Lecture 16
Putting JavaScript to Work: Web Workers

Samkeun Kim <skim@hknu.ac.kr>

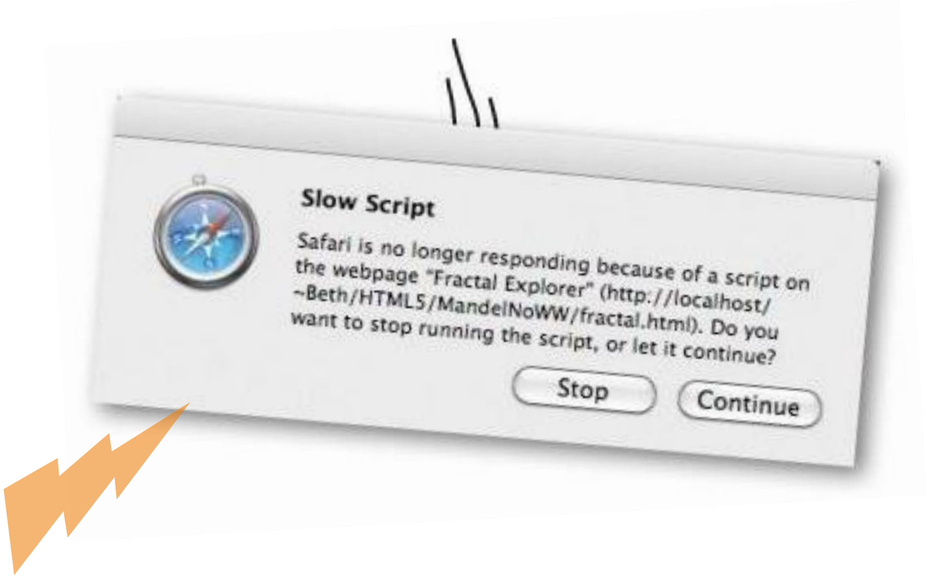
http://cyber.hknu.ac.kr/

국립한경대학교
HANKYONG NATIONAL UNIV.

The Dreaded Slow Script

자바스크립트에 관한 위대한 것 중의 하나가 '**한번에 한 가지 일만 한다**'는 것이다.

이것을 **Single-threaded**라고 부른다 => 프로그래밍을 직관적으로 만든다!!



Slow Script

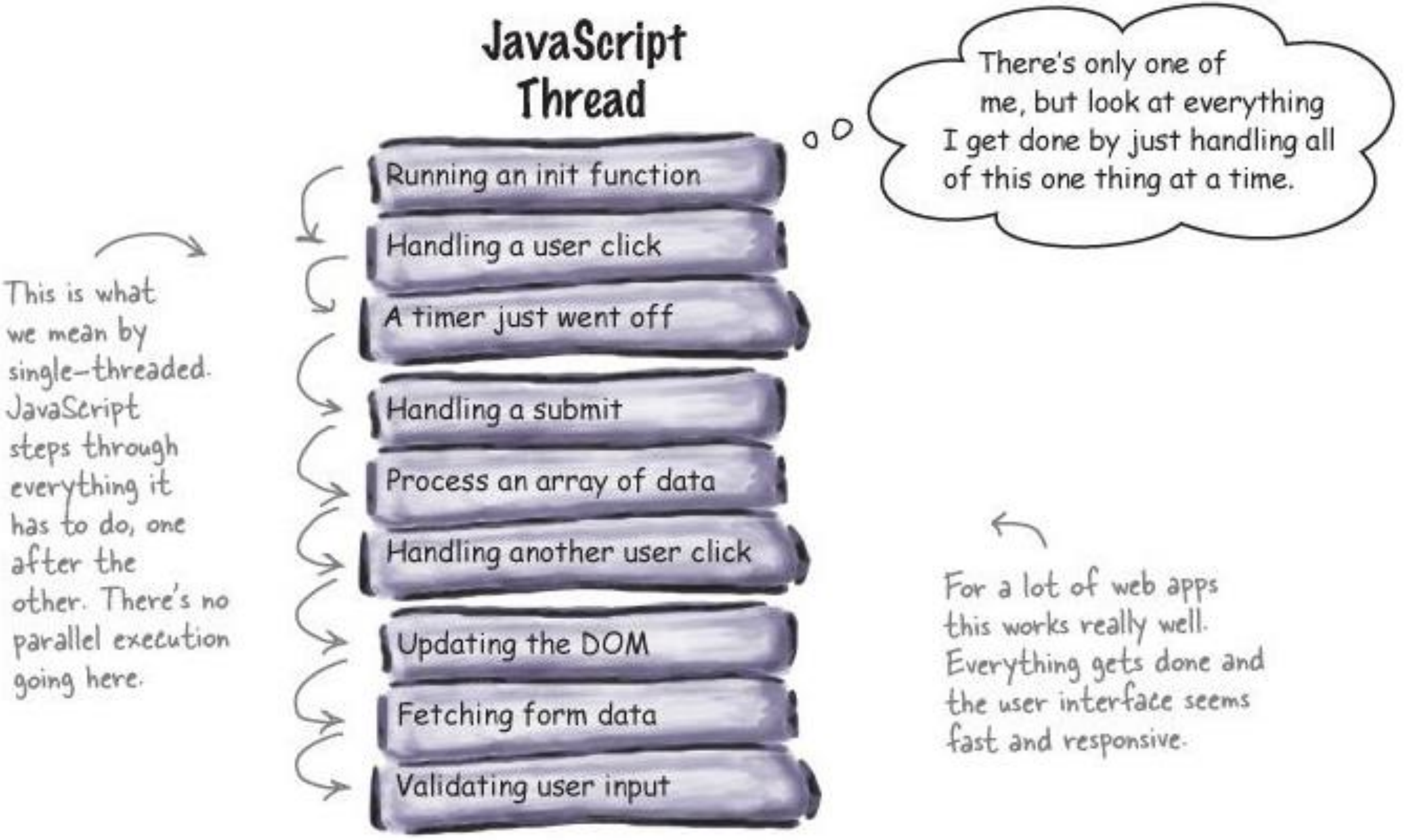
Safari is no longer responding because of a script on the webpage "Fractal Explorer" (http://localhost/~Beth/HTML5/MandelNoWW/fractal.html). Do you want to stop running the script, or let it continue?

Single-threaded 단점:

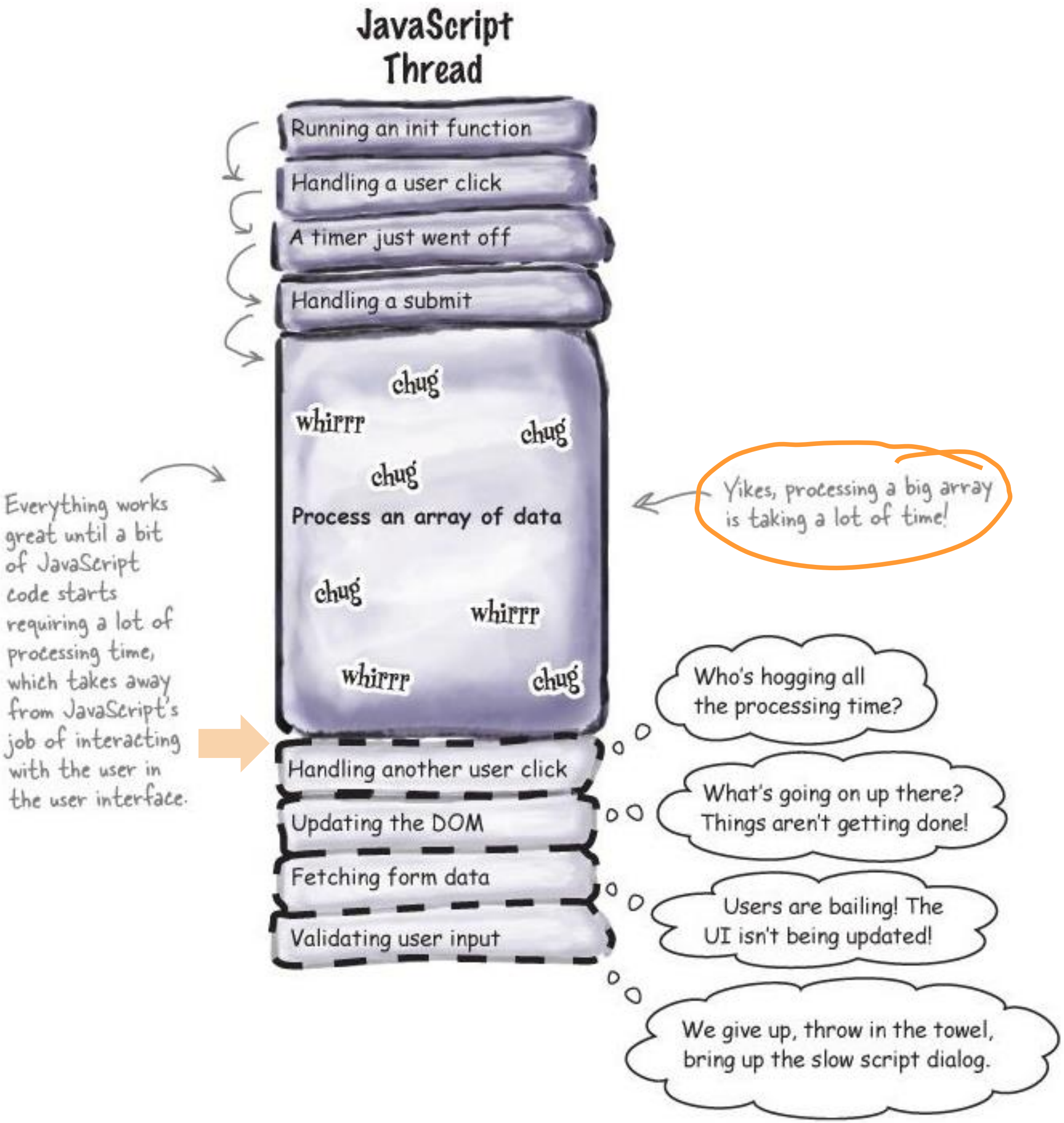
과대한 작업을 지시하면

⇒ 멈춰버리거나 "**slow script**" 다이얼로그 창이 뜬다.

How JavaScript spends its time

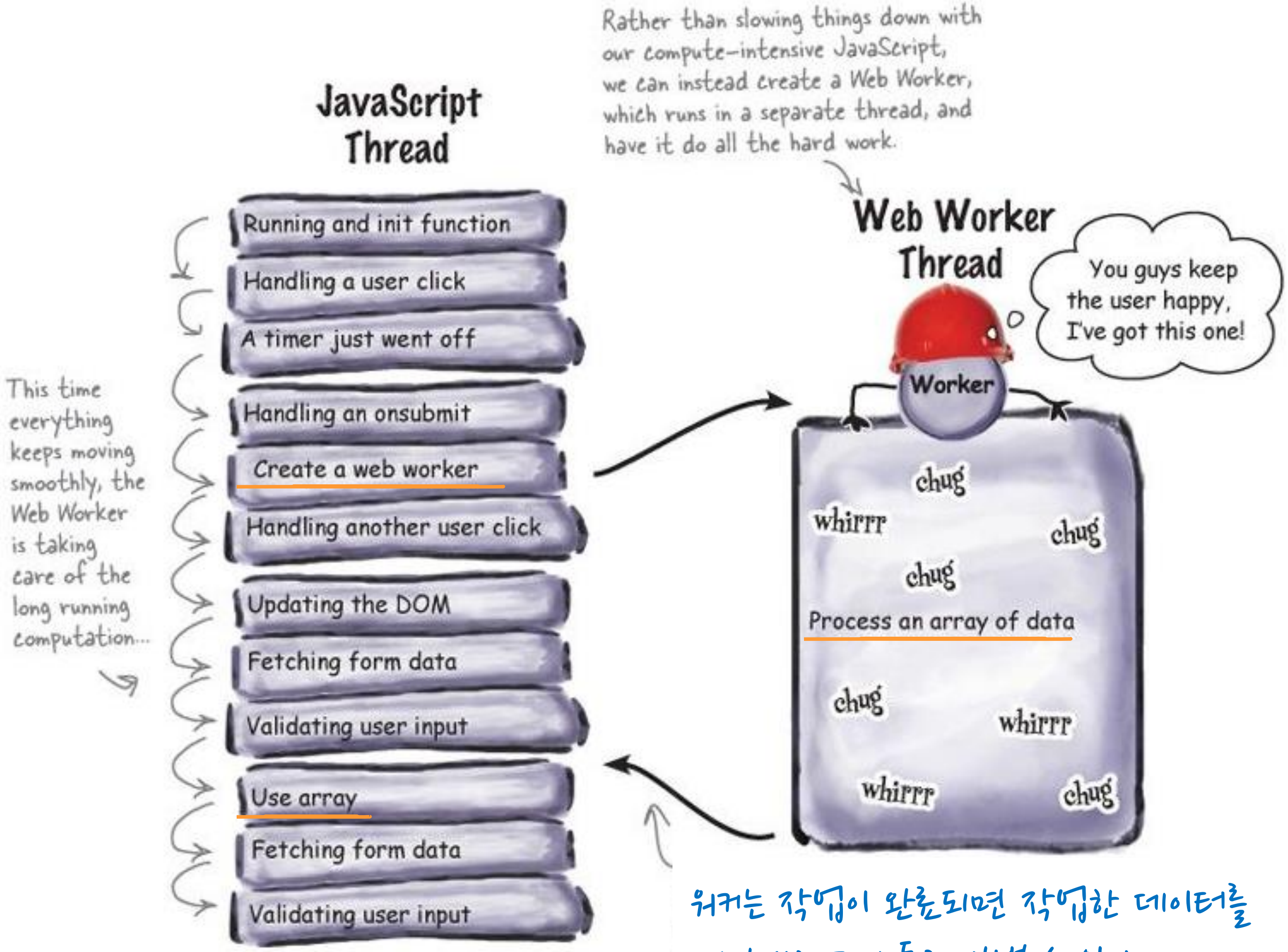

JavaScript Thread
Running an init function
Handling a user click
A timer just went off
Handling a submit
Process an array of data
Handling another user click
Updating the DOM
Fetching form data
Validating user input
There's only one of me, but look at everything I get done by just handling all of this one thing at a time.
This is what we mean by single-threaded. JavaScript steps through everything it has to do, one after the other. There's no parallel execution going here.
For a lot of web apps this works really well. Everything gets done and the user interface seems fast and responsive.

