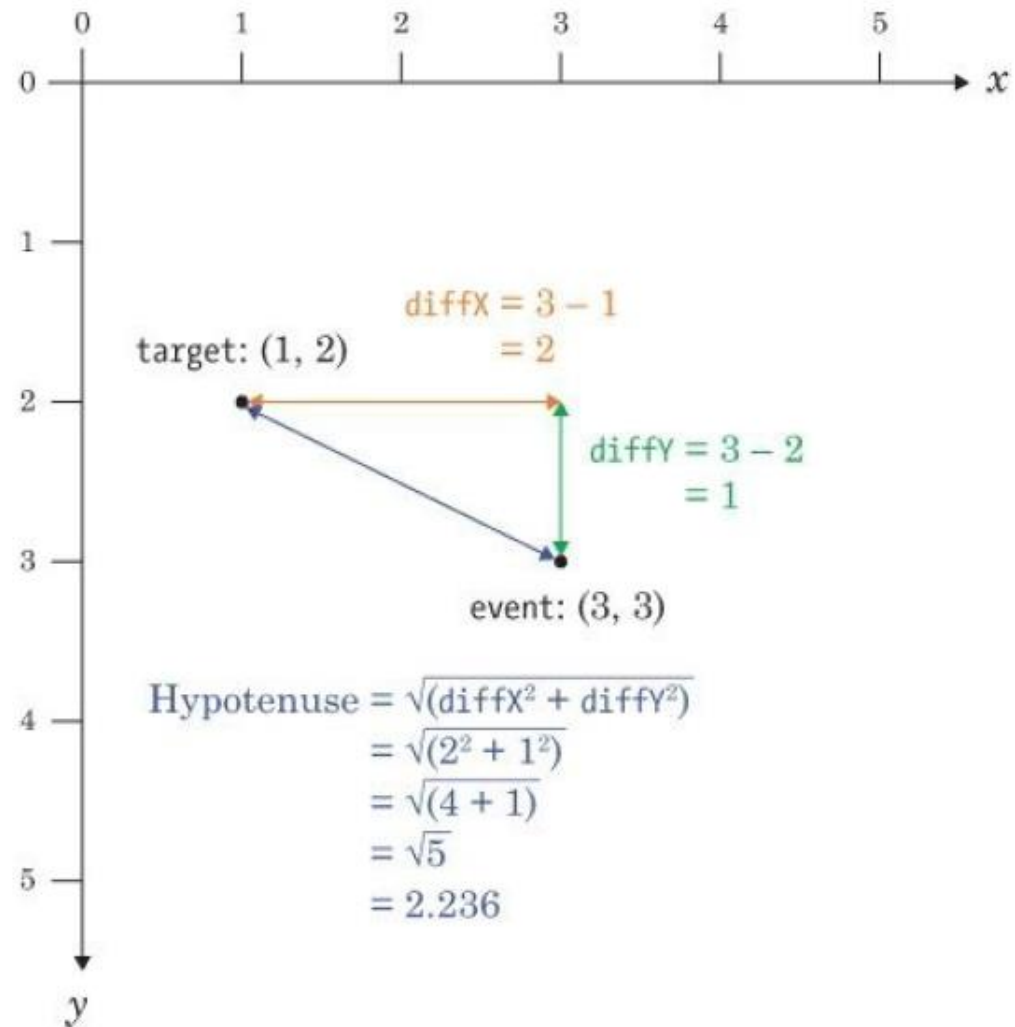
