

Chapter 9. Handling events

Asynchronous Coding



After this chapter you're going to realize you aren't in Kansas anymore. Up until now, you've been writing code that typically executes from top to bottom—sure, your code might be a little more complex than that, and make use of a few functions, objects and methods, but at some point the code just runs its course. Now, we're awfully sorry to break this to you this late in the book, but that's **not how you typically write JavaScript code**. Rather, most JavaScript is written to **react to events**. What kind of events? Well, how about a user clicking on your page, data arriving from the network, timers expiring in the browser, changes happening in the DOM and that's just a few examples. In fact, all kinds of events are happening **all the time**, behind the scenes, in your browser. In

this chapter we're going rethink our approach to JavaScript coding, and learn how and why we should write code that reacts to events.

BRAIN POWER

You know what a browser does, right? It retrieves a page and all that page's contents and then renders the page. But the browser's doing a lot more than just that. What else is it doing? Choose any of the tasks below you suspect the browser is doing behind the scenes. If you aren't sure just make your best guess.

☐ Knows when the page is fully loaded and displayed.

☐ Keeps track of all the clicks you make to the page, be it on a button, link or elsewhere.

☐ Knows when a user submits a form.

☐ Knows when the user presses keys on a keyboard.

☐ Knows when an element gets user interface focus.

☐ Watches all mouse movement.

☐ Watches the clock and manages timers and timed events.

☐ Retrieves additional data for your page.

☐ Tracks when the page has been resized or scrolled.



Knows when the cookies are finished baking.

SHARPEN YOUR PENCIL

Pick two of the events above. If the browser could notify your code when these events occurred, what cool or interesting code might you write?

NOTE

No, you can't use the cookie event as one of your examples!

What are events?

We're sure you know by now that after the browser retrieves and displays your page, it doesn't just sit there. Behind the scenes, a lot is going on: users are clicking buttons, the mouse location is being tracked, additional data is becoming available on the network, windows are getting resized, timers are going off, the browser's location could be changing, and so on. All these things cause *events* to be triggered.

NOTE

A browser's geo-location, as well as a number of other advanced types of events, is something we cover in *Head First HTML5 Programming*. In this book we'll stick to the bread & butter foundational types of events.

Whenever there's an event, there is an opportunity for your code to *handle it*; that is, to supply some code that will be invoked when the event occurs. Now, you're not required to handle any of these events, but you'll need to handle them if you want interesting things to happen when they occur—like, say, when the button click event happens, you might want to add a new song to a playlist; when new data arrives you might want to

process it and display it on your page; when a timer fires you might want to tell a user the hold on a front row concert ticket is going to expire, and so on.



Whenever there's an event, there is an opportunity for your code to handle it.

What's an event handler?

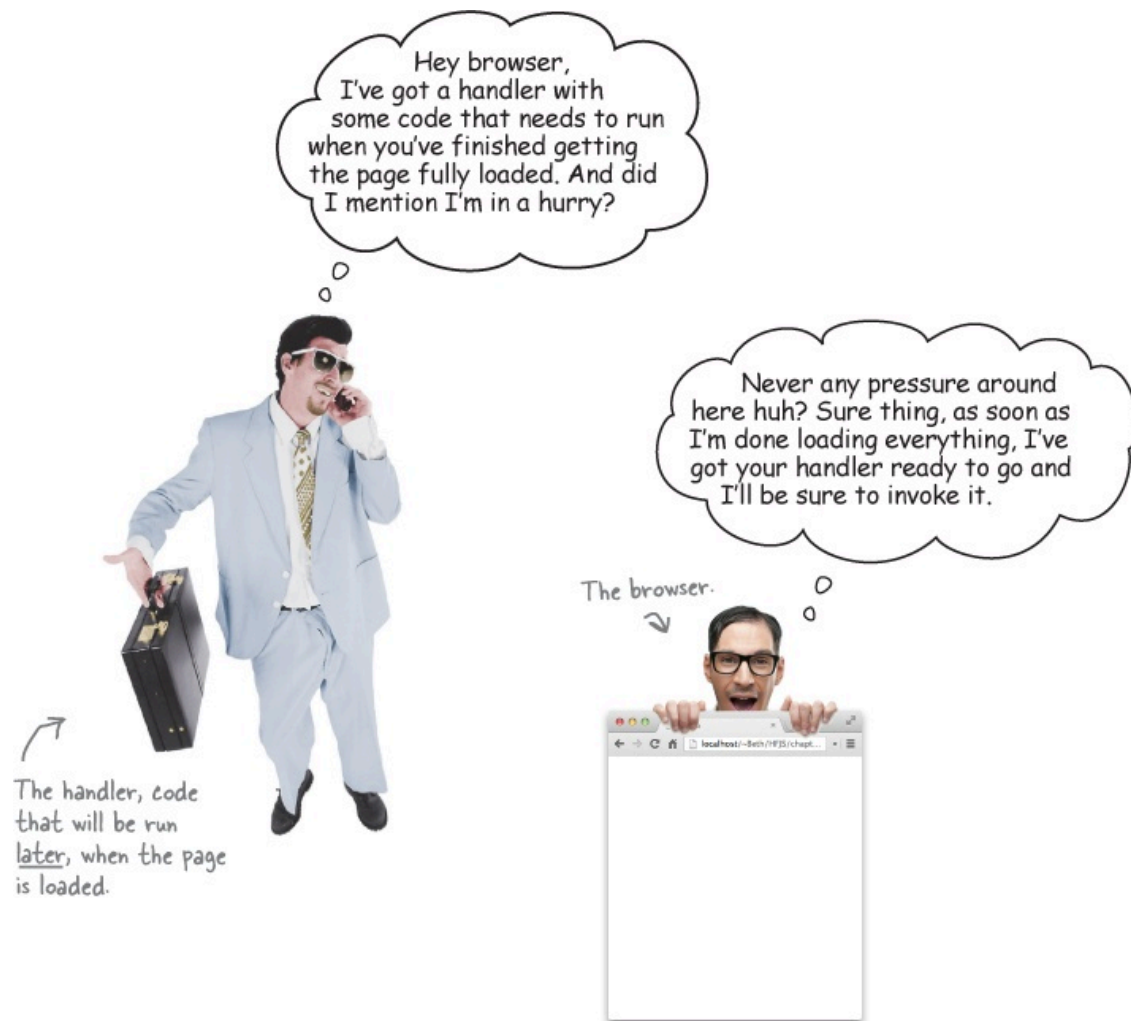
We write *handlers* to handle events. Handlers are typically small pieces of code that know what to do when an event occurs. In terms of code, a handler is just a function. When an event occurs, its handler function is called.

NOTE

You might also hear developers use the name callback or listener instead of handler.

To have your handler called when an event occurs, you first need to *register it*. As you'll see, there are a few different ways to do that depending on what kind of event it is. We'll get into all that, but for now let's get started

with a simple example, one you've seen before: the event that's generated when a page is fully loaded.



How to create your first event handler

There's no better way to understand events than by writing a handler and wiring it up to handle a real, live event. Now, remember, you've already seen a couple of examples of handling events—including the page load event—but we've never fully explained how event handling works. The page load event is triggered when the browser has fully loaded and displayed all the content in your page (and built out the DOM representing the page).

Let's step through what it takes to write the handler and to make sure it gets invoked when the page load event is triggered:

1. First we need to write a function that can handle the page load event when it occurs. In this case, the function is going to announce

to the world “I’m alive!” when it knows the page is fully loaded.