When single-threaded goes BAD


JavaScript Thread
Running an init function
Handling a user click
A timer just went off
Handling a submit
chug
whirrr
chug
chug
Process an array of data
chug
whirrr
whirrr
chug
Yikes, processing a big array is taking a lot of time!
Everything works great until a bit of JavaScript code starts requiring a lot of processing time, which takes away from JavaScript's job of interacting with the user in the user interface.
Handling another user click
Updating the DOM
Fetching form data
Validating user input
Who's hogging all the processing time?
What's going on up there? Things aren't getting done!
Users are bailing! The UI isn't being updated!
We give up, throw in the towel, bring up the slow script dialog.

Adding another thread of control to help

HTML5 이전에는 웹 페이지나 앱들이 오로지 한 개의 쓰레드에 의해 제어되었다.
그러나 이제는 계산 작업을 도와줄 **또 다른 쓰레드를 생성할 방법**을 가지게 되었다.


Rather than slowing things down with our compute-intensive JavaScript, we can instead create a Web Worker, which runs in a separate thread, and have it do all the hard work.
JavaScript Thread
Running and init function
Handling a user click
A timer just went off
Handling an onsubmit
Create a web worker
Handling another user click
Updating the DOM
Fetching form data
Validating user input
Use array
Fetching form data
Validating user input
This time everything keeps moving smoothly, the Web Worker is taking care of the long running computation...
Web Worker Thread
You guys keep the user happy, I've got this one!
Worker
chug
whirrr
chug
chug
Process an array of data
chug
whirrr
whirrr
chug

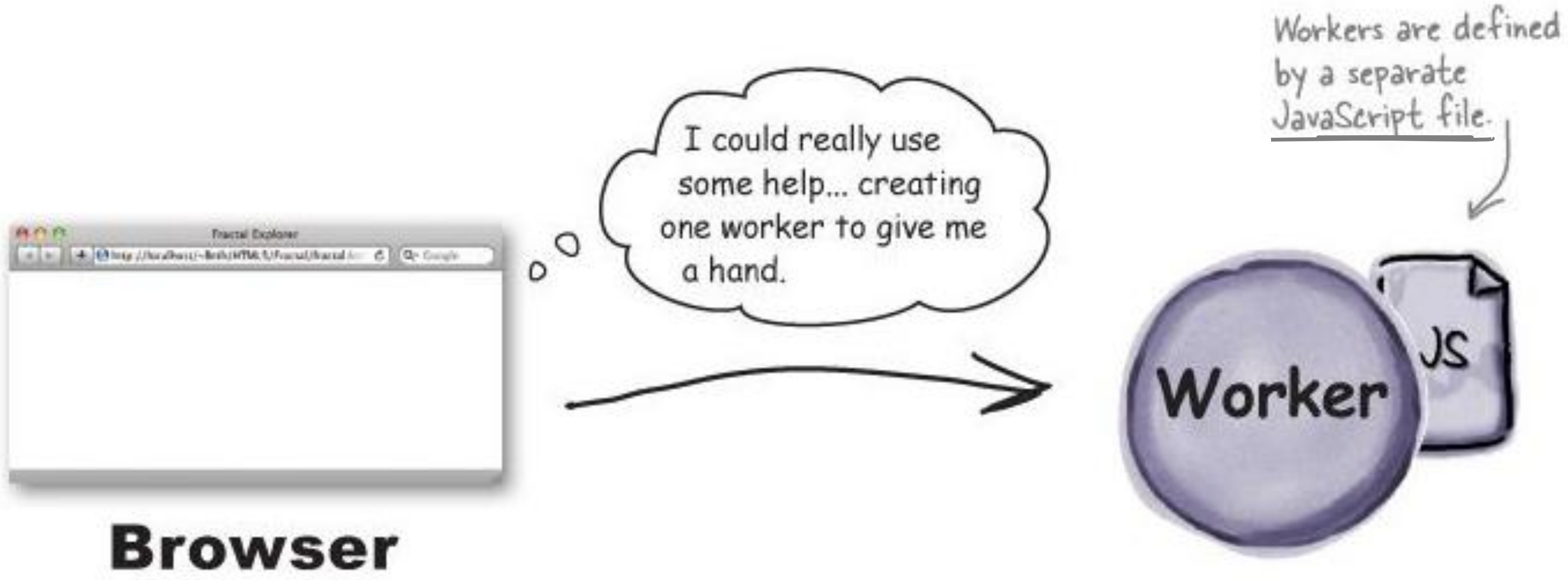
워커는 작업이 완료되면 작업한 데이터를
메인 쓰레드에 돌려 보낼 수 있다.

How Web Workers work

Web Worker 사용

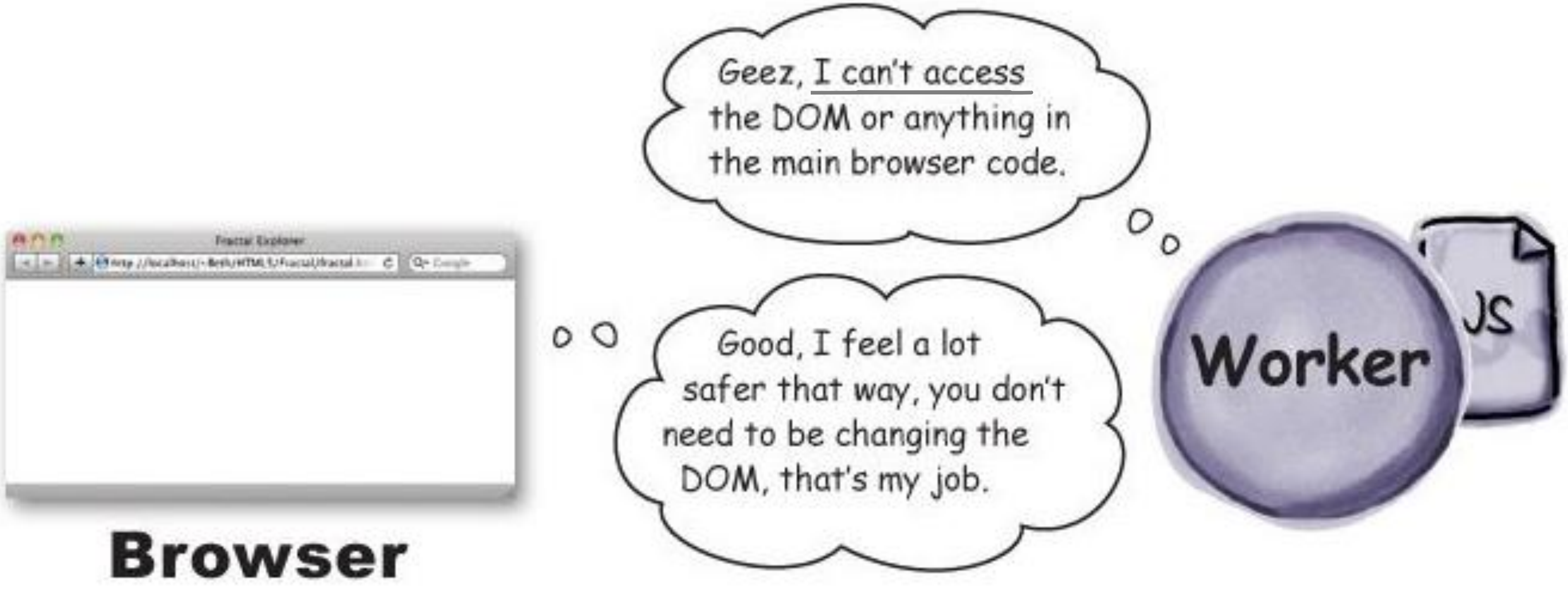
⇒ 브라우저가 먼저 **계산 작업을 돕기 위한 워커들을 생성**

⇒ 각 워커는 작업을 수행하는데 필요한 모든 코드를 포함하는 **자신의 자바스크립트 파일로 정의**


Workers are defined by a separate JavaScript file.
I could really use some help... creating one worker to give me a hand.
Browser
Worker
JS
Fractal Explorer

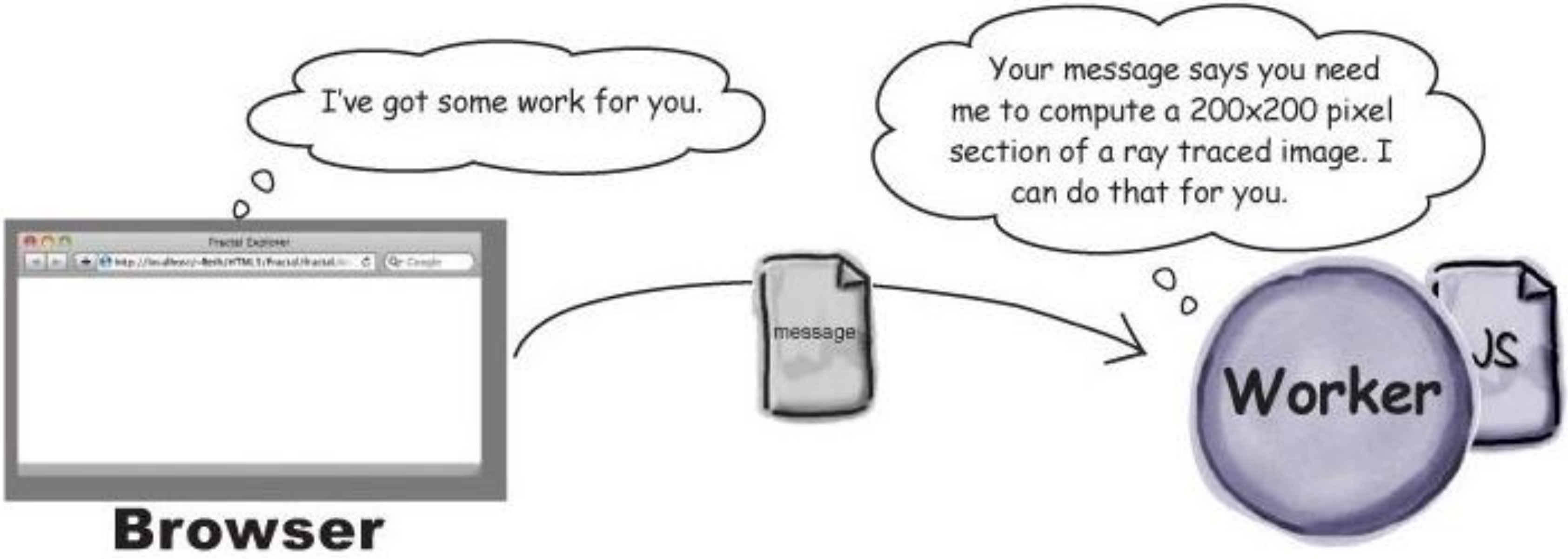
워커는 매우 **제한된 세상**에서 살아간다

그들은 메인 브라우저가 자유롭게 접근할 수 있는 **DOM 또는 변수/함수와 같은 많은 Runtime 객체들을 접근할 수 없다!!**


Geez, I can't access the DOM or anything in the main browser code.
Good, I feel a lot safer that way, you don't need to be changing the DOM, that's my job.
Fractal Explorer
Browser
Worker
JS

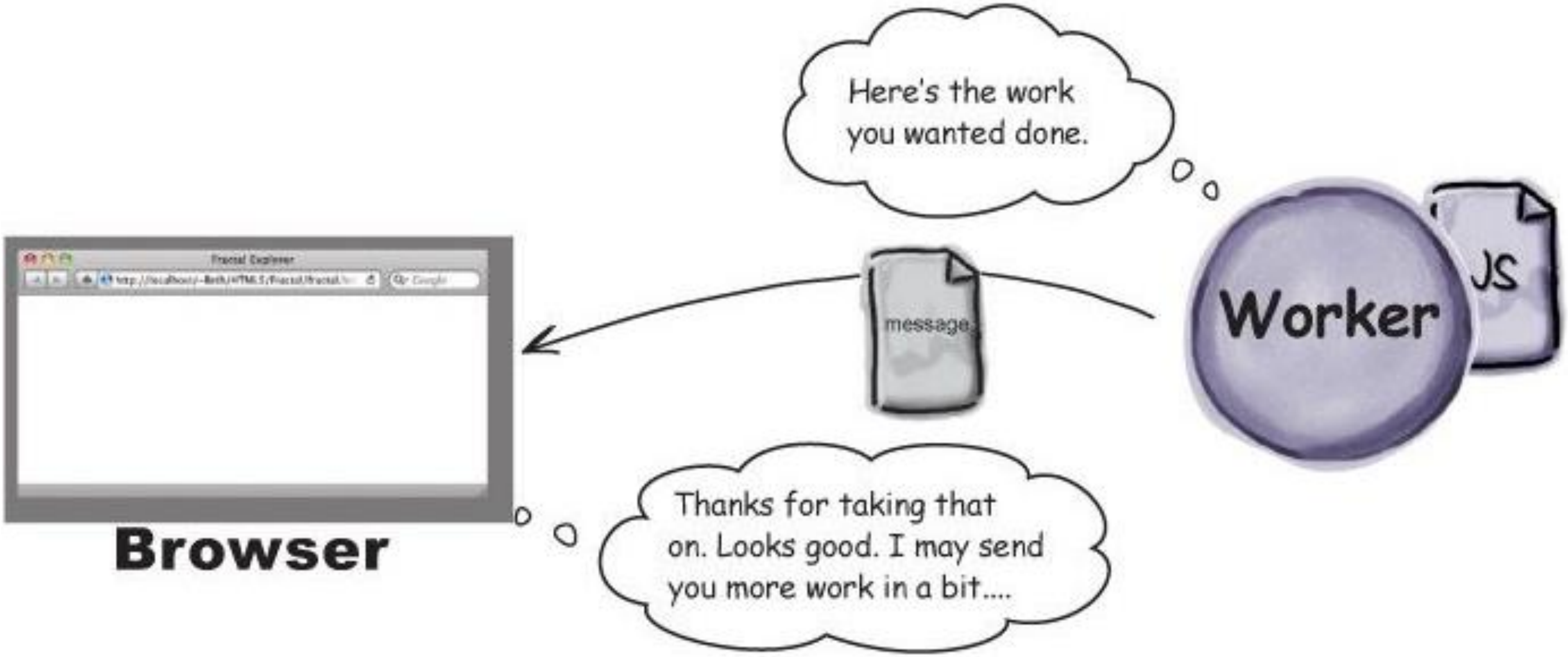
브라우저가 워커에게 작업 시작 지시 => 전형적으로 **워커에게 메시지를 보낸다!**

워커는 메시지를 받아서 자신에게 주어진 특별한 지시를 파악한 후 작업을 시작한다.