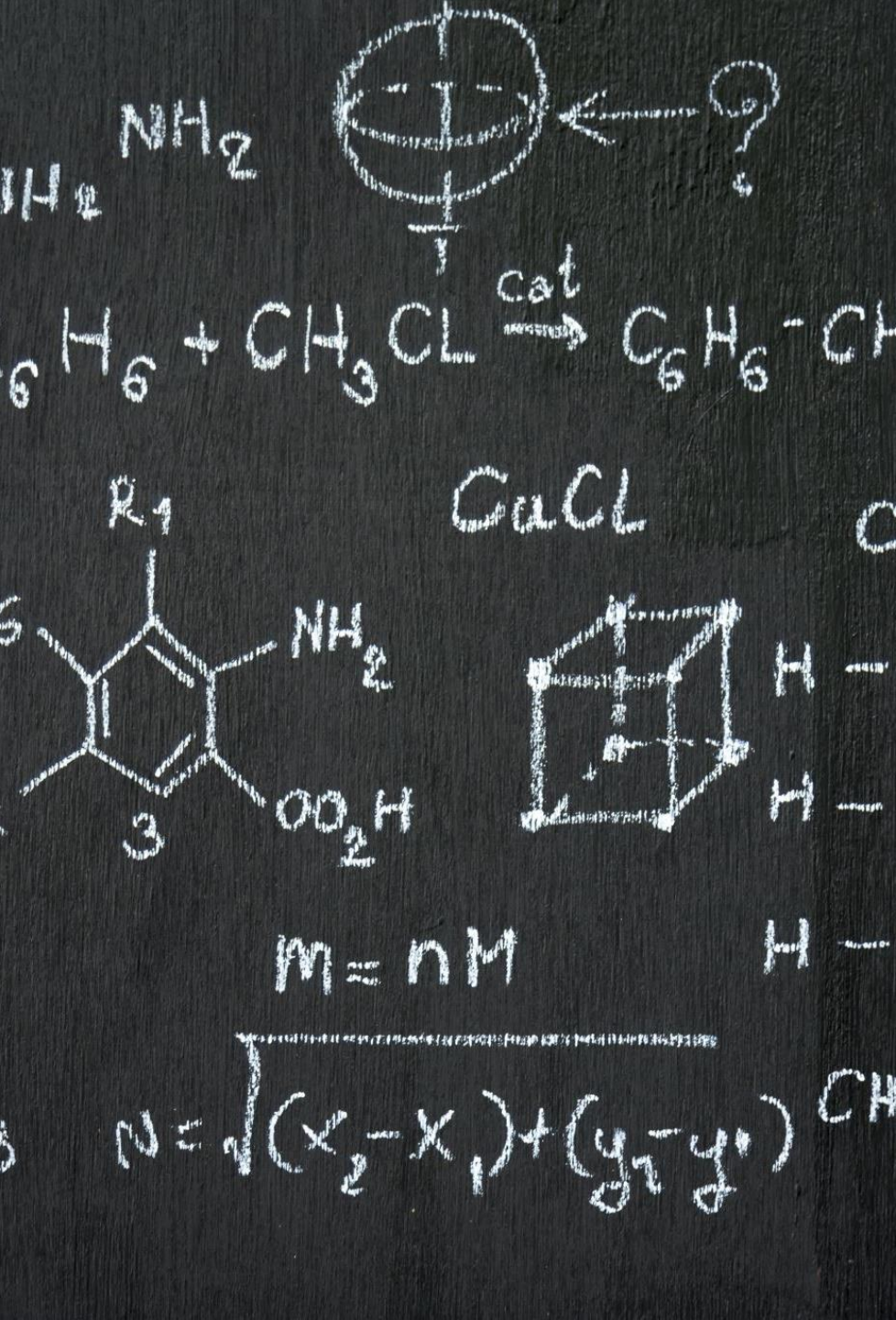