A handler is just an ordinary function.

```
function pageLoadedHandler() {  
    alert("I'm alive!");  
}
```

Here's our function, we'll name it `pageLoadedHandler`, but you can call it anything you like.

Remember we often refer to this as a handler or a callback.

This event handler doesn't do much. It just creates an alert.

2. Now that we have a handler written and ready to go, we need to wire things up so the browser knows there's a function it should invoke when the load event occurs. To do that we use the `onload` property of the window object, like this:

```
window.onload = pageLoadedHandler;
```

In the case of the load event, we assign the name of the handler to the window's `onload` property.

Now when the page load event is generated, the `pageLoadedHandler` function is going to be called.

We're going to see that different kinds of events are assigned handlers in different ways.

3. That's it! Now, with this code written, we can sit back and know that the browser will invoke the function assigned to the `window.onload` property when the page is loaded.

Test drive your event



Go ahead and create a new file, “event.html”, and add the code to test your load event handler. Load the page into the browser and make sure you see the alert.



BRAIN POWER

If we didn't have functions, could we have event handlers?



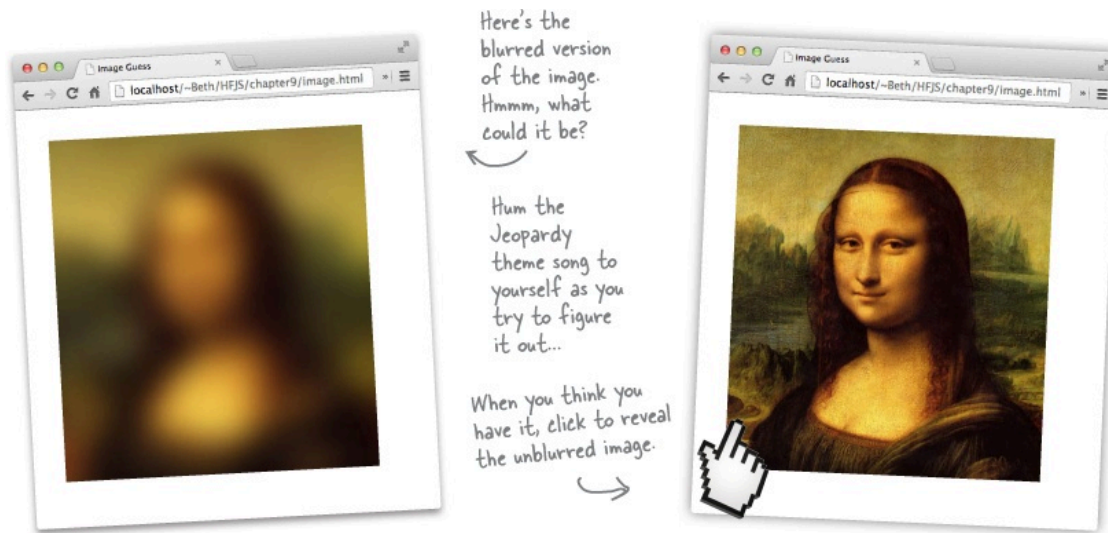
As we already mentioned, up until now you’ve taken a rather, let’s say, linear approach to writing code: you took an algorithm, like computing the best bubble solution, or generating the 99 bottles song, and wrote the code stepwise, top to bottom.

But remember the Battleship game? The code for that game didn’t quite fit the linear model—sure, you wrote some code that set up the game, initialized the model, and all that—but then the main part of the game operated in a *different way*. Each time you wanted to fire at another ship you entered your guess into a form input element and pressed the “Fire” button. That button then caused a whole sequence of actions that resulted in the next move of the game being executed. In that case your code was *reacting* to the user input.

Organizing code around reacting to events is a different way of thinking about how you write your code. To write code this way, you need to consider the events that can happen, and how your code should react. Computer science types like to say that this kind of code is *asynchronous*, because we’re writing code to be invoked *later*, *if* and *when* an event occurs. This kind of coding also changes your perspective from one of encoding an algorithm step-by-step into code, into one of gluing together an application that is composed of many handlers handling many different kinds of events.

Getting your head around events... by creating a game

The best way to understand events is with experience, so let’s get some more by writing a simple game. The game works like this: you load a page and are presented with an image. Not just any image, but a really blurred image. Your job is to guess what the image is. And, to check your answer, you click on the image to unblur it.



Let's start with the markup. We'll use two JPG images. One is blurred and the other isn't. We've named them "zeroblur.jpg" and "zero.jpg" respectively. Here's the markup:

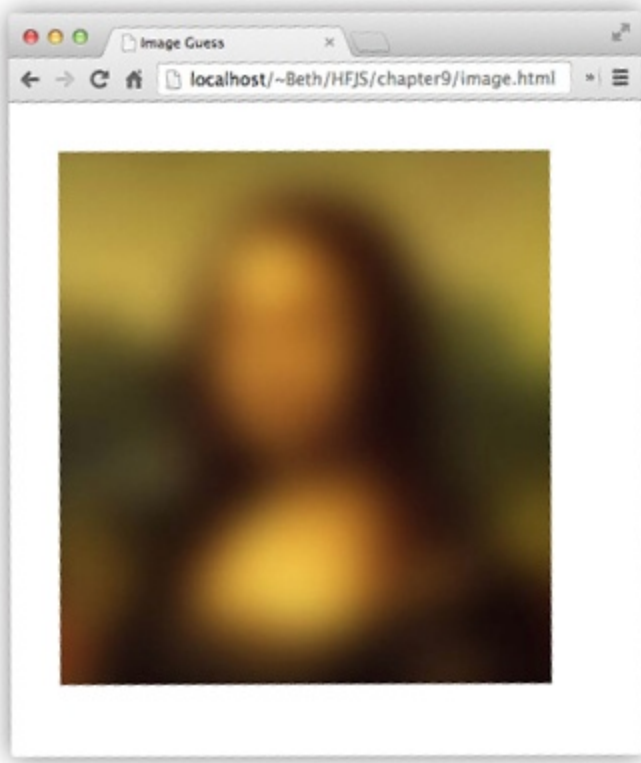
```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title> Image Guess </title>
  <style> body { margin: 20px; } </style>
  <script> </script>
</head>
<body>
  
</body>
</html>
```

Just some basic HTML, with a `<script>` element all ready for our code. Rather than use a separate file for the JavaScript, we'll keep it simple and add the script here. As you'll see, there is very little code needed to implement this.

And here's the blurred image, placed in the page. We'll give it an id of "zero". You'll see how we use the id in a sec...

Implementing the game

Go ahead and load this markup in your browser and you'll see the blurred image. To implement the game, we need to react to a click on the image in order to display the unblurred version of the image.



Lucky for us, every time an HTML element in the page is clicked (or touched on a mobile device), an event is generated. Your job is to create a handler for that event, and in it write the code to display the unblurred version of the image. Here's how you're going to do that:

1. **Access the image object in the DOM and assign a handler to its onclick property.**
2. **In your handler, write the code to change the image src attribute from the blurred image to the unblurred one.**

Let's walk through these steps and write the code.

Step 1: access the image in the DOM

Getting access to the image is old hat for you; we just need to use our old friend, the `getElementById` method, to get a reference to it.

```
var image = document.getElementById("zero");
```

← Here we're grabbing a reference to the image element and assigning it to the image variable.