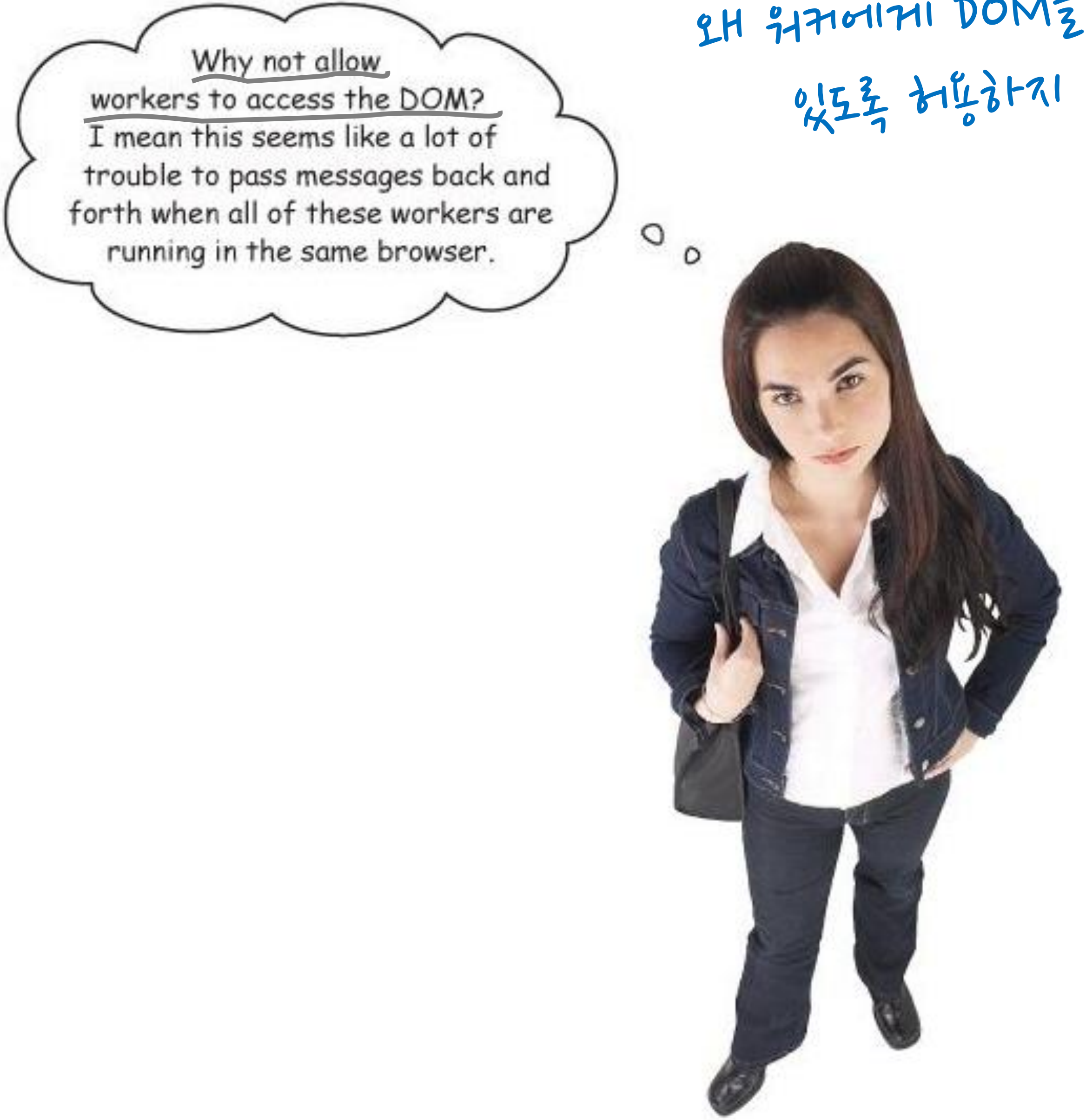

I've got some work for you.
Fractal Explorer
message
Your message says you need me to compute a 200x200 pixel section of a ray traced image. I can do that for you.
JS
Worker
Browser

워커가 작업을 완료 => **작업했던 결과와 함께 메시지를 돌려 보냄**

메인 브라우저 코드는 이러한 결과를 가져와서 적절한 방식으로 **페이지에 통합시킨다.**


Here's the work you wanted done.
message
Worker
JS
Browser
Thanks for taking that on. Looks good. I may send you more work in a bit....

왜 워커에게 DOM을 액세스할 수 있도록 허용하지 않나요?


Why not allow workers to access the DOM? I mean this seems like a lot of trouble to pass messages back and forth when all of these workers are running in the same browser.

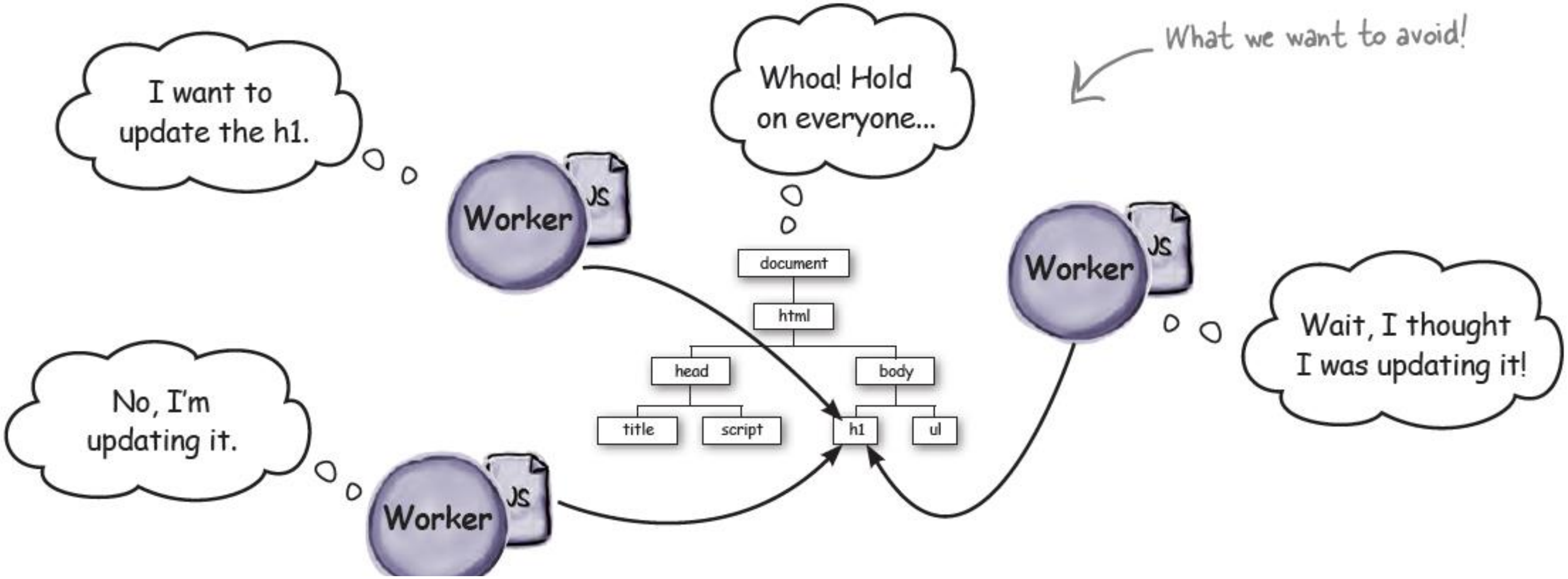
To keep things efficient.

DOM과 JavaScript가 성공해 왔던 한 가지 이유는 **DOM 접근권한을 오로지 하나의 쓰레드에만 국한**시켜서 DOM연산을 최적화시켰기 때문이다.

만일 여러 개의 쓰레드가 동시에 DOM을 변경할 수 있도록 허용한다면 **퍼포먼스에 심각한 영향**을 줄 수 있다.

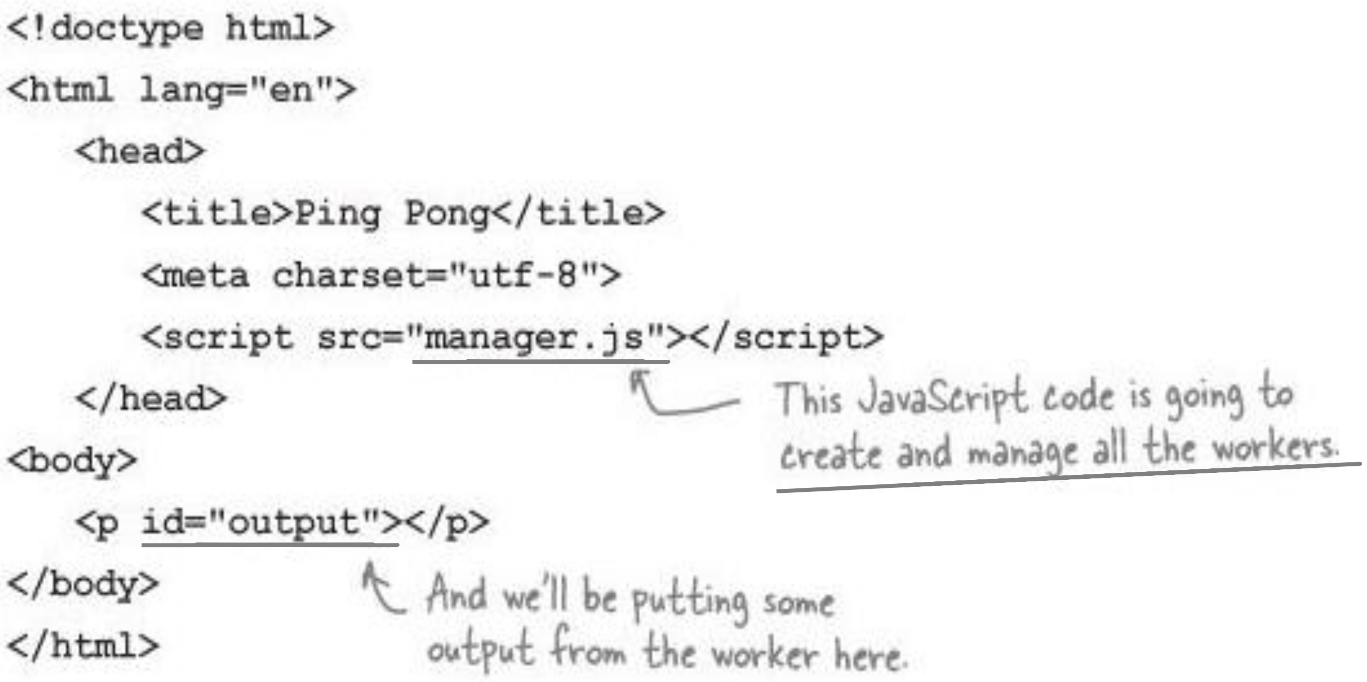
더 심각한 문제는 DOM에 한 무더기의 변경을 동시에 수행한다면 DOM이 쉽게 **불일치 상태**로 될 가능성이 높다.

➔ Bad



Your first Web Worker…

pingpong.html



```
<!doctype html>
<html lang="en">
   <head>
      <title>Ping Pong</title>
      <meta charset="utf-8">
      <script src="manager.js"></script>
   </head>
<body>
   <p id="output"></p>
</body>
</html>
```

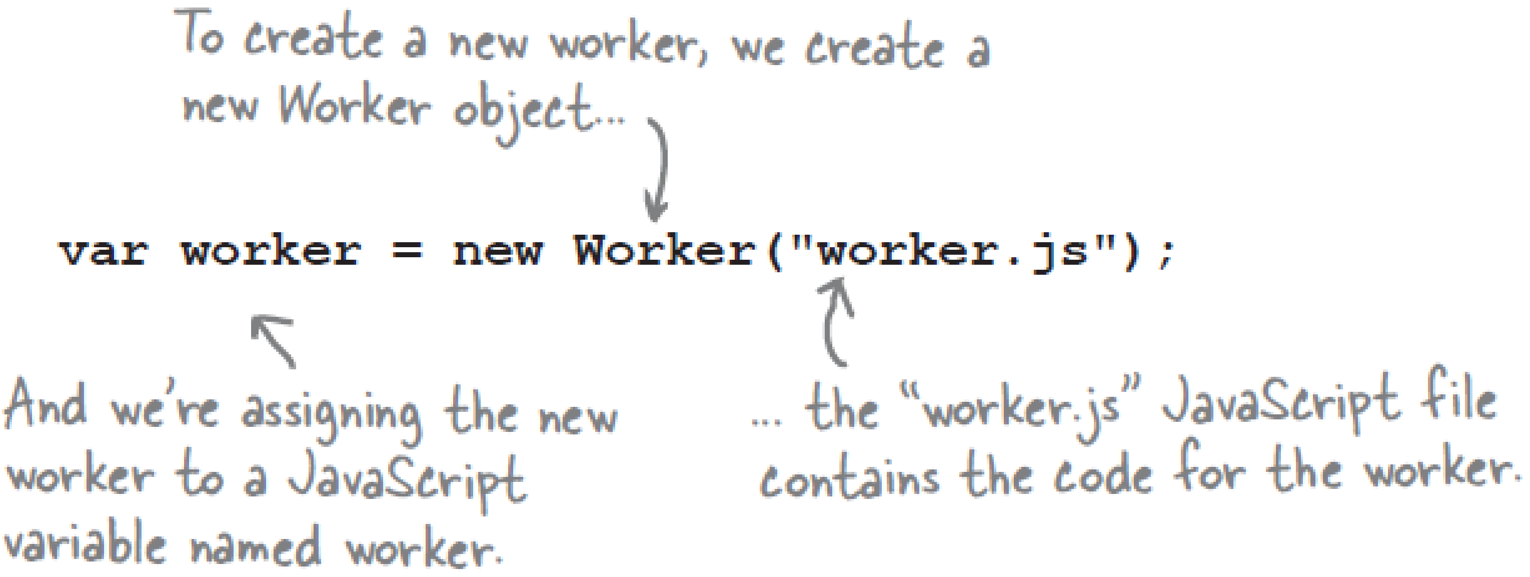
This JavaScript code is going to create and manage all the workers.

And we'll be putting some output from the worker here.