Figure 11-3. Calculating the hypotenuse to find out the distance between event and target



Using the Pythagorean Theorem

- To get the length of the hypotenuse, we first have to square diffX and diffY. We then add these squared values together, and get the square root using the JavaScript function Math.sqrt. So our complete formula for calculating the distance between the click and the target looks like this:

$\text{Math.sqrt}((\text{diffX} * \text{diffX}) + (\text{diffY} * \text{diffY}))$

- The getDistance function calculates this and returns the result.

A close-up photograph of a person's hand, wearing a dark red sleeve, reaching over a Go board. The board is made of light-colored wood and is covered with black and white Go stones. The background is a blurred green field, suggesting an outdoor setting.

Telling the Player How Close They Are

- Once we know the distance between the player's click and the treasure, we want to display a hint telling the player how close they are to the treasure, without telling them exactly how far away the treasure is.

- For this, we use the `getDistanceHint` function shown here:

```
var getDistanceHint = function (distance)
{
    if (distance < 10) { return "Boiling hot!"; }
    else if (distance < 20) { return "Really hot"; }
    else if (distance < 40) { return "Hot"; }
    else if (distance < 80) { return "Warm"; }
    else if (distance < 160) { return "Cold"; }
    else if (distance < 320) { return "Really cold"; }
    }
    else { return "Freezing!"; }
};
```

Telling the Player How Close They Are

- This function returns different strings depending on the calculated distance from the treasure. If the distance is less than 10, the function returns the string "Boiling hot!".
- If the distance is between 10 and 20, the function returns "Really hot". The strings get colder as the distance increases, up to the point where we return "Freezing!" if the distance is greater than 320 pixels.

Telling the Player How Close They Are

- We display the message to the player by adding it as text in the p element of the web page. The following code will go inside our click handler to calculate the distance, pick the appropriate string, and display that string to the player:

```
var distance = getDistance(event, target);  
var distanceHint = getDistanceHint(distance);  
$("#distance").text(distanceHint);
```

Telling the Player How Close They Are

- As you can see, we first call `getDistance` and then save the result as the variable `distance`. Next we pass that `distance` to the `getDistanceHint` function to pick the appropriate string and save it as `distanceHint`.
- The code `$("#distance").text(distanceHint);` selects the element with the id of "distance" (in this case the `p` element) and sets its text to `distanceHint` so that each time the player clicks the map, our web page tells them how close they are to the target.

Checking If the Player Won

- Finally, our click handler needs to check whether the player has won. Because pixels are so small, instead of making the player click the exact location of the treasure, we'll let them win if they click within 8 pixels.
- This code checks the distance to the treasure and displays a message telling the player that they've won:

```
if (distance < 8) {  
    alert("Found the treasure in " + clicks + "  
    clicks!");  
}
```
- If the distance is less than 8 pixels, this code uses alert to tell the player they found the treasure and how many clicks it took them to do so.

Program Integration

SECTION 11

```

// Get a random number from 0 to size
var getRandomNumber = function (size) {
    return Math.floor(Math.random() * size);
};
// Calculate distance between click event and target
var getDistance = function (event, target) {
    var diffX = event.offsetX - target.x;
    var diffY = event.offsetY - target.y;
    return Math.sqrt((diffX * diffX) + (diffY * diffY));
};
// Get a string representing the distance
var getDistanceHint = function (distance) {
    if (distance < 10) {return "Boiling hot!";}
    else if (distance < 20) {return "Really hot";}
    else if (distance < 40) {return "Hot";}
    else if (distance < 80) { return "Warm";}
    else if (distance < 160) {return "Cold";}
    else if (distance < 320) { return "Really cold";}
    else { return "Freezing!"; }
};

```

```

// Set up our variables
❶ var width = 400; var height = 400; var clicks = 0;
// Create a random target location
❷ var target = {
    x: getRandomNumber(width),
    y: getRandomNumber(height)
};
// Add a click handler to the img element
❸ $("#map").click(function (event) { clicks++;
// Get distance between click event and target
❹ var distance = getDistance(event, target);
// Convert distance to a hint
❺ var distanceHint = getDistanceHint(distance);
// Update the #distance element with the new hint
❻ $("#distance").text(distanceHint);
// If the click was close enough, tell them they won
❼ if (distance < 8) {
    alert("Found the treasure in " + clicks + " clicks!"); }
});

```


Project Integration

- First, we have the three functions getRandomNumber, getDistance, and getDistanceHint, which we've already looked at. Then, at ❶, we set up the variables we'll need: width, height, and clicks.
- After that, at ❷, we create the random location for the treasure.