Oh, but we also need this code to run only *after* the DOM for the page has been created, so let's use the window's `onload` property to ensure that.

We'll place our code into a function, `init`, that we'll assign to the `onload` property.

NOTE

Remember, we can't get the image from the DOM until the page has finished loading.

```
window.onload = init;
function init() {
  var image = document.getElementById("zero");
}
```

↖ In the code of `init`, we'll grab a reference to the image with `id="zero"`.

↖ We create a function `init`, and assign it to the window's `onload` property to make sure this code doesn't run until the page is fully loaded.

↖ Remember in JavaScript the order in which you define your functions doesn't matter. So we can define `init` after we assign it to the `onload` property.

Step 2: add the handler, and update the image

To add a handler to deal with clicks on the image, we simply assign a function to the image's `onclick` property. Let's call that function `showAnswer`, and we'll define it next.

```
window.onload = init;
function init() {
  var image = document.getElementById("zero");
  image.onclick = showAnswer;
}
```

↖ Using the image object from the DOM, we're assigning a handler to its `onclick` property.

Now we need to write the `showAnswer` function, which unblurs the image by resetting the image element's `src` property to the unblurred image:

```
function showAnswer() {
  var image = document.getElementById("zero");
  image.src = "zero.jpg";
}
```

First, we have to get the image from the DOM again.

↖ Remember the blurred version is named `"zeroblur.jpg"` and the unblurred is named `"zero.jpg"`.

↖ Once we have the image, we can change it by setting its `src` property to the unblurred image.

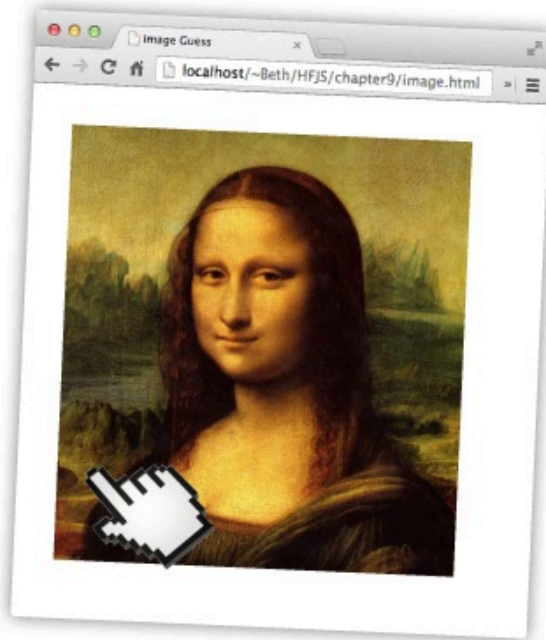
Test drive

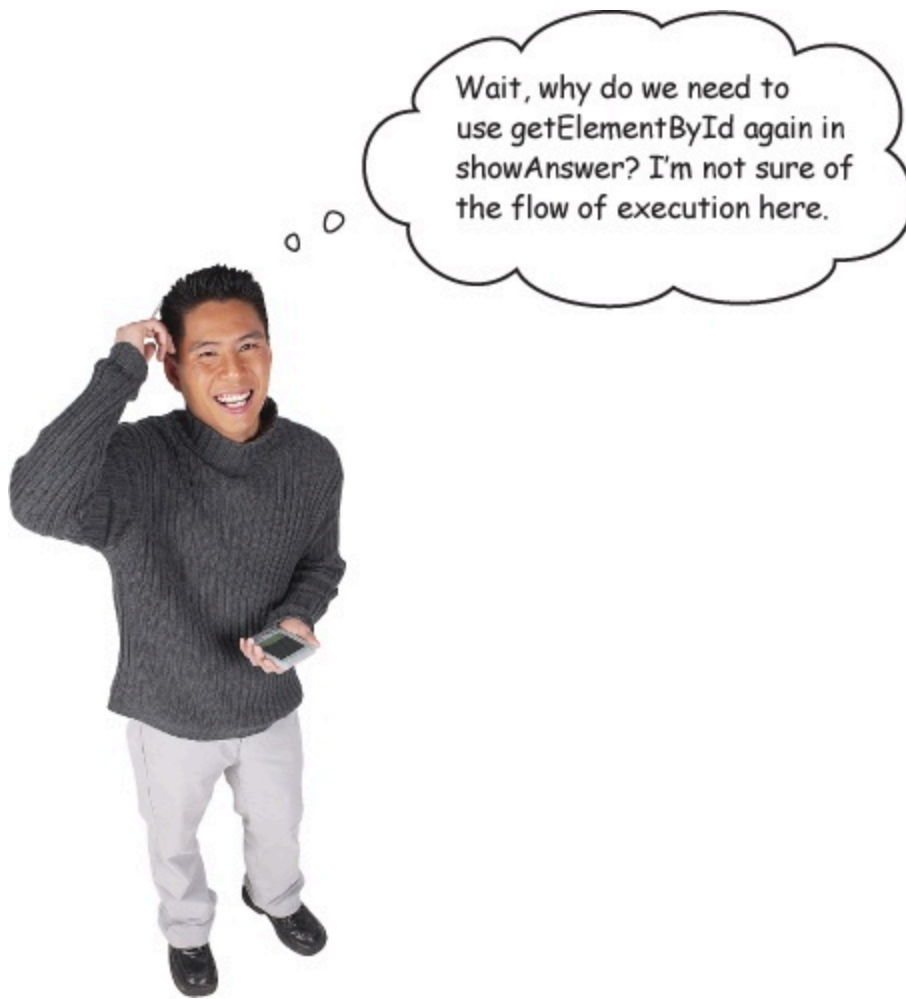


Let's take this simple game for a test drive. Make sure you've got all the HTML, CSS and JavaScript typed into a file named "image.html", and that you've got the images you downloaded from

<http://wickedlysmart.com/hfjs> in the same folder. Once all that's done, load up the file in your browser and give it a try!

Click anywhere on the image to have the showAnswer handler called. When that happens, the src of the image is changed to reveal the answer.





Ah, yes. It can get tricky to follow the flow of execution in code with a lot of event handlers. Remember, the `init` function is called when the page is loaded. But the `showAnswer` function isn't called until later, when you click the image. So these two event handlers get called at two different times.

In addition, remember your scope rules. In the `init` function we're putting the object returned by `getElementById` into a *local* variable `image`, which means when that function completes, the variable falls out of scope and is destroyed. So later, when the `showAnswer` function is called, we have to get the image object again from the DOM. Sure, we could have put this in a global variable, but overuse of globals can lead to confusing and buggy code, which we'd like to avoid.

Q: Q: Is setting the src property of the image the same as setting the src attribute using setAttribute?

A: *In this case, yes, it is. When you get an HTML element from the DOM using `getElementById`, you're getting an element object that has several methods and properties. All element objects come with a property, `id`, that is set to the id of the HTML element (if you've given it one in your HTML). The image element object also comes with a `src` property that is set to the image file specified in the `src` attribute of the `` element.*

Not all attributes come with corresponding object properties, however, so you will need to use `setAttribute` and `getAttribute` for those. And in the case of `src` and `id`, you can use either the properties or get/set them using `getAttribute` and `setAttribute` and it does the same thing.

Q: Q: So do we have a handler called within a handler?