Your first Web Worker…

How to create a Web Worker

manager.js를 구현하기 전에 웹 워커를 실제로 어떻게 생성하는 지를 살펴보자:



To create a new worker, we create a new Worker object…

```
var worker = new Worker("worker.js");
```

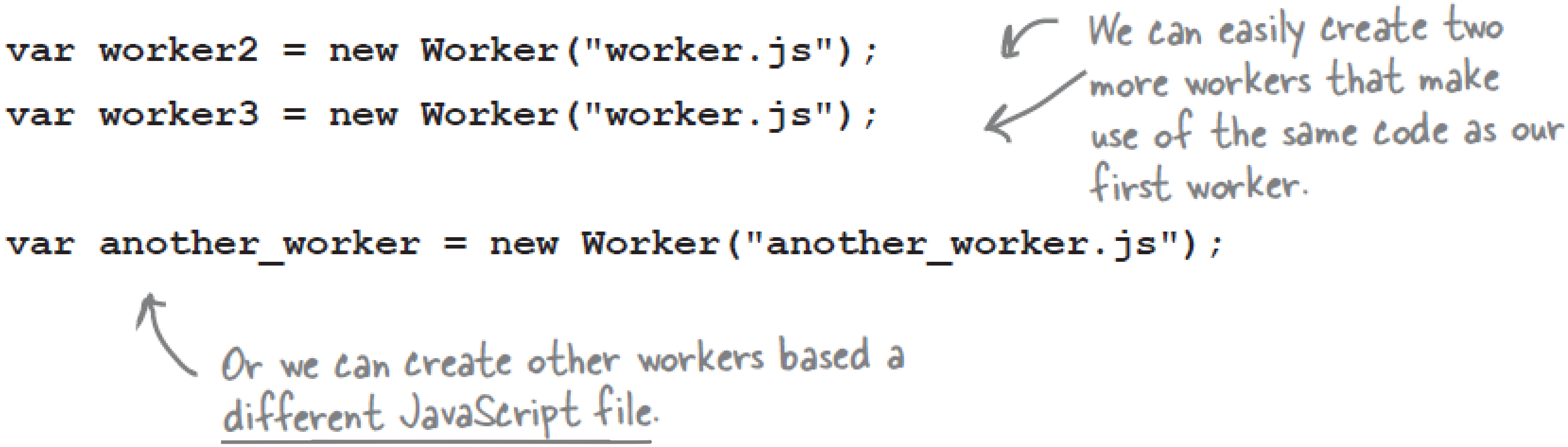
And we're assigning the new worker to a JavaScript variable named worker.

… the "worker.js" JavaScript file contains the code for the worker.

Your first Web Worker…

How to create a Web Worker

원하는 만큼의 워커들을 생성할 수 있다:



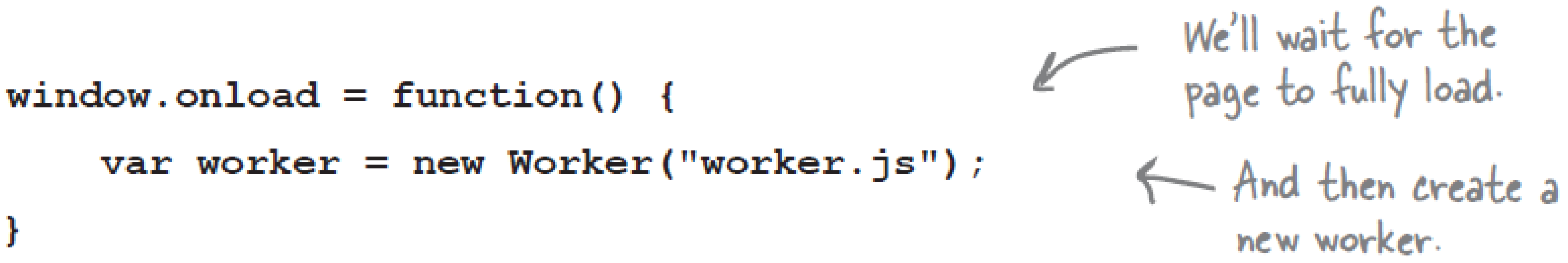
```
var worker2 = new Worker("worker.js");
var worker3 = new Worker("worker.js");

var another_worker = new Worker("another_worker.js");
```

We can easily create two more workers that make use of the same code as our first worker.

Or we can create other workers based a different JavaScript file.

Writing manager.js

지금은 하나의 워커만 생성하자. **manager.js**에 다음 코드를 추가하자:



```
window.onload = function() {
    var worker = new Worker("worker.js");
}
```

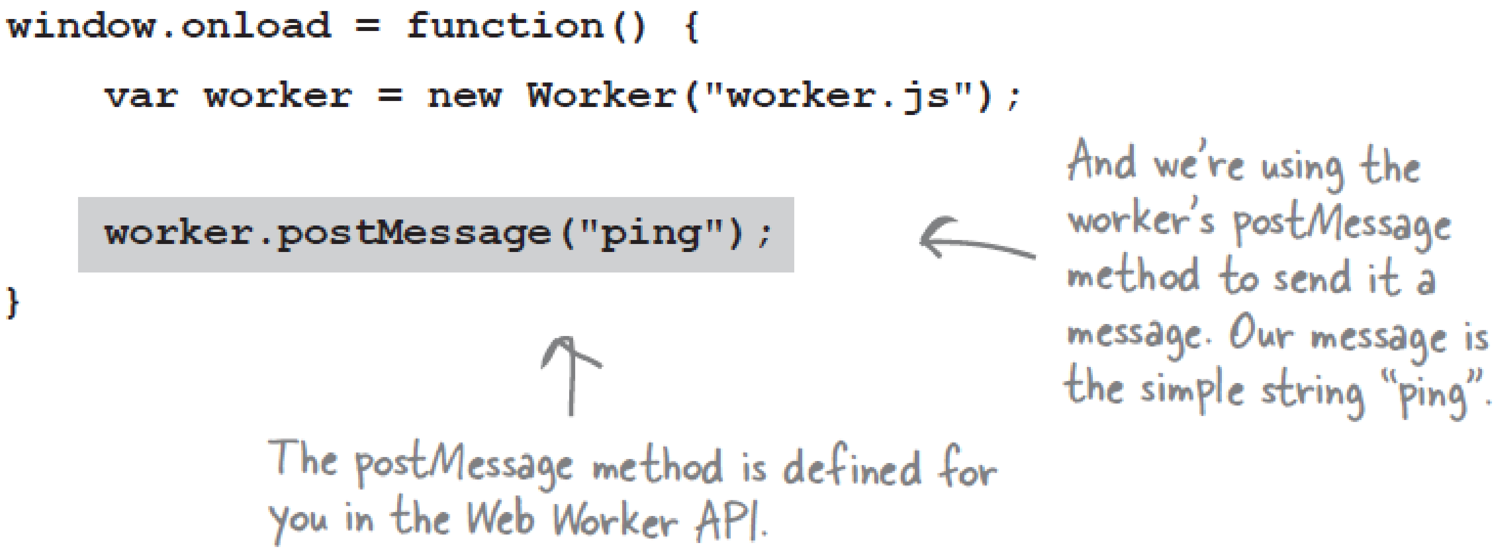
We'll wait for the page to fully load.

And then create a new worker.

Writing manager.js

워커에게 메시지를 보내서 실제로 작업을 시작시키고 싶다.

메시지를 보내기 위해 워커 객체의 **postMessage** 메소드를 사용한다:



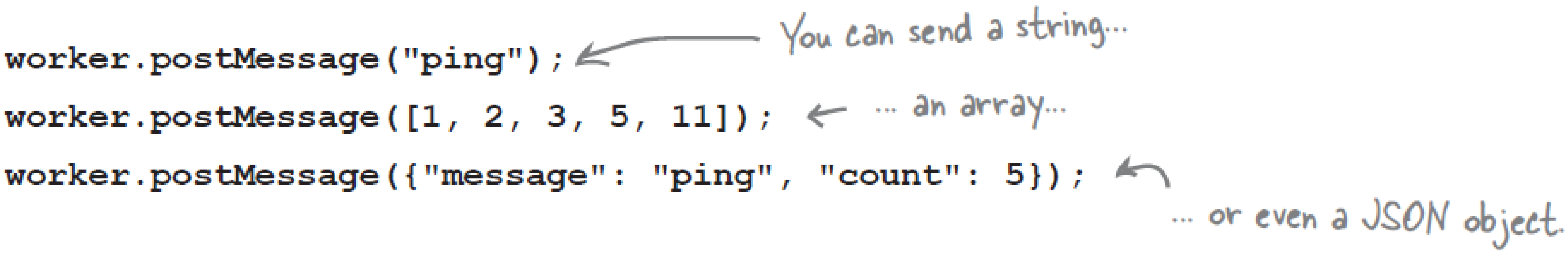
```
window.onload = function() {
    var worker = new Worker("worker.js");

    worker.postMessage("ping");
}
```

And we're using the worker's postMessage method to send it a message. Our message is the simple string "ping".

The postMessage method is defined for you in the Web Worker API.

postMessage Up Close

postMessage에 스트링 이상의 것들을 실어서 보낼 수 있다:



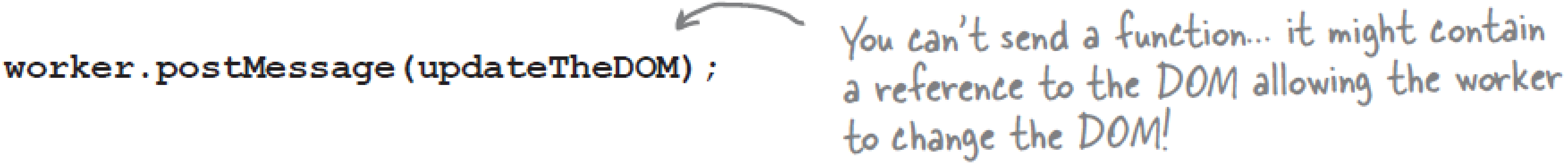
```
worker.postMessage("ping");
worker.postMessage([1, 2, 3, 5, 11]);
worker.postMessage({"message": "ping", "count": 5});
```

You can send a string...

... an array...

... or even a JSON object.

함수는 보낼 수 없다:



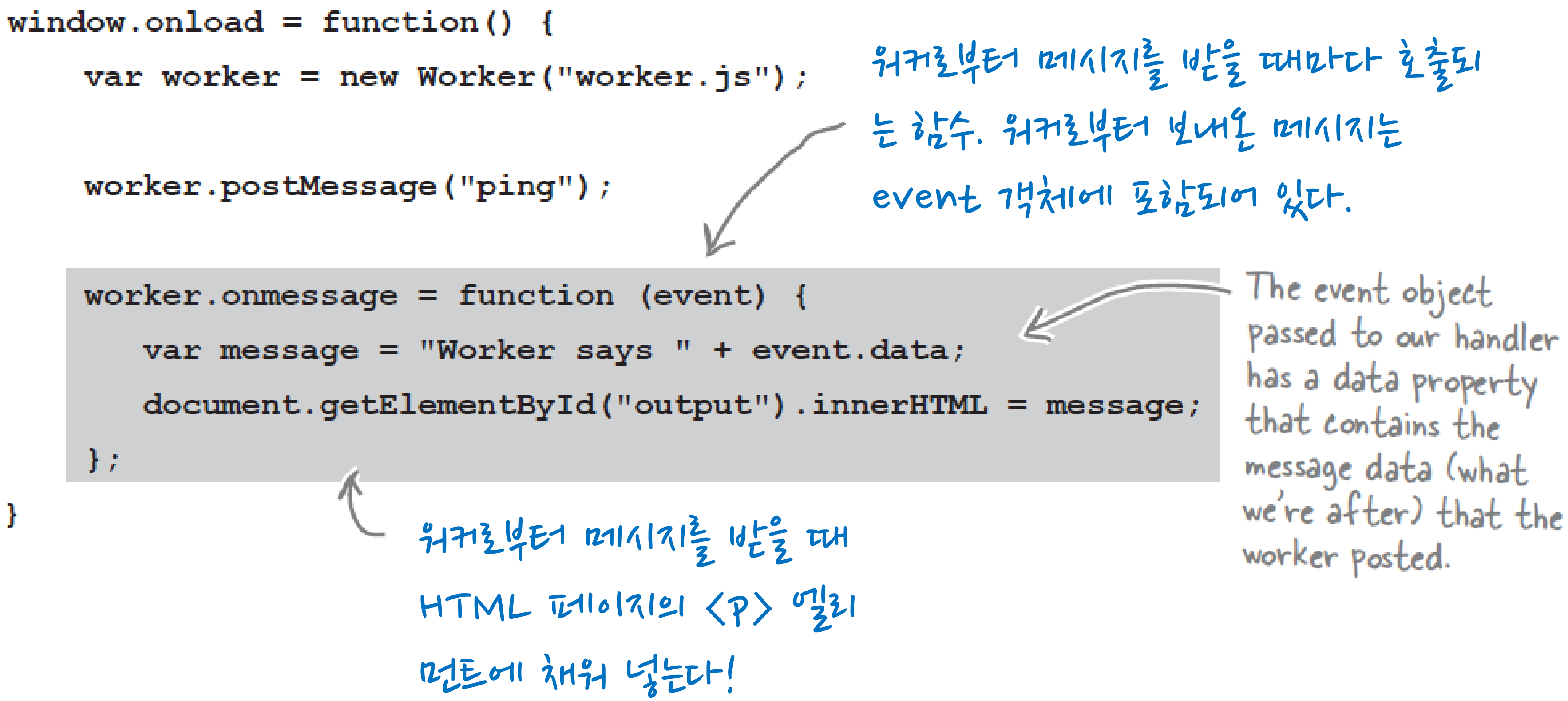
```
worker.postMessage(updateTheDOM);
```

You can't send a function... it might contain a reference to the DOM allowing the worker to change the DOM!

Receiving messages from the worker

manager.js 코드 => 워커로부터 메시지를 받는 일이 아직 남아 있다.