Project Integration

- At ③ we create a click handler on the map element. The first thing this does is increment the clicks variable by 1. Then, at ④, it works out the distance between event (the click location) and target (the treasure location). At ⑤ we use the function `getDistanceHint` to convert this distance into a string representing the distance ("Cold", "Warm", and so on). We update the display at ⑥ so the user can see how far they are. Finally, at ⑦, we check to see whether the distance is under 8, and if so, we tell the player they've won and in how many clicks.
- This is the entire JavaScript for our game. If you add this to the second `<script>` tag in *treasure.html*, you should be able to play it in your browser! How many clicks does it take you to find the treasure?



Demonstration Program

TREASURE.HTML



Overview of Object-Oriented Programming

SECTION 12

Object-Oriented Programming

- Chapter 4 discussed JavaScript objects — collections of keys paired with values. In this chapter, we'll look at ways to create and use objects as we explore **object-oriented programming**.
- Object-oriented programming is a way to design and write programs so that all of the program's important parts are represented by objects. For example, when building a racing game, you could use object-oriented programming techniques to represent each car as an object and then create multiple car objects that share the same properties and functionality.

A Simple Object

SECTION 13



A Simple Object

- In Chapter 4, you learned that objects are made up of properties, which are simply pairs of keys and values. For example, in the following code the object `dog` represents a dog with the properties `name`, `legs`, and `isAwesome`:

```
var dog = {  
    name: "Pancake",  
    legs: 4,  
    isAwesome: true  
};
```


A Simple Object

- Once we create an object, we can access its properties using dot notation (discussed in Accessing Values in Objects). For example, here's how we could access the name property of our dog object:

```
dog.name;  
"Pancake"
```

- We can also use dot notation to add properties to a JavaScript object, like this:

```
dog.age = 6;
```

- This adds a new key-value pair (age: 6) to the object, as you can see below:

```
dog;  
Object {name: "Pancake", legs: 4,  
         isAwesome: true, age: 6  
}
```

Adding Methods to Objects

SECTION 14

Adding Methods to Objects

- In the preceding example, we created several properties with different kinds of values saved to them: a string ("Pancake"), numbers (4 and 6), and a Boolean (true).
- In addition to strings, numbers, and Booleans, you can save a *function* as a property inside an object. When you save a function as a property in an object, that property is called a *method*.
- In fact, we've already used several built-in JavaScript methods, like the join method on arrays and the **toUpperCase** method on strings.

Adding Methods to Objects

- Now let's see how to create our own methods. One way to add a method to an object is with dot notation. For example, we could add a method called bark to the dog object like this:
 - ❶ `dog.bark = function () {`
 - ❷ `console.log("Woof woof! My name is " + this.name + "!");`
 - ❸ `};`
 - ❹ `dog.bark();`
- Woof woof! My name is Pancake!

Adding Methods to Objects

- At ❶ we add a property to the dog object called bark and assign a function to it. At ❷, inside this new function, we use console.log to log Woof woof! My name is Pancake!.
- Notice that the function uses this.name, which retrieves the value saved in the object's name property. Let's take a closer look at how the this keyword works.

Using the this Keyword

- You can use the `this` keyword inside a method to refer to the object on which the method is currently being called. For example, when you call the `bark` method on the `dog` object, `this` refers to the `dog` object, so `this.name` refers to `dog.name`.
- The `this` keyword makes methods more versatile, allowing you to add the same method to multiple objects and have it access the properties of whatever object it's currently being called on.