A: *Not really. The load handler is the code that is called when the page is fully loaded. When the load handler is called, we assign a handler to the image's `onclick` property, but it won't be called until you actually click on the image. When you do that (potentially a long time after the page has loaded), the `showAnswer` click handler is called. So the two handlers get called at different times.*



Below, you'll find your game code. Your job is to play like you're the browser and to figure out what you need to do after each event. After you've done the exercise, look at the end of the chapter to see if you got everything. We've done the first bit for you.

```
window.onload = init;
function init() {
  var image = document.getElementById("zero");
  image.onclick = showAnswer;
}

function showAnswer() {
  var image = document.getElementById("zero");
  image.src = "zero.jpg";
}
```

↖ Here's the code you're executing...

When page is being loaded...

First define the functions init and showAnswer

When page load event occurs...

When image click event occurs...

↖ Your answers go here.

BRAIN POWER

What if you had an entire page of images that could each be individually deblurred by clicking? How would you design your code to handle this? Make some notes. What might be the naive way of implementing this? Is there a way to implement this with minimal code changes to what you've already written?



Judy: Hey guys. So far the image guessing game works great. But we really should expand the game to include more images on the page.

Jim: Sure, Judy, that's exactly what I was thinking.

Joe: Hey, I've already got a bunch of images ready to go, we just need the code. I've followed the naming convention of "zero.jpg", "zeroblur.jpg", "one.jpg", "oneblur.jpg", and so on...

Jim: Are we going to need to write a new click event handler for each image? That's going to be a lot of repetitive code. After all, every event handler's going to do exactly the same thing: replace the blurred image with its unblurred version, right?

Joe: That's true. But I'm not sure I know how to use the same event handler for multiple images. Is that even possible?

Judy: What we can do is assign the same handler, which really means the same function, to the `onclick` property of every image in the game.

Joe: So the same function gets called for every image that is clicked on?

Judy: Right. We'll use `showAnswer` as the handler for every image's click event.

Jim: Hmm, but how will we know which image to deblur?

Joe: What do you mean? Won't the click handler know?

Jim: How will it know? Right now, our `showAnswer` function assumes we clicked on the image with the id "zero". But if we're calling `showAnswer` for every image's click event, then our code needs to work for any of the images.

Joe: Oh... right... so how do we know which image was clicked?

Judy: Actually I've been reading up on events, and I think there is a way for the click handler to know the element the user clicked on. But let's deal with that part later. First let's add some more images to the game, and see how to set the same event handler for all of them... then we'll figure out how to determine which image the user clicked.

Joe, Jim: Sounds good!

Let's add some more images

We've got a whole set of new images, so let's start by adding them to the page. We'll add five more images for a total of six. We'll also modify the CSS to add a little whitespace between the images:

GET THE IMAGES



You'll find all the images in the **Chapter 9** folder you downloaded from <http://wickedlysmart.com/hfjs>.

```

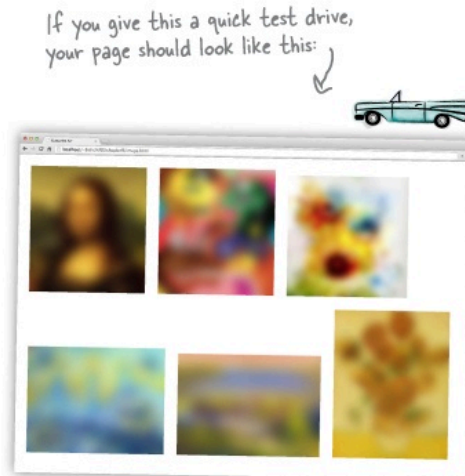
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title> Image Guess </title>
  <style>
    body { margin: 20px; }
    img { margin: 20px; }
  </style>
  <script>
    window.onload = init;
    function init() {
      var image = document.getElementById("zero");
      image.onclick = showAnswer;
    }
    function showAnswer() {
      var image = document.getElementById("zero");
      image.src = "zero.jpg";
    }
  </script>
</head>
<body>
  
  
  
  
  
  
</body>
</html>

```

We're just adding a margin of 20px between the images with this CSS property.

If you give this a quick test drive, your page should look like this:

And here are the five new images we're adding. Notice we're using the same id & src naming scheme (and image naming scheme) for each one. You'll see how this is going to work in a bit...



Now we need to assign the same event handler to each image's onclick property

Now we have more images in the page, but we have more work to do. Right now you can click on the first image (of the *Mona Lisa*) and see the unblurred image, but what about the other images?

We *could* write a new, separate handler function for each image, but, from the discussion so far you know that would be tedious and wasteful. Here, have a look:

```
window.onload = init;
```

```
function init() {
```

```
    var image0 = document.getElementById("zero");
```

```
    image0.onclick = showImageZero;
```

```
    var image1 = document.getElementById("one");
```

```
    image1.onclick = showImageOne;
```

```
    ...
```

```
}
```

```
function showImageZero() {
```

```
    var image = document.getElementById("zero");
```

```
    image.src = "zero.jpg";
```

```
}
```

```
function showImageOne() {
```

```
    var image = document.getElementById("one");
```

```
    image.src = "one.jpg";
```

```
}
```

```
...
```

← We could get each image element from the page and assign a separate click handler to each one. We'd have to do this six times... we're only showing two here.

← The other four would be set here.

← And we'd need six different click handlers, one for each image.

←

← And we'd need four more handler functions here.

What are the disadvantages of writing a separate handler for each image? Check all that apply:

- ☐ Lots of redundant code in each handler.
- ☐ If we need to change the code in one handler, we're probably going to have to change them all.
- ☐ Generates a lot of code.
- ☐ Hard to keep track of all the images and handlers.
- ☐ Hard to generalize for an arbitrary number of images.
- ☐ Harder for others to work on the code.

How to reuse the same handler for all the images

Clearly writing a handler for each image isn't a good way to solve this problem. So what we're going to do instead is use our existing handler, `showAnswer`, to handle all of the click events for all the images. Of course,

we'll need to modify `showAnswer` a little bit to make this work. To use `showAnswer` for all the images we need to do two things:

1. Assign the `showAnswer` click handler function to every image on the page.
2. Rework `showAnswer` to handle unblurring any image, not just `zero.jpg`.

And we'd like to do both these things in a generalized way that works even if we add more images to the page. In other words, if we write the code right, we should be able to add images to the page (or delete images from the page) without any code changes. Let's get started.

Assigning the click handler to all images on the page

Here's our first hurdle: in the current code we use the `getElementById` method to grab a reference to image "zero", and assign the `showAnswer` function to its `onclick` property. Rather than hardcoding a call to `getElementById` for each image, we're going to show you an easier way: we'll grab all the images at once, iterate through them, and set up the click handler for each one. To do that we'll use a DOM method you haven't seen yet: `document.getElementsByTagName`. This method takes a tag name, like `img` or `p` or `div`, and returns a list of elements that match it. Let's put it to work:

```
function init() {  
  var image = document.getElementById("zero");  
  image.onclick = showAnswer;  
  
  var images = document.getElementsByTagName("img");  
  for (var i = 0; i < images.length; i++) {  
    images[i].onclick = showAnswer;  
  }  
};
```

We'll get rid of the old code to get image "zero" and set its handler.

Now we're getting elements from the page using a tag name, `img`. This finds every image in the page and returns them all. We store the resulting images in the `images` variable.

Then we iterate over the images, and assign the `showAnswer` click handler to each image in turn. Now the `onclick` property of each image is set to the `showAnswer` handler.