워커의 메시지를 받기 위해 워커의 **onmessage 프로퍼티에 대한 핸들러**를 정의할 필요가 있다:



```
window.onload = function() {
    var worker = new Worker("worker.js");

    worker.postMessage("ping");

    worker.onmessage = function (event) {
        var message = "Worker says " + event.data;
        document.getElementById("output").innerHTML = message;
    };
}
```

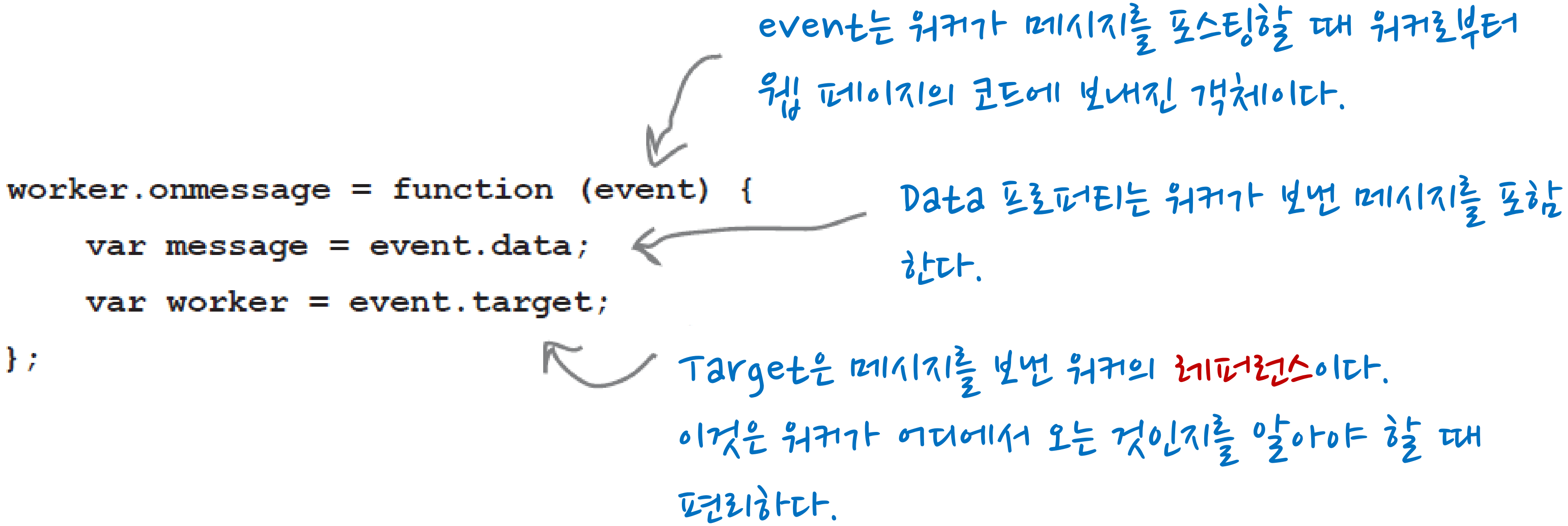
워커로부터 메시지를 받을 때마다 호출되는 함수. 워커로부터 보내온 메시지는 event 객체에 포함되어 있다.

The event object passed to our handler has a data property that contains the message data (what we're after) that the worker posted.

워커로부터 메시지를 받을 때 HTML 페이지의 <p> 엘리먼트에 채워 넣는다!

onmessage Up Close

onmessage 핸들러가 워커로부터 받는 메시지를 간단히 살펴 보자.
앞서 말했듯이 이 메시지는 **Event** 객체에 감싸져 있다.
Event 객체는 **data**와 **target**이라는 두 가지 프로퍼티를 가지고 있다:



```
worker.onmessage = function (event) {
    var message = event.data;
    var worker = event.target;
};
```

event는 워커가 메시지를 포스팅할 때 워커로부터 웹 페이지의 코드에 보내진 객체이다.

Data 프로퍼티는 워커가 보낸 메시지를 포함한다.

Target은 메시지를 보낸 워커의 레퍼런스이다.
이것은 워커가 어디에서 오는 것인지를 알아야 할 때 편리하다.

Now let’s write the worker

첫째, **워커가 manager.js로부터 보내진 메시지를 받을 수 있나를 확인**한다

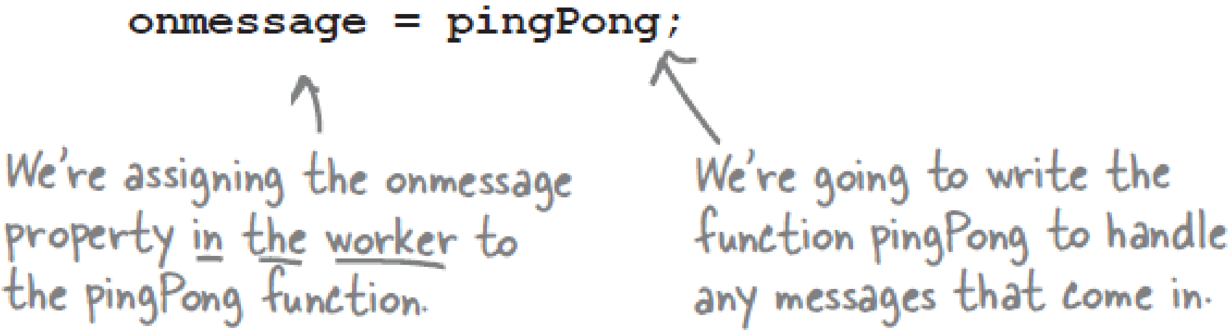
⇒ 워커가 작업 주문을 받는 방법

⇒ 이를 위해 **또 다른 onmessage 핸들러**(워커 자체 내에 있다)를 이용한다.

⇒ 모든 워커는 메시지를 받을 준비가 되어 있다

⇒ 단지 그들을 처리할 핸들러를 워커에게 제공하기만 하면 된다

worker.js 파일을 생성해서 다음 코드를 추가한다:



```
onmessage = pingPong;
```

We're assigning the onmessage property in the worker to the pingPong function.

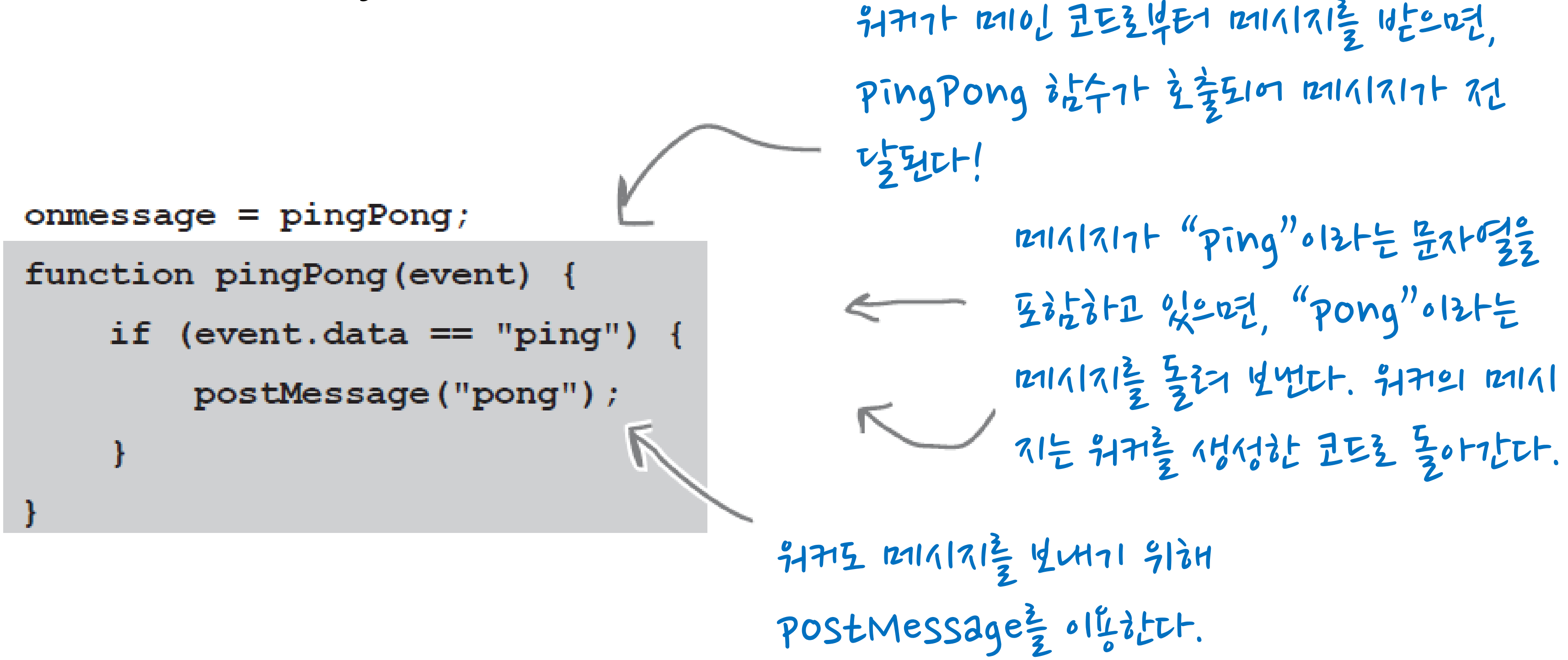
We're going to write the function pingPong to handle any messages that come in.

Now let’s write the worker

Writing the worker’s message handler

함수 pingPong은 메시지를 받아서 “pong”으로 응답한다.

다음 코드를 **worker.js**에 추가하자:



```
onmessage = pingPong;
function pingPong(event) {
    if (event.data == "ping") {
        postMessage("pong");
    }
}
```

워커가 메인 코드로부터 메시지를 받으면, pingPong 함수가 호출되어 메시지가 전달된다!

메시지가 “ping”이라는 문자열을 포함하고 있으면, “pong”이라는 메시지를 돌려 보낸다. 워커의 메시지는 워커를 생성한 코드로 돌아간다.

워커도 메시지를 보내기 위해 postMessage를 이용한다.

실습과제 16-1 Serving up a test drive

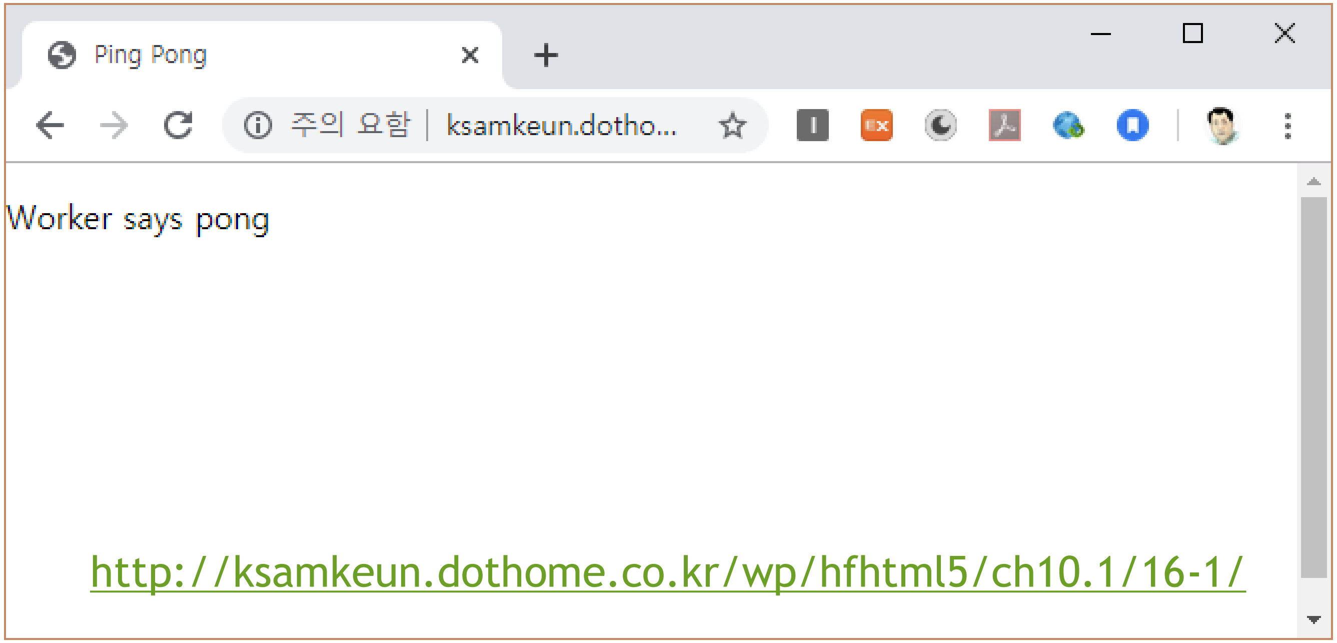
pingpong.html, manager.js, worker.js의 내용을 입력하고 저장하자.

manager.js => 새 워커를 만들어서 메시지 핸들러에게 할당한 다음 워커에게 “ping”을 보낸다.

워커 => pingPong이 메시지 핸들러로 설정되었는지 확인한 다음 기다린다.

어떤 시점에서 워커는 manager로부터 메시지를 받는다.

워커는 manager로부터 핸들러에게 메시지를 보내면 “ping"이 포함되어 있는지 확인한 다음 “pong" 메시지를 되돌려 보낸다.



```
Worker says pong
```

http://ksamkeun.dothome.co.kr/wp/hfhtml5/ch10.1/16-1/

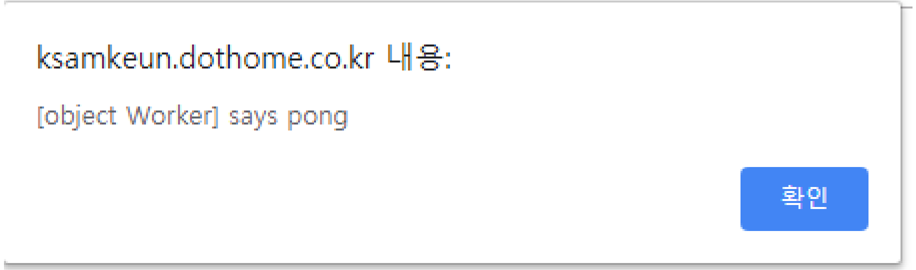
실습과제 16-2

Let's add a couple of workers to our **pingPong** game. Your job is to fill in the blanks to complete the code so we have three pings sent to the workers, and three pongs back from the workers.

```
window.onload = function() {
    var numWorkers = 3;
    var workers = [];
    for (var i = 0; i < ..........................; i++) {
        var worker = new ..........................("worker.js");
        worker. ......................  = function(event) {

            alert(event.target + " says "
                                 + event. ................);
        };
        workers.push(worker);
    }
    for (var i = 0; i < ..........................; i++) {
        workers[i]. .........................("ping");
    }
}
```

We're creating three workers, and storing them in an array, workers.

Here, we're adding the new worker to the workers array.


ksamkeun.dothome.co.kr 내용:

[object Worker] says pong

I've been wondering how to include additional JavaScript files in my worker. I've got some financial libraries I'd like to make use of and copying and pasting them into my worker would result in a huge file that's not very maintainable.