Sharing a Method Between Multiple Objects

- Let's create a new function called **speak** that we can use as a method in multiple objects that represent different animals. When **speak** is called on an object, it will use the object's name (**this.name**) and the sound the animal makes (**this.sound**) to log a message.

```
var speak = function () {  
    console.log(this.sound + "! My name is "  
                + this.name + "!");  
};
```

Sharing a Method Between Multiple Objects

- Now let's create another object so we can add speak to it as a method:

```
var cat = {  
    sound: "Miaow",  
    name: "Mittens",  
    ① speak: speak  
};
```


Sharing a Method Between Multiple Objects

- Here we create a new object called `cat`, with `sound`, `name`, and `speak` properties. We set the `speak` property at ❶ and assign it the **`speak`** function we created earlier. Now **`cat.speak`** is a method that we can call by entering **`cat.speak()`**. Since we used the `this` keyword in the method, when we call it on `cat`, it will access the `cat` object's properties. Let's see that now:

```
cat.speak();
```

- Miaow! My name is Mittens!



Sharing a Method Between Multiple Objects

- When we call the `cat.speak` method, it retrieves two properties from the cat object: `this.sound` (which is "Miaow") and **`this.name`** (which is "Mittens").
- We can use the same `speak` function as a method in other objects too:

Sharing a Method Between Multiple Objects

```
var pig = {  
    sound: "Oink",  
    name: "Charlie",  
    speak: speak  
};  
var horse = {  
    sound: "Neigh",  
    name: "Marie",  
    speak: speak  
};  
pig.speak();  
Oink! My name is Charlie!  
horse.speak();  
Neigh! My name is Marie!
```

Sharing a Method Between Multiple Objects

- Again, each time this appears inside a method, it refers to the object on which the method is called. In other words, when you call `horse.speak()`, this will refer to `horse`, and when you call `pig.speak()`, this refers to `pig`.
- To share methods between multiple objects, you can simply add them to each object, as we just did with `speak`. But if you have lots of methods or objects, adding the same methods to each object individually can become annoying, and it can make your code messier, too. Just imagine if you needed a whole zoo full of 100 animal objects and you wanted each to share a set of 10 methods and properties.
- JavaScript object constructors offer a better way to share methods and properties between objects, as we'll see next.

Constructors

SECTION 15

Creating Objects Using Constructors

- A JavaScript *constructor* is a function that creates objects and gives them a set of built-in properties and methods. Think of it as a specialized machine for creating objects, kind of like a factory that can churn out tons of copies of the same item.
- Once you've set up a constructor, you can use it to make as many of the same object as you want. To try it out, we'll build the beginnings of a racing game, using a Car constructor to create a fleet of cars with similar basic properties and methods for steering and acceleration.

Anatomy of the Constructor

- Each time you call a constructor, it creates an object and gives the new object built-in properties. To call a normal function, you enter the function name followed by a pair of parentheses.
- To call a constructor, you enter the keyword `new` (which tells JavaScript that you want to use your function as a constructor), followed by the constructor name and parentheses. Figure 12-1 shows the syntax for calling a constructor.

The new object
is saved into
this variable.



Arguments passed
to the constructor



```
var car = new Car(100, 200)
```



The name of
the constructor

Figure 12-1. The syntax for calling a constructor named `Car` with two arguments

Creating a Car Constructor

Now let's create a Car constructor that will add an x and y property to each new object it creates.

These properties will be used to set each car's onscreen position when we draw it.

Creating the HTML Document

- Before we can build our constructor, we need to create a new HTML document. Make a new file called *cars.html* and enter this HTML into it:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Cars</title>
  </head>
  <body>
    <script src="https://code.jquery.com/jquery-3.4.1.js"></script>
    <script>
      // Code goes here
    </script>
  </body>
</html>
```



Demonstration Program

CARS0.HTML

The Car Constructor function

- Now add this code to the empty `<script>` tags in *cars.html* (replacing the comment `// Code goes here`) to create the Car constructor that gives each car a set of coordinates.