The `document.getElementsByTagName` method works a lot like `document.getElementById`, except that instead of getting an element by its id, we're getting elements by tag name, in this case the tag name "img". Of course, your HTML can include many `` elements, so this method may return many elements, or one element, or even zero elements, depending on how many images we have in our page. In our image game example, we have six `` elements, so we'll get back a list of six image objects.

What we get back is a list of element objects that match the specified tag name.

```
var images = document.getElementsByTagName("img");
```

What's returned is an array-like list of objects. It's not exactly an array but has qualities similar to an array.

Notice the "s" here. That means we might get many elements back.

Put the tag name in quotes here (and don't include the < and >!).

Q: Q: You said `getElementsByTagName` returns a list. Do you mean an array?

A: *It returns an object that you can treat like an array, but it's actually an object called a `NodeList`. A `NodeList` is a collection of `Nodes`, which is just a technical name for the element objects that you see in the DOM tree. You can iterate over this collection by getting its length using the `length` property, and then access each item in the `NodeList` using an index with the bracket notation, just like an array. But that's pretty much where the similarities of a `NodeList` and an array end, so beyond this, you'll need to be careful in how you deal with the `NodeList` object. You typically won't need to know more about `NodeList` until you want to start adding and removing elements to and from the DOM.*

Q: Q: So I can assign a click handler to any element?

A: *Pretty much. Take any element on the page, get access to it, and assign a function to its `onclick` property. Done. As you've seen, that handler might be specific to that one element, or you might reuse a handler for events on many elements. Of course elements that don't have a visual presence in your page, like the `<script>` and `<head>` elements, won't support events like the click event.*

Q: Q: Do handler functions ever get passed any arguments?

A: *Ah, good question, and very timely. They do, and we're just about to look at the event object that gets passed to some handlers.*

Q: Q: Do elements support other types of events? Or is the click the only one?

A: *There are quite a few others; in fact, you've already seen another one in the code of the battleship game: the `keypress` event. There, an event handler function was called whenever the user pressed Enter from the form input. We'll take a look at a few other event types in this chapter.*



Okay Judy, now we have a single event handler, `showAnswer`, to handle clicks on all the images. You said you know how to tell which image was clicked on when `showAnswer` is called?

Judy: Yes I do. Whenever the click event handler is called, it's passed an *event object*. You can use that object to find out details about the event.

Joe: Like which image was clicked on?

Judy: Well, more generally, the element on which the event occurred, which is known as the *target*.

Joe: What's the target?

Judy: Like I said, it's the element that generated the event. Like if you click on a specific image, the target will be that image.

Joe: So if I click on the image with the id "zero", then the target will be set to that image?

Judy: More precisely, the element object that represents that image.

Joe: Come again?

Judy: Think of the element object as exactly the same thing you get if you call `document.getElementById` with a value of "zero". It's the object that represents the image in the DOM.

Joe: Okay, so how do we get this target? It sounds like that's what we need to know which image was clicked on.

Judy: The target is just a property of the event object.

Joe: Great. That sounds perfect for `showAnswer`. We'll be done in a snap...
Wait, so `showAnswer` is passed the event object?

Judy: That's right.

Joe: So how did our code for the `showAnswer` function work up until now? It's being passed this event object, but we don't have a parameter defined for the event object in the function!

Judy: Remember, JavaScript lets you ignore parameters if you want.

Joe: Oh, right.

Judy: Now Joe, don't forget, you'll need to figure out how to change the `src` of the image to the correct name for the unblurred version. Right now we're assuming the name of the unblurred image is "zero.jpg", but that won't work any more.

Joe: Maybe we can use the `id` attribute of the image to figure out the unblurred image name. The ids of all the images match the names of the unblurred version of each image.

Judy: Sounds like a plan!

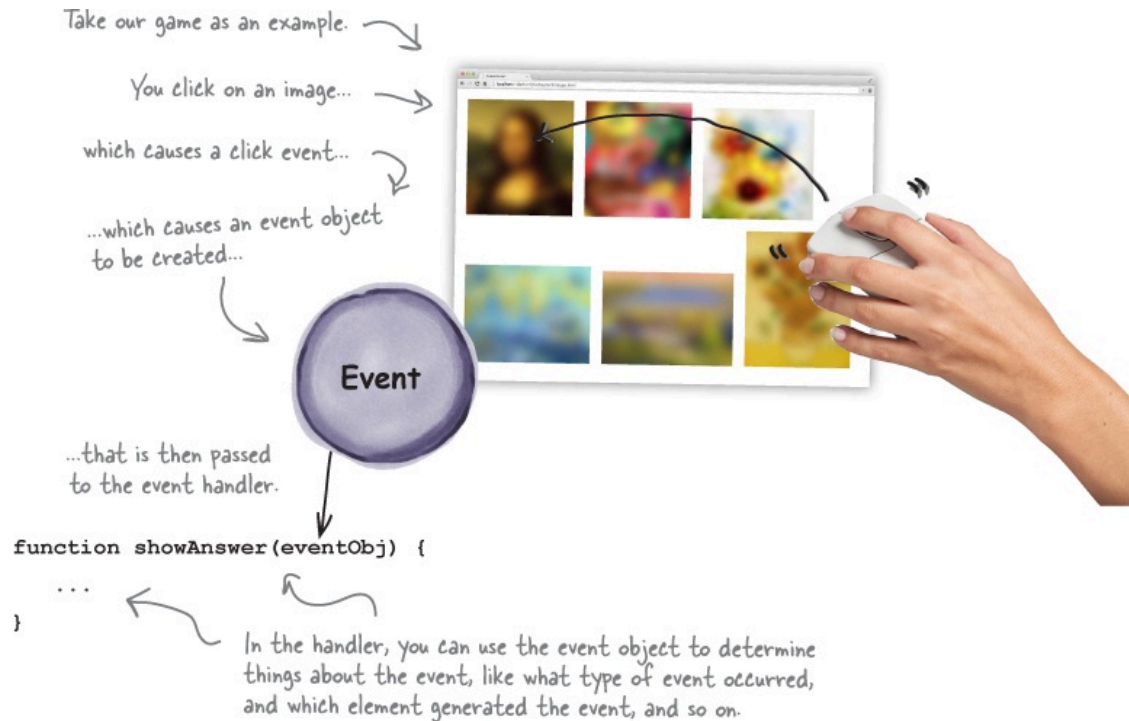
How the event object works

When the click handler is called, it's passed an *event object*—and in fact, for most of the events associated with the document object model (DOM) you'll be passed an event object. The event object contains general information about the event, such as what element generated the event and what time the event happened. In addition, you'll get information specific to the event, so if there was a mouse click, for instance, you'll get the coordinates of the click.

NOTE

There are other kinds of events too (that is, other than DOM events), and we'll see an example later in the chapter...

Let's step through how event objects work:



So, what is *in* an event object? Like we said, both general and specific information about the event. The specific information depends on the type of the event, and we'll come back to that a bit later. The general information includes the `target` property that holds a reference to the object that generated the event. So, if you click on a page element, like an image, that's the target, and we can access it like this:

```
function showAnswer(eventObj) {  
  var image = eventObj.target;  
}
```

The target tells us what element generated the event.

WATCH IT!

If you're running IE8 or older, check the appendix.

With older versions of IE, you need to set up the event object a little differently.