Take a look at **importScripts**.

웹 워커는 **importScripts**라는 **전역 함수**를 가지고 있다.

importScripts:
워커에 하나 이상의 자바스크립트 파일을 임포트하는데 사용:

```
importScripts("http://bigscience.org/nuclear.js",
              "http://nasa.gov/rocket.js",
              "mylibs/atomsmasher.js");
```

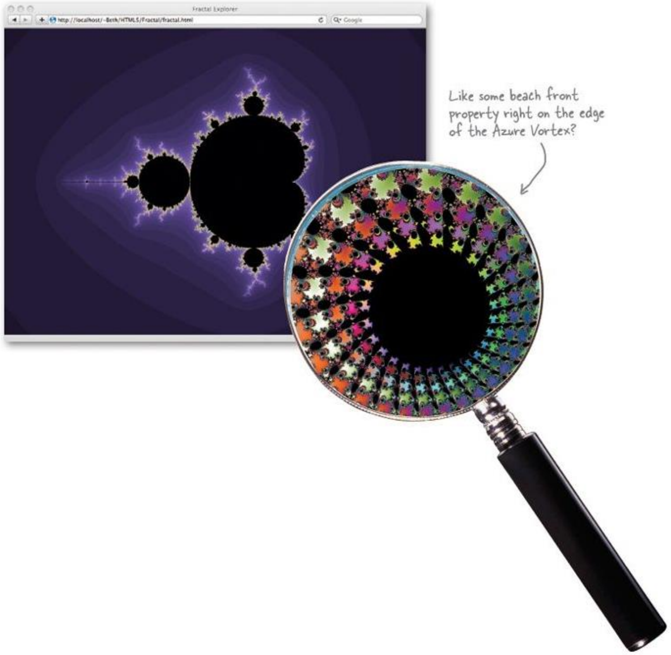
Place zero or more comma-separated JavaScript URLs in importScripts.

```
if (taskType == "songdetection") {
   importScripts("audio.js");
}
```

Because importScripts is a function, you can import code as the task demands.

Virtual Land Grab

Explorers of the **Mandelbrot Set** have already grabbed areas of the virtual countryside and given them names like the lovely "Seahorse Valley," "Rainbow Islands," and the dreaded "Black Hole." And given the value of physical real estate these days, the only play left seems to be in the virtual spaces. So, **we're going to build an explorer for the Mandelbrot Set to get in on the action.** Actually, we have to confess, we already have built it, but **it's slow** — navigating around in the entire Mandelbrot Set could take a very long time — so **we're hoping together we can speed it up**, and we have a hunch **Web Workers may be the answer.**



Like some beach front property right on the edge of the Azure Vortex?

Take a look around

Well, if you happen to be a mathematician then you know the Mandelbrot Set is the equation:

$$z_n+1 = z_n^2 + c$$

and that it was discovered and studied by **Benoit Mandelbrot**. You also know that it’s simply a **set of complex numbers** (numbers with a real part, and an imaginary part) generated by this equation.

How to compute a Mandelbrot Set

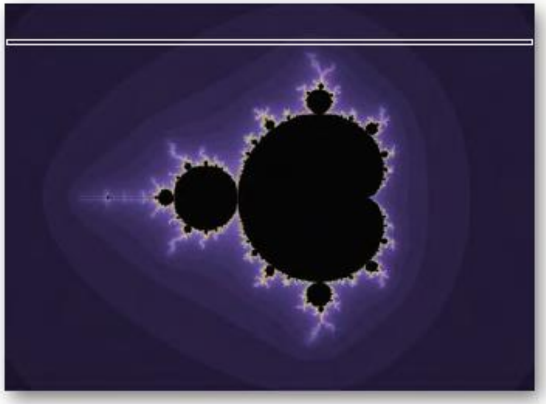
Mandelbrot 집합을 계산하기 위한 코드를 살펴보자:

To compute the Mandelbrot Set we loop over each row of the image.

```
for (i = 0; i < numberOfRows; i++) {
    var row = computeRow(i);
    drawRow(row);
}
```

And for each row we compute the pixels for that row.

And then we draw each row on the screen. You can probably see the row-by-row display when you run the test code in your browser.

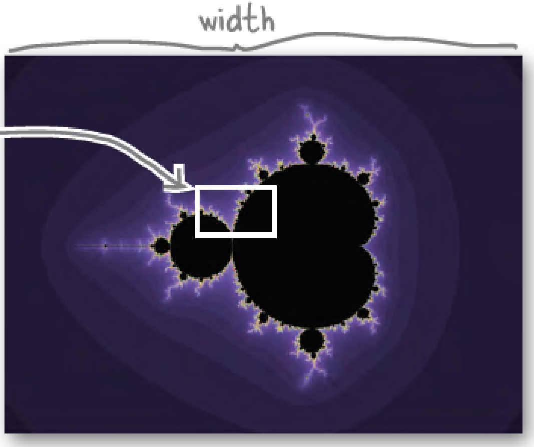


Now this code is just meant to be simple pseudo-code.
We can capture all those details in a **task object** like this:

```
for (i = 0; i < numberOfRows; i++) {
    var taskForRow = createTaskForRow(i);
    var row = computeRow(taskForRow);
    drawRow(row);
}
```

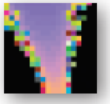
And we pass taskForRow into computeRow, which returns the computed row.

The taskForRow object holds all the data needed to compute a row.



width

The zoom factor

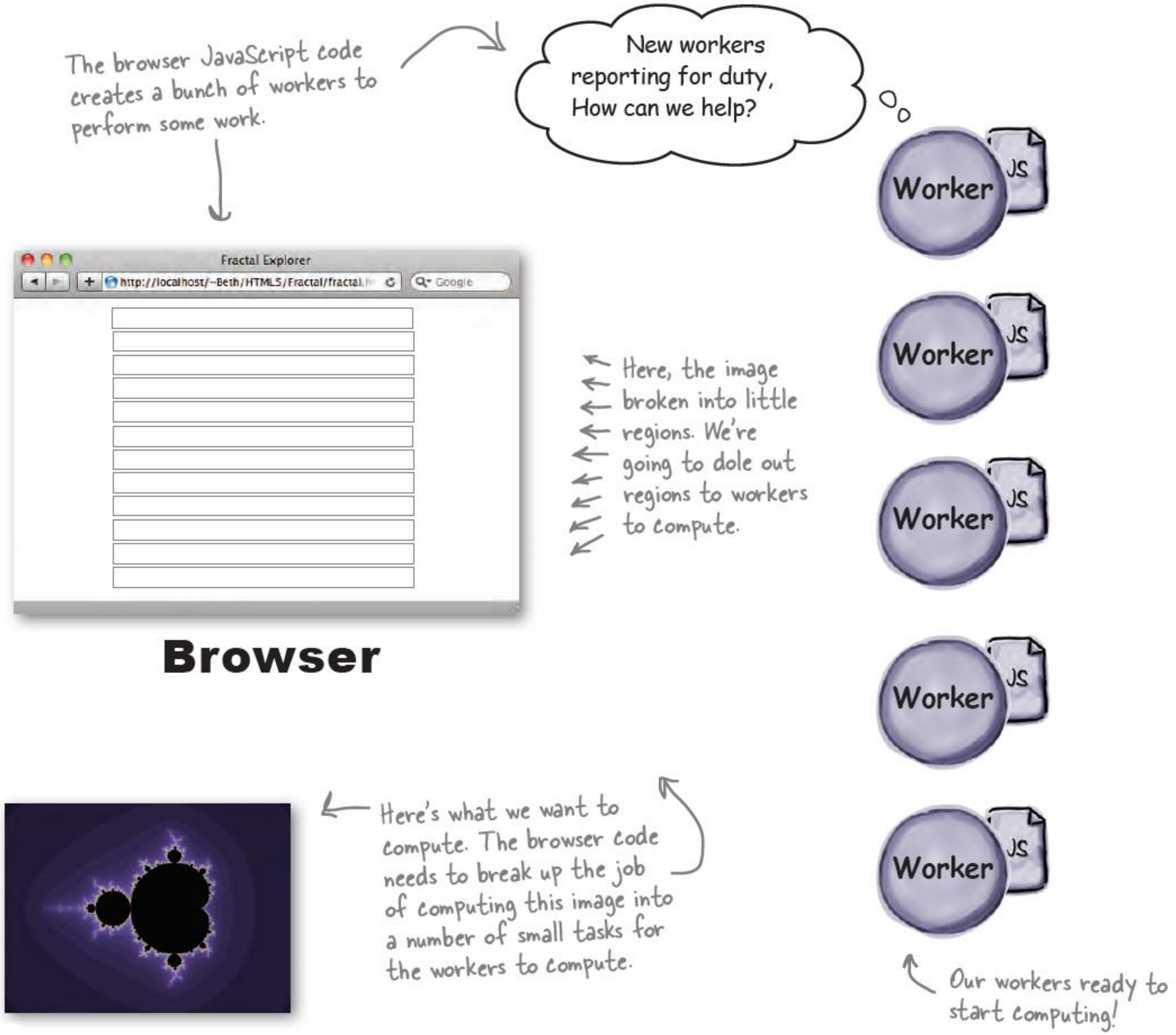


Level of precision to compute

여기서 트릭은 수많은 워커들이 작업할 수 있도록 전체 작업을 나누고, 나누어진 작업을 워커들에게 분배하고, 워커들이 작업을 완료하면 결과를 취합하는 코드를 추가하면 된다!

How to use multiple workers

브라우저는 먼저 한 묶음의 워커들을 생성한다. 너무 많은 워커들을 생성하면 오히려 비용이 더 들 수 있다. 5개의 워커들을 생성한다:


New workers reporting for duty, How can we help?
Fractal Explorer
http://localhost/~Beth/HTML5/Fractal/fractal.h
Google
Browser
Worker
JS
Worker
JS
Worker
JS
Worker
JS
Worker
JS

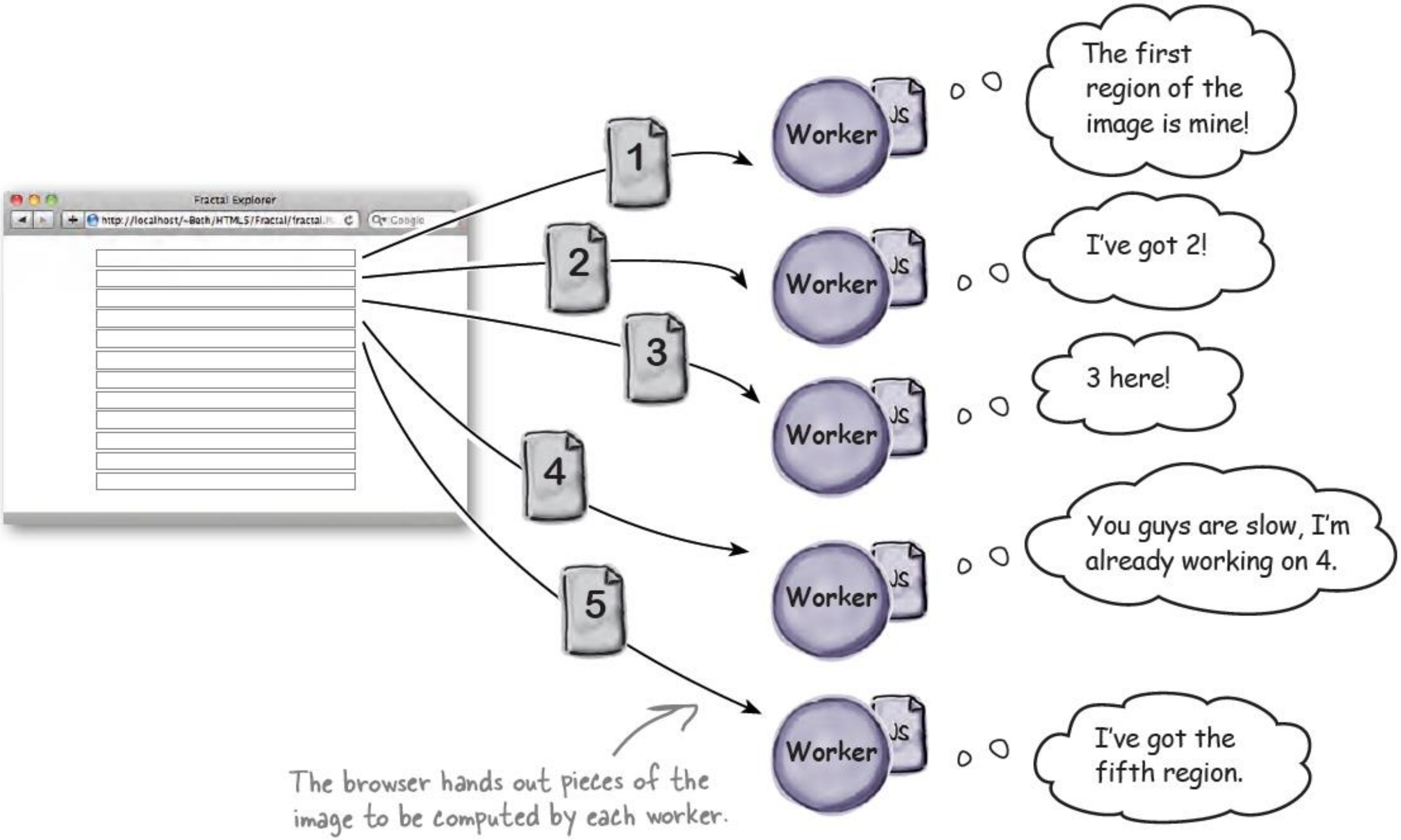
The browser JavaScript code creates a bunch of workers to perform some work.

Here, the image broken into little regions. We're going to dole out regions to workers to compute.

Here's what we want to compute. The browser code needs to break up the job of computing this image into a number of small tasks for the workers to compute.