```
<script>
    var Car = function (x, y) {
        this.x = x;
        this.y = y;
    };
</script>
```

- Our new constructor Car takes the arguments x and y. We've added the properties `this.x` and `this.y` to store the x and y values passed to Car in our new object. This way, each time we call Car as a constructor, a new object is created with its x and y properties set to the arguments we specify.

Calling the Car Constructor

- As I mentioned earlier, the keyword `new` tells JavaScript that we're calling a constructor to create a new object. For example, to create a car object named `tesla`, open *`cars.html`* in a web browser and then enter this code in the Chrome JavaScript console:

```
var tesla = new Car(10, 20);  
tesla;  
Car {x: 10, y: 20}
```

Calling the Car Constructor

- The code `new Car(10, 20)` tells JavaScript to create an object using `Car` as a constructor, pass in the arguments 10 and 20 for its `x` and `y` properties, and return that object. We assign the returned object to the `tesla` variable with `var tesla`.
- Then when we enter `tesla`, the Chrome console returns the name of the constructor and its `x` and `y` values: `Car {x: 10, y: 20}`.

Drawing the Cars

SECTION 16

Drawing the Cars

- To show the objects created by the Car constructor, we'll create a function called `drawCar` to place an image of a car at each car object's (x, y) position in a browser window. Once we've seen how this function works, we'll rewrite it in a more object-oriented way in Adding a draw Method to the Car Prototype.

Add this code between the <script> tags in *cars.html*:

```
<script>
```

```
    var Car = function (x, y) {  
        this.x = x;  
        this.y = y;  
    };
```

```
    var drawCar = function (car) {
```

```
❶  var carHtml = '';
```

```
❷  var carElement = $(carHtml);
```

```
❸  carElement.css({  
        position: "absolute",  
        left: car.x,  
        top: car.y  
    });
```

```
❹  $("body").append(carElement);  
    };
```

```
</script>
```

Drawing the Cars

- At ❶ we create a string containing HTML that points to an image of a car. (Using single quotes to create this string lets us use double quotes in the HTML.)
- At ❷ we pass carHTML to the \$ function, which converts it from a string to a jQuery element. That means the carElement variable now holds a jQuery element with the information for our tag, and we can tweak this element before adding it to the page.

Drawing the Cars

- At ③ we use the `css` method on `carElement` to set the position of the car image. This code sets the left position of the image to the car object's `x` value and its top position to the `y` value.
- In other words, the left edge of the image will be `x` pixels from the left edge of the browser window, and the top edge of the image will be `y` pixels down from the top edge of the window.

Drawing the Cars

- Finally, at ④ we use jQuery to append the carElement to the body element of the web page. This final step makes the carElement appear on the page. (For a reminder on how append works, see [Creating New Elements with jQuery](#).)



Demonstration Program

CARS1.HTML

Testing Function

SECTION 17

Testing the drawCar Function

- Let's test the drawCar function to make sure it works. Add this code to your *cars.html* file (after the other JavaScript code) to create two cars.