You’ve already seen that the event object (for DOM events) has properties that give you more information about the event that just happened. Below you’ll find other properties that the event object can have. Match each event object property to what it does.

target	Want to know how far from the top of the browser window the user clicked? Use me.
type	I hold the object on which the event occurred. I can be different kinds of objects, but most often I’m an element object.
timeStamp	Using a touch device? Then use me to find out how many fingers are touching the screen.
keyCode	I’m a string, like “click” or “load”, that tells you what just happened.
clientX	Want to know when your event happened? I’m the property for you.
clientY	Want to know how far from the left side of the browser window the user clicked? Use me.
touches	I’ll tell you what key the user just pressed.

Putting the event object to work

So, now that we’ve learned a little more about events—or more specifically, how the event object is passed to the click handler—let’s figure out how to use the information in the event object to deblur any image on the page. We’ll start by revisiting the HTML markup.

<!doctype html> Here's the HTML again.

```
...
<body>
  
  
  
  
  
  
</body>
</html>
```

Each of the images has an id, and the id corresponds to the unblurred image name. So the image with id "zero" has an unblurred image of "zero.jpg". And the image with id "one" has an unblurred image of "one.jpg" and so on...

Notice that the value of each image's id corresponds to the name of the unblurred image (minus the ".jpg" extension). Now, if we can access this id, then we can simply take that name and add on ".jpg" to create the name of the corresponding unblurred image. Once we have that we can change the image `src` property to the unblurred version of the image. Let's see how:

Remember you're getting passed an event object each time an image is clicked on.

```
function showAnswer(eventObj) {
  var image = eventObj.target;

  var name = image.id;
  name = name + ".jpg";
  image.src = name;
}
```

The event object's target property is a reference to the image element that was clicked.

We can then use the id property of that object to get the name of the unblurred image.

And finally, we'll set the src of the image to that name.

As you know, once you change the src property of the image, the browser will immediately retrieve that new image and display it in the page in place of the blurred version.

Test drive the event object and target



Make sure you've updated all the code in your "image.html" file, and take it for a test drive. Guess the image, click, and see the unblurred version be revealed. Think about how this app is designed not as a top-to-bottom running program, but purely as a set of actions that result from an event being generated when you click on an image. Also think about how you

handled the events on all the images with one piece of code that's smart enough to know which image was clicked on. Play around. What happens if you click twice? Anything at all?



THERE ARE NO DUMB QUESTIONS

Q: **Q: Does the onload event handler get passed an event object too?**

A: *It does, and it includes information like the target, which is the window object, the time it happened, and the type of the event, which is just the type "load". It's safe to say you don't typically see the event object used much in load handlers because there really isn't anything that is useful in it for this kind of event. You're going to find that sometimes the event object will be useful to you, and sometimes it won't, depending on the type of event. If you're unsure what the event object contains for a specific kind of event, just grab a JavaScript reference.*

BRAIN POWER

What if you want to have an image become blurred again a few seconds after you've revealed the answer. How might that work?

This week's interview: Talking to the browser about events

HEAD FIRST: Hey Browser, it's always good to have your time. We know how busy you are.

BROWSER: My pleasure, and you're right, managing all these events keeps me on my toes.

HEAD FIRST: Just how do you manage them anyway? Give us a behind-the-scenes look at the magic.

BROWSER: As you know, events are almost continually happening. The user moves the mouse around, or makes a gesture on a mobile device; things arrive over the network; timers go off... it's like Grand Central. That's a lot to manage.

HEAD FIRST: I would have assumed you don't need to do much unless there happens to be a handler defined somewhere for an event?

BROWSER: Even if there's no handler, there's still work to do. Someone has to grab the event, interpret it, and see if there is a handler waiting for it. If there is, I have to make sure that handler gets executed.

HEAD FIRST: So how do you keep track of all these events? What if lots of events are happening at the same time? There's only one of you after all.

BROWSER: Well, yes, lots of events can happen over a very short amount of time, sometimes too fast for me to handle all in real time. So what I do is throw them all on a queue as they come in. Then I go through the queue and execute handlers where necessary.

HEAD FIRST: Boy, that sounds like my days as a short-order cook!

BROWSER: Sure, if you had orders coming in every millisecond or so!

HEAD FIRST: You have go through the queue one by one?

BROWSER: I sure do, and that's an important thing to know about JavaScript: there's one queue and one "thread of control," meaning there is only one of me going through the events one at a time.

HEAD FIRST: What does that really mean for our readers learning JavaScript?

BROWSER: Well, say you write a handler and it requires a lot of computation—that is, something that takes a long time to compute. As long as your handler is chugging along computing, I’m sitting around waiting until it’s done. Only then can I continue with the queue.

HEAD FIRST: Oh wow. Does that happen a lot, that is, you end up waiting on slow code?

BROWSER: It happens, but it also doesn’t take long for a web developer to figure out the page or app isn’t responsive because the handlers are slow. So, it’s not a common problem as long as the web developers know how event queues work.

HEAD FIRST: And now all our readers do! Now, back to events, are there lots of different kinds of events?

BROWSER: There are. We’ve got network-based events, timer events, DOM events related to the page and a few others. Some kinds of events, like DOM events, generate event objects that contain a lot more detail about the event—like a mouse click event will have information about where the user clicked, and a keypress event will have information about which key was pressed, and so on.

HEAD FIRST: So, you spend a lot of time dealing with events. Is that really the best use of your time? After all, you’ve got to deal with retrieving, parsing and rendering pages and all that.

BROWSER: Oh, it’s very important. These days you’ve got to write code that makes your pages interactive and engaging, and for that you need events.

HEAD FIRST: Oh for sure, the days of simple pages are gone.

BROWSER: Exactly. Oh shoot, this queue is about to overflow. Gotta run!

HEAD FIRST: Okay... until next time!

Events and queues

You already know that the browser maintains a queue of events. And that behind the scenes the browser is constantly taking events off that queue

and processing them by calling the appropriate event handler for them, if there is one.

It's important to know that the browser processes these events one at a time, so, where possible, you need to keep your handlers short and efficient. If you don't, the whole event queue could stack up with waiting events, and the browser will get backed up dealing with them all. The downside to you? Your interface could really start to become slow and unresponsive.

NOTE

If things get really bad you'll get the slow script dialog box, which means the browser is giving up!

Ahoy matey! You've got a treasure map in your possession and we need your help in determining the coordinates of the treasure. To do that you're going to write a bit of code that displays the coordinates on the map as you pass the mouse over the map. We've got some of the code on the next page, but you'll have to help finish it.



The mousemove event notifies your handler when a mouse moves over a particular element. You set up your handler using the element's **onmousemove** property. Once you've done that you'll be passed an event object that provides these properties:

clientX, clientY: the x (and y) position in pixels of your mouse from the left side (and top) of the browser window.

screenX, screenY: the x (and y) position in pixels of your mouse from the left side (and top) of the user's screen.

pageX, pageY: the x (and y) position in pixels of your mouse from the left side (and top) of the browser's page.

Blimey! The code is below. So far it includes the map in the page and creates a paragraph element to display the coordinates. You need to make all the event code work. Good luck. We don't want to see you go to Davy Jones' locker anytime soon...

So far we've seen three types of events: the load event, which occurs when the browser has loaded the page; the click event, which occurs when a user clicks on an element in the page and the mousemove event, which occurs when a user moves the mouse over an element. You're likely to run into many other kinds of events too, like events for data arriving over the network, events about the geolocation of your browser, and time-based events (just to name a few).