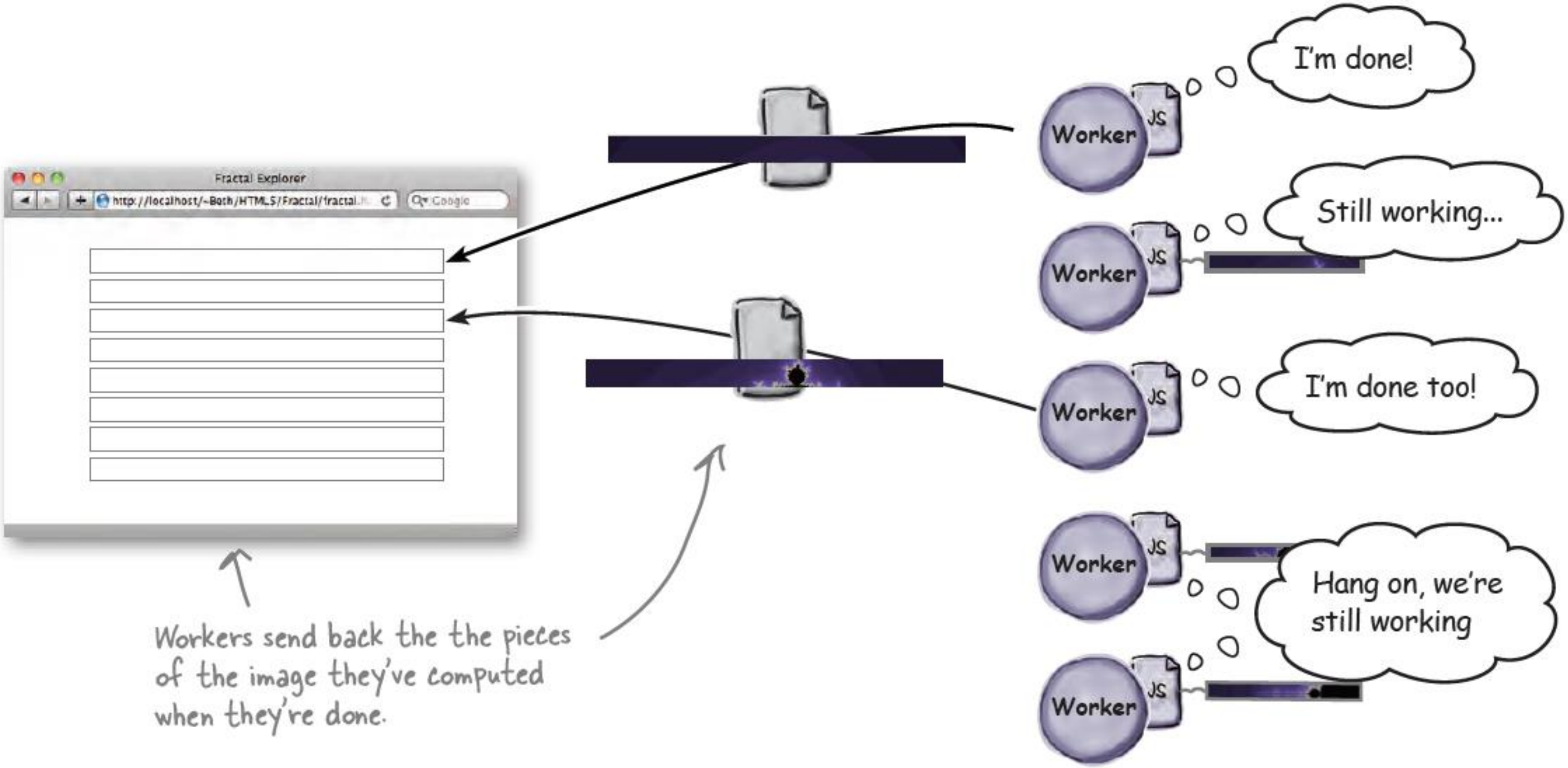
Our workers ready to start computing!

다음은, 브라우저 코드에서 각 워커가 계산할 서로 다른 파트의 이미지를 나누어 준다:


Fractal Explorer
http://localhost/~Beth/HTML5/Fractal/fractal
1
2
3
4
5
Worker
JS
The first region of the image is mine!
I've got 2!
3 here!
You guys are slow, I'm already working on 4.
I've got the fifth region.

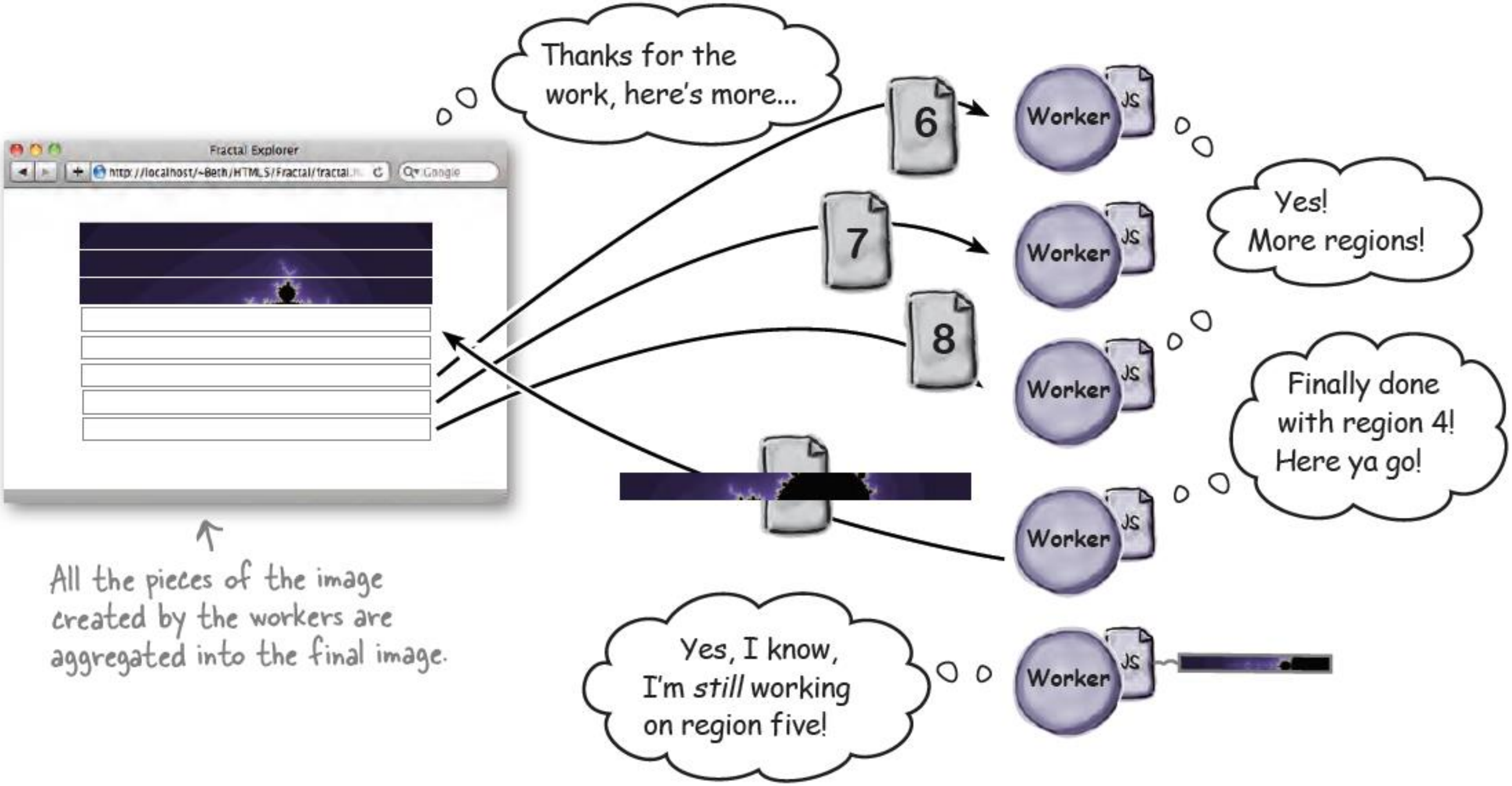
The browser hands out pieces of the image to be computed by each worker.

각 워커는 자신이 맡은 이미지 부분을 **개별적으로** 계산한다.
워커가 자신의 작업을 완료하면, 결과를 묶어서 되돌려 준다.


Fractal Explorer
http://localhost/~Beth/HTML5/Fractal/fractal.h
Google
Worker
JS
I'm done!
Still working...
I'm done too!
Hang on, we're still working

Workers send back the the pieces of the image they've computed when they're done.

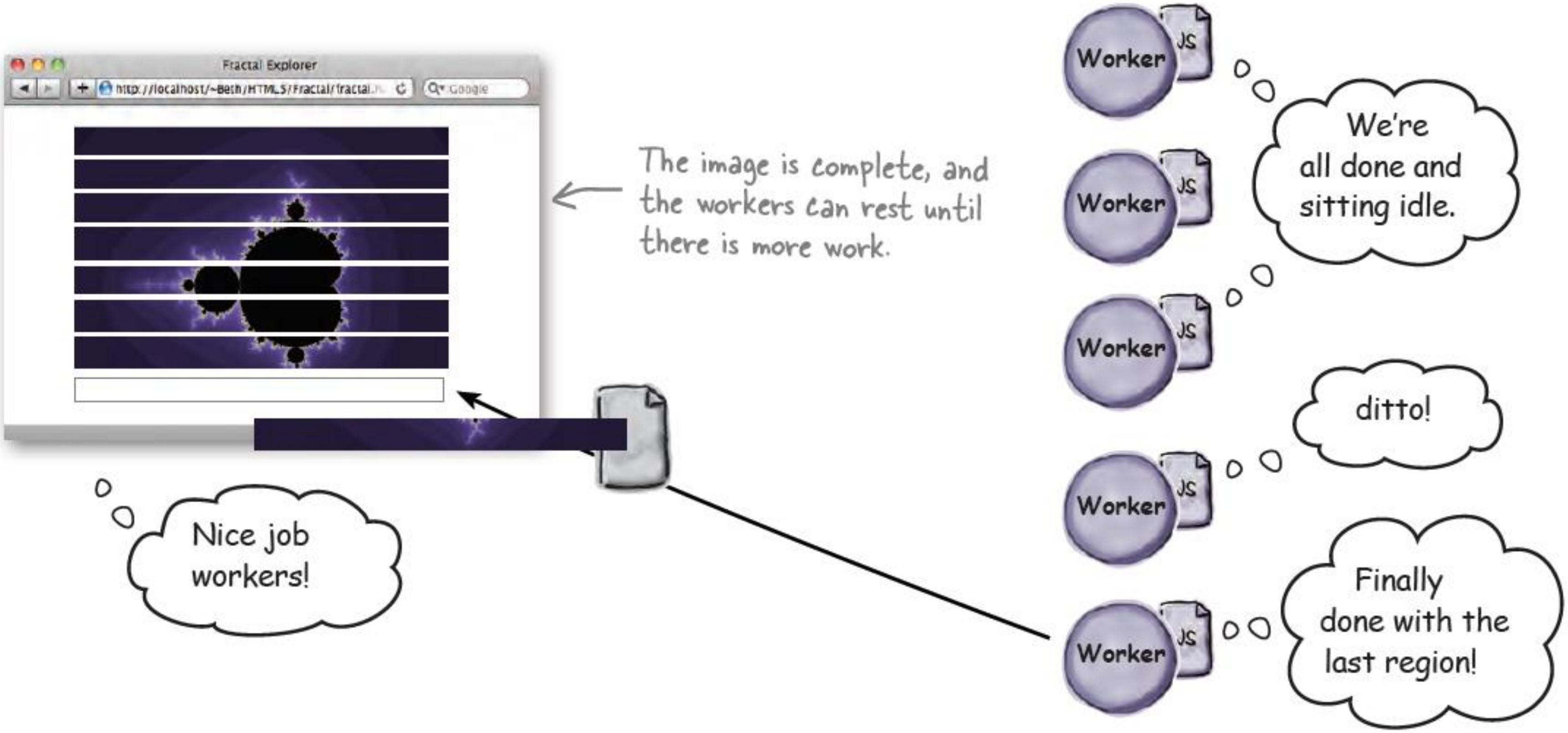
이미지 조각들이 워커들로부터 되돌아 오면, 브라우저의 이미지로 모아진다.
계산할 이미지 조각이 남아 있다면, 놀고 있는(idle) 워커들에게 새로운 작업이 맡겨진다.



Fractal Explorer
http://localhost/~Beth/HTML5/Fractal/fractal.h
Google
Thanks for the work, here's more...
6
7
8
Worker
JS
Yes! More regions!
Finally done with region 4! Here ya go!
Yes, I know, I'm *still* working on region five!

All the pieces of the image created by the workers are aggregated into the final image.

마지막 이미지 조각의 계산이 완료되면, 이미지는 완성되고 워커들은 모두 놀고 있는 상태로 남아있는다. 그러다 사용자가 줌 인을 클릭하면 이 모든 것이 다시 시작된다...


Fractal Explorer
http://localhost/~Beth/HTML5/Fractal/fractal.h
Google
The image is complete, and the workers can rest until there is more work.
Nice job workers!
Worker
JS
We're all done and sitting idle.
ditto!
Finally done with the last region!

아무리 Task를 쪼개서 워커들에게 분배한다 해도 여전히 동일한 CPU를 사용하고 있는데 어떻게 더 빨라질 수 있나요?

It can be faster in two ways...

First consider an **application that has a lot of "computing"** going on that also has to be responsive to the user... By adding workers to such an app you can immediately improve the feel of the app for your users. Why? Because JavaScript has a chance to respond to user interaction in between getting results from the workers, something it doesn't have a chance to do if everything's being computed on the main thread. So the UI is more responsive — and your app's just going to **feel faster**.

The second way really is faster. Almost all modern desktops and devices today are shipping with **multicore processors** (and perhaps even multiple processors). Multicore just means that the processor can do multiple things concurrently. With just a single thread of control, JavaScript in the browser doesn't make use of your extra cores or your extra processors, they're just wasted. However, if you use Web Workers, the workers can take advantage of running on your different cores and you'll see a **real speedup** in your app because you've got more processor power being thrown at it. If you've got a multicore machine, just wait, you're going to see the difference soon.

In theory, not in practice.