```
$ ("body") .append (carElement) ;  
};  
var tesla = new Car (20, 20) ;  
var nissan = new Car (100, 200) ;  
drawCar (tesla) ;  
drawCar (nissan) ;  
</script>
```

- Here, we use the Car constructor to create two car objects, one at the coordinates (20, 20) and the other at (100, 200), and then we use drawCar to draw each of them in the browser. Now when you open *cars.html*, you should see two car images in your browser window, as shown in Figure 12-2.

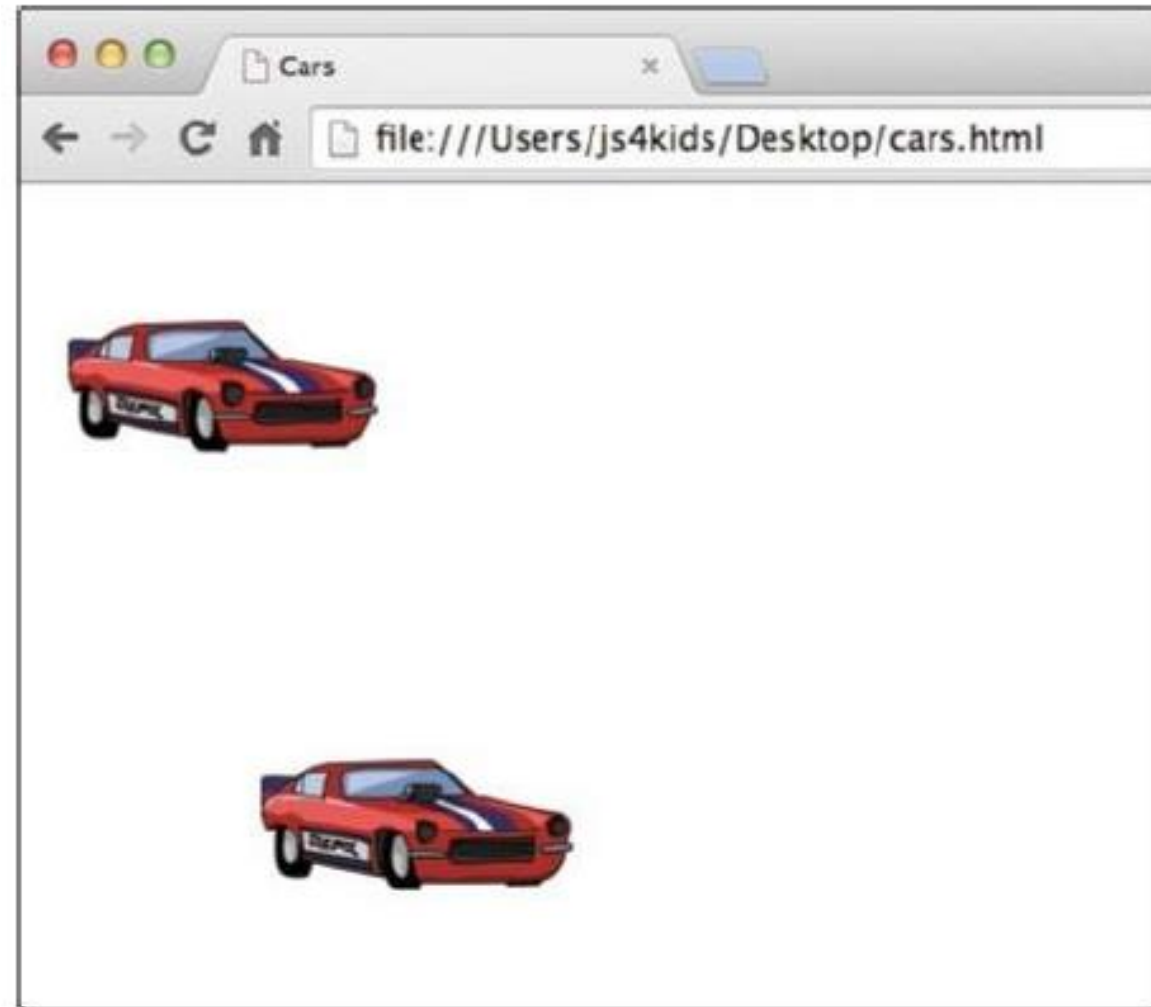


Figure 12-2. Drawing cars using drawCar



Demonstration Program

CARS.HTML

Prototypes

SECTION 18



Customizing Objects with Prototypes

- A more object-oriented way to draw our cars would be to give each car object a draw method. Then, instead of writing `drawCar(tesla)`, you'd write `tesla.draw()`. In object-oriented programming, we want objects to have their own functionality built in as methods. In this case, the `drawCar` function is always meant to be used on car objects, so instead of saving `drawCar` as a separate function, we should include it as part of each car object.
- JavaScript *prototypes* make it easy to share functionality (as methods) between different objects. All constructors have a prototype property, and we can add methods to it. Any method that we add to a constructor's prototype property will be available as a method to all objects created by that constructor.
- Figure 12-3 shows the syntax for adding a method to a prototype property.

The
constructor
name



The
method
name