For all the events you've seen, to wire up a handler, you've always assigned the handler to some property, like `onload`, `onmouseover` or `onclick`. But not all events work like this—for example, with time-based events, rather than assigning a handler to a property, you call a function, `setTimeout`, instead and pass it your handler.

Here's an example: say you want your code to wait five seconds before doing something. Here's how you do that using `setTimeout` and a handler:

BRAIN POWER

How would you make the alert appear every 5 seconds, over and over?

Test drive your timer

Don't just sit there! It's time to test this code! Throw this code into a basic HTML page, and load the page. At first you won't see anything, but after five seconds you'll see the alert.

How setTimeout works

Let's step through what just happened.

1. When the page loads we do two things: we define a handler named `timerHandler`, and we call `setTimeout` to create a time event that will be generated in 5000 milliseconds. When the time event happens, the handler will be executed.
2. The browser continues its normal job as the timer counts down in milliseconds.

3. When the browser's countdown gets to zero, the browser calls the handler.

4. The handler has been called. I'm finished with that timer.

Good catch! Remember we said up front that in this chapter you're going to feel like you aren't in Kansas anymore? Well this is that point in the movie where everything goes from black and white to color. Back to your question; yes, we defined a function and then took that function, and passed it to `setTimeout` (which is actually a method).

Why would we do this and what does it mean? Let's think through this: the `setTimeout` function essentially creates a countdown timer and associates a handler with that timer. That handler is called when the timer hits zero. Now to tell `setTimeout` what handler to call, we need to pass it *a reference to the handler function*. `setTimeout` stores the reference away to use later when the timer has expired.

If you're saying "That makes sense," then great. On the other hand, you might be saying "Excuse me? Pass a function to a function? Say what?" In that case, you probably have experience with a language like C or Java, where *you don't just go around passing functions to other functions like this...* well, in JavaScript, you do, and in fact, being able to pass functions around is incredibly powerful, especially when we're writing code that reacts to events.

More likely at this point you're saying, "I think I sort of get it, but I'm not sure." If so, no worries. For now, just think of this as giving `setTimeout` a reference to the handler it's going to need to invoke when the timer expires. We're going to be talking a lot more about functions and what you can do with them (like passing them to other functions) in the next chapter. So just go with it for now.

EXERCISE

Take a look at the code below and see if you can figure out what `setInterval` does. It's similar to `setTimeout`, but with a slight twist. Check your answer at the end of the chapter.

JAVASCRIPT CONSOLE

Tick

Tick

Tick

Tick

Tick

Tick

Tick

Tick

Here's the output.

Q: Q: Is there a way to stop `setInterval`?

A: *There is. When you call `setInterval`, it returns a timer object. You can pass that timer object to another function, `clearInterval`, to stop the timer.*

Q: Q: You said `setTimeout` was a method, but it looks like a function. Where's the object it's a method of?

A: *Good catch. Technically we could write `window.setTimeout`, but because the window object is considered the global object, we can omit the object name, and just use `setTimeout`, which we'll see a lot in practice.*

Q: Q: Can I omit window on the `window.onload` property too?

A: *You can, but most people don't because they are worried `onload` is a common enough property name (other elements can have the `onload` property too) that not specifying which `onload` property might be confusing.*

Q: Q: With `onload` I'm assigning one handler to an event. But with `setTimeout`, I seem to be able to assign as many handlers as I want to as many timers as I want?

A: *Exactly. When you call `setTimeout`, you are creating a timer and associating a handler with it. You can create as many timers as you like. The browser keeps track of associating each timer with its handler.*

Q: Q: Are there other examples of passing functions to functions?

A: *Lots of them. In fact, you'll find that passing around functions is fairly common in JavaScript. Not only do lots of built-in functions, like `setTimeout` and `setInterval`, make use of function passing, but you'll also discover there's a lot of code you'll write yourself that accepts functions as arguments. But that's only part of the story, and in the next chapter we're going to dive deep into this topic and discover that you can do all sort of interesting things with functions in JavaScript.*

Joe: That sounds cool... I bet you're using `setTimeout`?

Frank: That's my plan, although I'm not sure how to know which image to reblur.

Jim: What do you mean?

Frank: I have my code so that when you click on an image and reveal it, I set up a timer that fires in two seconds. When the timeout event fires, it calls a new handler I wrote called `reblur`.

Joe: And in `reblur`, you need to know which image to reblur?

Frank: Right. I'm not passing any arguments to the handler, it's just being called by the browser when the time expires, and so I have no way to tell my handler the correct image to reblur. I'm kinda stuck.

Jim: Have you looked at the `setTimeout` documentation?

Frank: No, I know only what Judy told me: that `setTimeout` takes a function and a time duration in milliseconds.

Jim: You can add an argument to the call to `setTimeout` that is passed on to the handler when the time event fires.

Frank: Oh that's perfect. So I can just pass in a reference to the correct image to reblur and that will get passed on to the handler when it is called?

Jim: You got it.

Frank: See what a little talking through code gets ya Joe?

Joe: Oh for sure. Let's give this a try...

Finishing the image game

Now it's time to put the final polish on the image game. What we want is for an image to automatically reblur a few seconds after it's revealed. And, as we just learned, we can pass along an argument for the event handler when we call `setTimeout`. Let's check out how to do this:

WATCH IT!

setTimeout does not support extra arguments in IE8 and earlier.

That's right. This code is not going to work for you or your users if you're using IE8 or earlier. But you shouldn't be using IE8 for this book anyway! That said, you'll see another way to do this a little later in the book that will take care of this for IE8 (and earlier).

Test driving the timer

That wasn't much code to add, but it sure makes a big difference in how the image game works. Now when you click, behind the scenes, the browser (through the timer events) is tracking when it needs to call the `reblur` handler, which blurs the image again. Note how *asynchronous* this feels—you're in control of when the images are clicked, but behind-

the-scenes code is being invoked at various times based on the click event and on timer events. There's no über algorithm driving things here, controlling what gets called and when; it's just a lot of little pieces of code that set up, create and react to events.

Q: Q: Can I pass just one argument to the `setTimeout` handler?

A: *No you can actually pass as many as you like: zero, one or more.*

Q: Q: What about the event object? Why doesn't `setTimeout` pass the event handler one?

A: *The event object is mostly used with DOM-related event handlers. `setTimeout` doesn't pass any kind of event object to its handler, because it doesn't occur on a specific element.*

Q: Q: `showAnswer` is a handler, and yet it sets up a new handler, `reblur`, in its code. Is that right?

A: *You've got it. You'll actually see this fairly often in JavaScript. It's perfectly normal to see a handler set up additional event handlers for various events. And this is the style of programming we were referring to in the beginning of the chapter: asynchronous programming. To create the image game we didn't just translate an algorithm that runs top down. Rather we're hooking up event handlers to handle the execution of the game as events occur. Trace through a few different examples of clicking on images and the various calls that get made to reveal and reblur the image.*

Q: Q: So there are DOM-based events, and timer events... are there lots of different kinds of events?