```
Car.prototype.draw = function () {  
    // The body of the method  
}
```

Figure 12-3. The syntax for adding a method to a prototype property



Adding a draw Method to the Car Prototype

- Let's add a draw method to Car.prototype so that all objects we create using Car will have the draw method. Using File ► Save As, save your cars.html file as cars2.html.
- Then replace all of the JavaScript in your second set of <script> tags in cars2.html with this code:

```
❶ var Car = function (x, y) {  
    this.x = x;  
    this.y = y;  
  
    };  
  
❷ Car.prototype.draw = function () {  
    var carHtml = '';  
  
❸    this.carElement = $(carHtml);  
    this.carElement.css({  
        position: "absolute",  
  
❹        left: this.x,  
        top: this.y  
    });  
    $("body").append(this.carElement);  
};  
  
var tesla = new Car(20, 20);  
var nissan = new Car(100, 200);  
tesla.draw();  
nissan.draw();
```

Adding a draw Method to the Car Prototype

- After creating our Car constructor at ❶, we add a new method called draw to Car.prototype at ❷.
- This makes the draw method part of all of the objects created by the Car constructor.
- The contents of the draw method are a modified version of our drawCar function. First, we create an HTML string and save it as carHTML. At ❸ we create a jQuery element representing this HTML, but this time we save it as a property of the object by assigning it to this.carElement. Then at ❹, we use this.x and this.y to set the coordinates of the top-left corner of the current car image. (Inside a constructor, this refers to the new object currently being created.)

Adding a draw Method to the Car Prototype

- When you run this code, the result should look like Figure 12-2. We haven't changed the code's functionality, only its organization. The advantage to this approach is that the code for drawing the car is part of the car, instead of a separate function.



Demonstration Program

CARS2.HTML



Adding a moveRight Method

- Now let's add some methods to move the cars around, beginning with a moveRight method to move the car 5 pixels to the right of its current position. Add the following code after your definition of Car.prototype.draw:

```
this.carElement.css({  
    position: "absolute",  
    left: this.x,  
    top: this.y  
});  
$("body").append(this.carElement);  
};  
Car.prototype.moveRight = function () {  
    this.x += 5;  
    this.carElement.css({  
        left: this.x,  
        top: this.y  
    });  
};  
};
```

Adding a moveRight Method

- We save the moveRight method in Car.prototype to share it with all objects created by the Car constructor. With this.x += 5 we add 5 to the car's x value, which moves the car 5 pixels to the right.
- Then we use the css method on this.carElement to update the car's position in the browser.
- Try the moveRight method in the browser console. First, refresh *cars2.html*, and then open the console and enter these lines:

```
tesla.moveRight();  
tesla.moveRight();  
tesla.moveRight();
```
- Each time you enter tesla.moveRight, the top car should move 5 pixels to the right. You could use this method in a racing game to show the car moving down the racetrack.



Adding the Left, Up, and Down move Methods

- Now we'll add the remaining directions to our code so that we can move our cars around the screen in any direction. These methods are basically the same as `moveRight`, so we'll write them all at once.

- Each of these methods moves the car by 5 pixels in the specified direction by adding or subtracting 5 from each car's x or y value.



Add the following methods to *cars2.html* just after the code for *moveRight*:

```
Car.prototype.moveRight = function () {  
    this.x += 5;  
    this.carElement.css({  
        left: this.x,  
        top: this.y  
    });  
};  
  
Car.prototype.moveLeft = function () {  
    this.x -= 5;  
    this.carElement.css({  
        left: this.x,  
        top: this.y  
    });  
};  
  
Car.prototype.moveUp = function () {  
    this.y -= 5;  
    this.carElement.css({  
        left: this.x,  
        top: this.y  
    });  
};  
  
Car.prototype.moveDown = function () {  
    this.y += 5;  
    this.carElement.css({  
        left: this.x,  
        top: this.y  
    });  
}
```



Demonstration Program

CARS3.HTML