A: *Many of the events you deal with in JavaScript are DOM events (like when you click on an element), or timer events (created with `setTimeout` or `setInterval`). There are also API-specific events, like events generated by JavaScript APIs including Geolocation, LocalStorage, Web Workers, and so on (see *Head First HTML5 Programming* for more on these). And finally, there is a whole category of events related to I/O: like when you request data from a web service using `XmlHttpRequest` (again, see *Head First HTML5 Programming* for more), or Web Sockets.*

Judy: To make this work you'll want to make use of the mouseover event. You can set a handler for this event on any element with the `onmouseover` property:

```
myElement.onmouseover = myHandler;
```

Judy: Also, the mouseout event tells you when the mouse leaves your element. You can set its handler with the `onmouseout` property.

EXERCISE

Rework your code so that an image is revealed and reblurred by moving your mouse over and out of the image elements. Be sure to test your code, and check your answer at the end of the chapter:

NOTE

JavaScript code goes here.

With the image game complete, Judy wrote some code to review in the weekly team meeting. In fact, she started a little contest, awarding the first person to describe what the code does with lunch. Who wins? Jim, Joe, Frank? Or you?

```
<!doctype html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Don't resize me, I'm ticklish!</title>
<script>
    function resize() {
        var element = document.getElementById("display");
        element.innerHTML = element.innerHTML + " that tickles!";
    }
</script>
</head>
<body>
<p id="display">
    Whatever you do, don't resize this window! I'm warning you!
</p>
<script>
    window.onresize = resize;
</script>
</body>
</html>
```

NOTE

Put your notes here describing what this code does. What events are involved? How is the handler set up? And when does the event happen? Don't just make notes, give this a try in your browser.

CODE LABORATORY

We've found some highly suspicious code we need your help testing. While we've already done an initial analysis on the code and it looks like 100% standard JavaScript, something about it looks odd. Below you'll find two code specimens. For each specimen you'll need to identify what seems odd about the code, test to make sure the code works, and then try to analyze what exactly it does. Go ahead and make your notes on this page. You'll find our analysis on the next page.

At first glance this code appears to simply define a function that adds the number one to any parameter and return it.

Looking closer, this isn't a normal function definition. Rather, we are declaring a variable and assigning to it a function that appears to be missing its name.

Further, we're invoking the function with the variable name, not a name associated with the function as part of its definition.

Odd indeed (although it reminds us a bit of how object methods are defined).

Here we appear to have something similar. Instead of defining a function separately and assigning its name to the window.onload property, we're assigning a function directly to that property. And again, the function doesn't define its own name.

We added this code to an HTML page and tested it. The code appears to work as you might expect. With specimen #1, when the function assigned to addOne is in-

voked, we get a result that is one greater than the number we pass in, which seems right. With specimen #2, when we load the page, we get the alert “The page is loaded!”.

From these tests it would appear as if functions can be defined without names, and used in places where you’d expect an expression.

- Most JavaScript code is written to react to **events**.
 - There are many different kinds of events your code can react to.
 - To react to an event, you write an **event handler** function, and register it. For instance, to register a handler for the click event, you assign the handler function to the onclick property of an element.
 - You're not required to handle any specific event. You choose to handle the events you're interested in.
 - **Functions** are used for handlers because functions allow us to package up code to be executed later (when the event occurs).
 - Code written to handle events is different from code that executes top to bottom and then completes. Event handlers can run at any time and in any order: they are **asynchronous**.
 - Events that occur on elements in the DOM (DOM events) cause an event object to be passed to the event handler.
 - The **event object** contains properties with extra information about the event, including the type (like "click" or "load") and the target (the object on which the event occurred).
 - Older versions of IE (IE 8 and older) have a different event model from other browsers. See the appendix for more details.
 - Many events can happen very close together. When too many events happen for the browser to handle them as they occur, the events are stored in an **event queue** (in the order in which they occurred) so the browser can execute the event handlers for each event in turn.
 - If an event handler is computationally complex, it will slow down the handling of the events in the queue because only one event handler can execute at a time.
 - The functions **setTimeout** and **setInterval** are used to generate time-based events after a certain time has passed.
 - The method **getElementsByTagName** returns zero, one or more element objects in a NodeList (which is array-like, so you can iterate over it).
-

Practice your event reaction time by doing this crossword.

Across

1. Use this property of the event object to know when an event happened.
3. When you click your mouse, you'll generate a _____ event.
8. Events are handled _____.
12. 5000 milliseconds is _____ seconds.
13. Use this method to get multiple elements from the DOM using a tag name.
14. A function designed to react to an event is called an event _____.
15. The setTimeout method is used to create a _____ event.
16. The browser has only one _____ of control.
17. The browser can only execute one event _____ at a time.
19. The event object for a mouseover event has this property for the X position of the mouse from the left side of the browser window.
20. The event _____ is passed to an event handler for DOM events.
21. To pass an argument to a timer event handler, pass it as the _____ argument to setTimeout.

Down

1. You'll generate this event if you touch your touch screen device.
2. zero.jpg is the _____.
4. When you begin programming with events, you might feel like you're not in _____ any more.
5. JavaScript allows you to pass a _____ to a function.
6. If too many events happen close together, the browser stores the events in an event _____.
7. Events are generated for lots of things, but not for baking _____.
9. To make a time event happen over and over, use _____.
10. The window _____ property is for handling a page loaded event.
11. _____ is super ticklish.
14. To assign an event handler for a time event, pass the _____ to setTimeout as the first argument.
15. How you know which image was clicked on in the image game.
18. A program with code to handle events is not this.

Pick two of the events above. If the browser could notify your code when these events occurred, what cool or interesting code might you write?

Let's take an event that notifies us when a user submits a form. If we're notified of this event, then we could get all the data the user filled into the form and check to make sure it's valid (e.g. the user put something that looks like a phone number into a phone number field, or filled out the required fields). Once we've done that check, then we could submit the form to the server.

How about the mouse movement event? If we're notified whenever a user moves the mouse, then we could create a drawing application right in the browser.

If we're notified when the user scrolls down the page, we could do interesting things like reveal an image as they scroll down.



Below, you'll find the image game code. Your job is to play like you're the browser and to figure out what you need to do after each event. After you've done the exercise look at the end of the chapter to see if you got everything. Here's our solution.

```
window.onload = init;
function init() {
  var image = document.getElementById("zero");
  image.onclick = showAnswer;
}

function showAnswer() {
  var image = document.getElementById("zero");
  image.src = "zero.jpg";
}
```

WHO DOES WHAT? SOLUTION

You've already seen that the event object (for DOM events) has properties that give you more information about the event that just happened. Below you'll find other properties that the event object can have. Match each event object property to what it does.



Ahoy matey! You've got a treasure map in your possession and we need your help in determining the coordinates of the treasure. To do that you're going to write a bit of code that displays the coordinates on the map as you pass the mouse over the map.

Blimey! The code is below. So far it includes the map in the page and creates a paragraph element to display the coordinates. You need to make all the event-based code work. Good luck. We don't want to see you go to Davy Jones' locker anytime soon... And here's our solution.

EXERCISE SOLUTION

Take a look at the code below and see if you can figure out what `setInterval` does. It's similar to `setTimeout`, but with a slight twist. Here's our solution:



EXERCISE SOLUTION

Rework your code so that you can reveal and reblur an image by passing your mouse over and out of the image elements. Be sure to test your code. Here's our solution:



With the image game complete, Judy wrote some code to review in the weekly team meeting. In fact, she started a little contest, awarding the first person to describe what the code does with lunch. Who wins? Jim, Joe, Frank? Or you?

Practice your event reaction time by doing this crossword. Here's our solution.