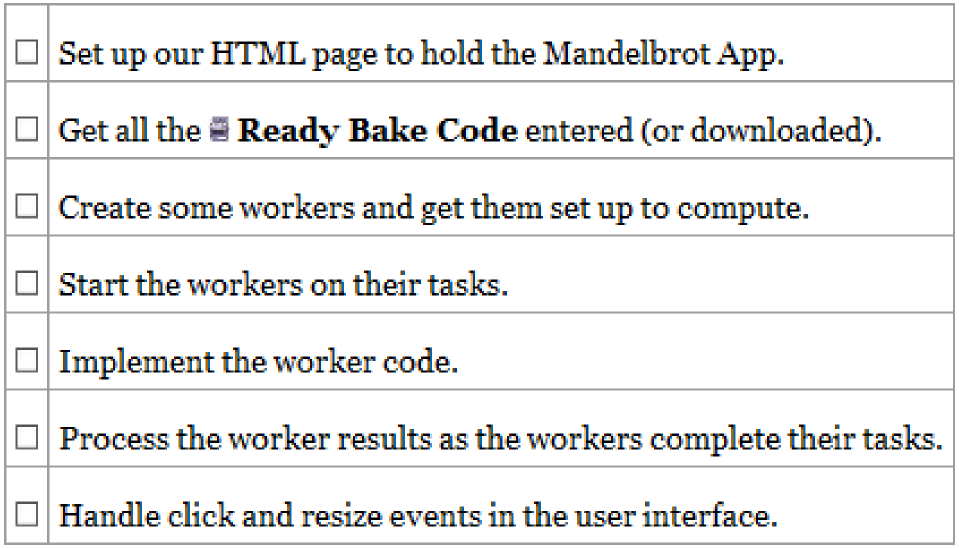
웹 워커는 대규모로 사용되지 않는 것이 좋다.

워커를 생성하는 일이 코드에서는 단순해 보일진 몰라도 실제 내부적으로는 여분의 메모리와 OS 스레드를 필요로 한다:
Start-up 시간 및 리소스 사용 측면에서 비용이 된다.

따라서 일반적으로 적절한 개수의 워커를 생성하여 시간차를 두고 **재사용**하는 것이 좋다.

Let’s build the Fractal Explorer app

Here’s what we need to do:



□ Set up our HTML page to hold the Mandelbrot App.

□ Get all the **Ready Bake Code** entered (or downloaded).

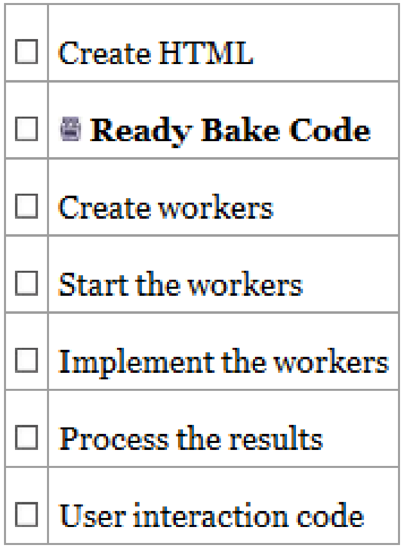
□ Create some workers and get them set up to compute.

□ Start the workers on their tasks.

□ Implement the worker code.

□ Process the worker results as the workers complete their tasks.

□ Handle click and resize events in the user interface.



□ Create HTML

□ **Ready Bake Code**

□ Create workers

□ Start the workers

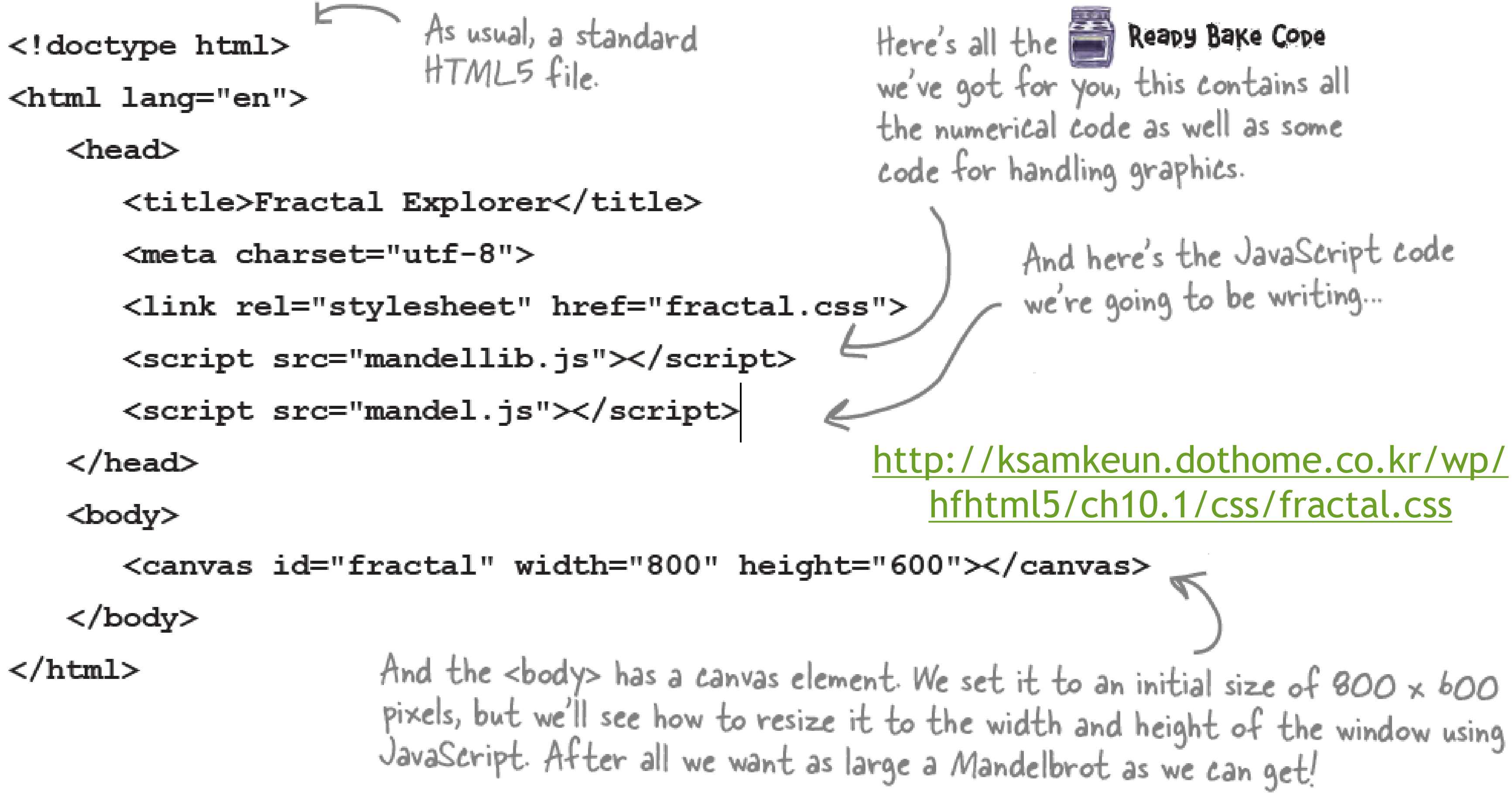
□ Implement the workers

□ Process the results

□ User interaction code

Creating the Fractal Viewer HTML Markup

먼저 앱을 저장할 HTML 페이지(**fractal.html**)를 설정한다:



```
<!doctype html>
<html lang="en">
   <head>
      <title>Fractal Explorer</title>
      <meta charset="utf-8">
      <link rel="stylesheet" href="fractal.css">
      <script src="mandellib.js"></script>
      <script src="mandel.js"></script>
   </head>
   <body>
      <canvas id="fractal" width="800" height="600"></canvas>
   </body>
</html>
```

As usual, a standard HTML5 file.

Here's all the Ready Bake Code we've got for you, this contains all the numerical code as well as some code for handling graphics.

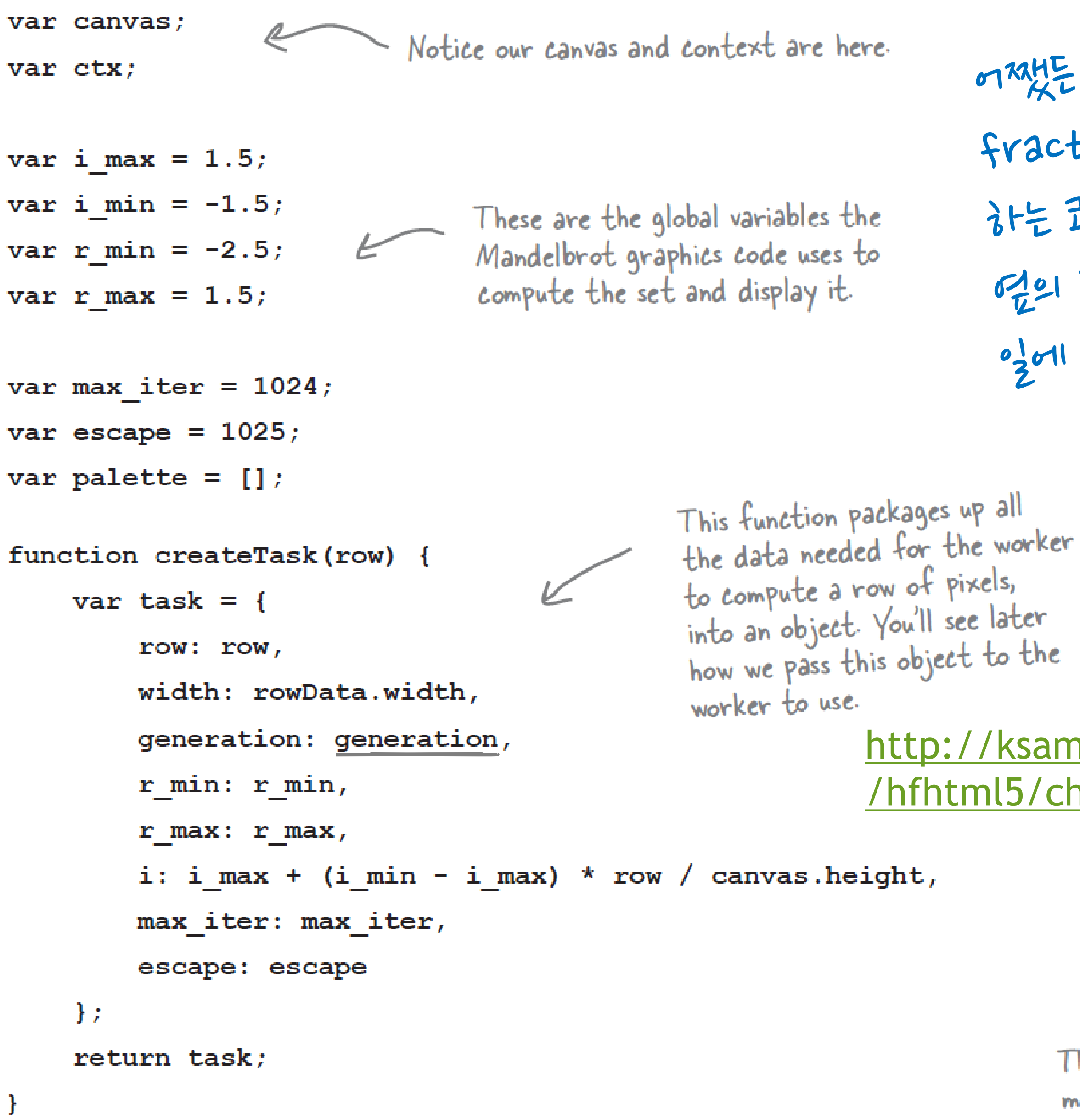
And here's the JavaScript code we're going to be writing...

http://ksamkeun.dothome.co.kr/wp/hfhtml5/ch10.1/css/fractal.css

And the <body> has a canvas element. We set it to an initial size of 800 x 600 pixels, but we'll see how to resize it to the width and height of the window using JavaScript. After all we want as large a Mandelbrot as we can get!

Ready Bake Code

어쨌든, 먼저 작업을 관리하고 fractal 이미지에 행들을 drawing 하는 코드를 얻었다.
옆의 코드를 “mandellib.js”라는 파일에 저장하자:



```
var canvas;
var ctx;

var i_max = 1.5;
var i_min = -1.5;
var r_min = -2.5;
var r_max = 1.5;

var max_iter = 1024;
var escape = 1025;
var palette = [];

function createTask(row) {
    var task = {
        row: row,
        width: rowData.width,
        generation: generation,
        r_min: r_min,
        r_max: r_max,
        i: i_max + (i_min - i_max) * row / canvas.height,
        max_iter: max_iter,
        escape: escape
    };
    return task;
}
```

Notice our canvas and context are here.

These are the global variables the Mandelbrot graphics code uses to compute the set and display it.

This function packages up all the data needed for the worker to compute a row of pixels, into an object. You'll see later how we pass this object to the worker to use.

This code goes in mandellib.js.

http://ksamkeun.dothome.co.kr/wp/hfhtml5/ch10.1/mandellib.zip

```
function makePalette() {
    function wrap(x) {
       x = ((x + 256) & 0x1ff) - 256;
       if (x < 0) x = -x;
        return x;
    }
    for (i = 0; i <= this.max_iter; i++) {
        palette.push([wrap(7*i), wrap(5*i), wrap(11*i)]);
    }
}
```

makePalette maps a large set of numbers into an array of rgb colors. We'll use this palette in drawRow (below) to convert the value we get back from a worker to a color for the graphic display of the set (the fractal image).

drawRow takes the results from the worker and draws them into the canvas.

```
function drawRow(workerResults) {
    var values = workerResults.values;
    var pixelData = rowData.data;
    for (var i = 0; i < rowData.width; i++) {
        var red = i * 4;
        var green = i * 4 + 1;
        var blue = i * 4 + 2;
        var alpha = i * 4 + 3;
        pixelData[alpha] = 255; // set alpha to opaque
        if (values[i] < 0) {
            pixelData[red] = pixelData[green] = pixelData[blue] = 0;
        } else {
            var color = this.palette[values[i]];
            pixelData[red] = color[0];
            pixelData[green] = color[1];
            pixelData[blue] = color[2];
        }
    }
    ctx.putImageData(this.rowData, 0, workerResults.row);
}
```

It uses this rowData variable to do it; rowData is a one-row ImageData object that holds the actual pixels for that row of the canvas.

Here's where we use the palette to map the result from the worker (just a number) to a color.

This code should be familiar; it's similar to what we did in Chapter 8 with video and canvas.

And here's where we write the pixels to the ImageData object in the context of the canvas!

This code goes in mandellib.js.

setUpGraphics sets up the global variables used by all the graphics drawing code as well as the Mandelbrot computation.

```
function setupGraphics() {

    canvas = document.getElementById("fractal");
    ctx = canvas.getContext("2d");

    canvas.width = window.innerWidth;
    canvas.height = window.innerHeight;

    var width = ((i_max - i_min) * canvas.width / canvas.height);
    var r_mid = (r_max + r_min) / 2;
    r_min = r_mid - width/2;
    r_max = r_mid + width/2;

    rowData = ctx.createImageData(canvas.width, 1);

    makePalette();
}
```

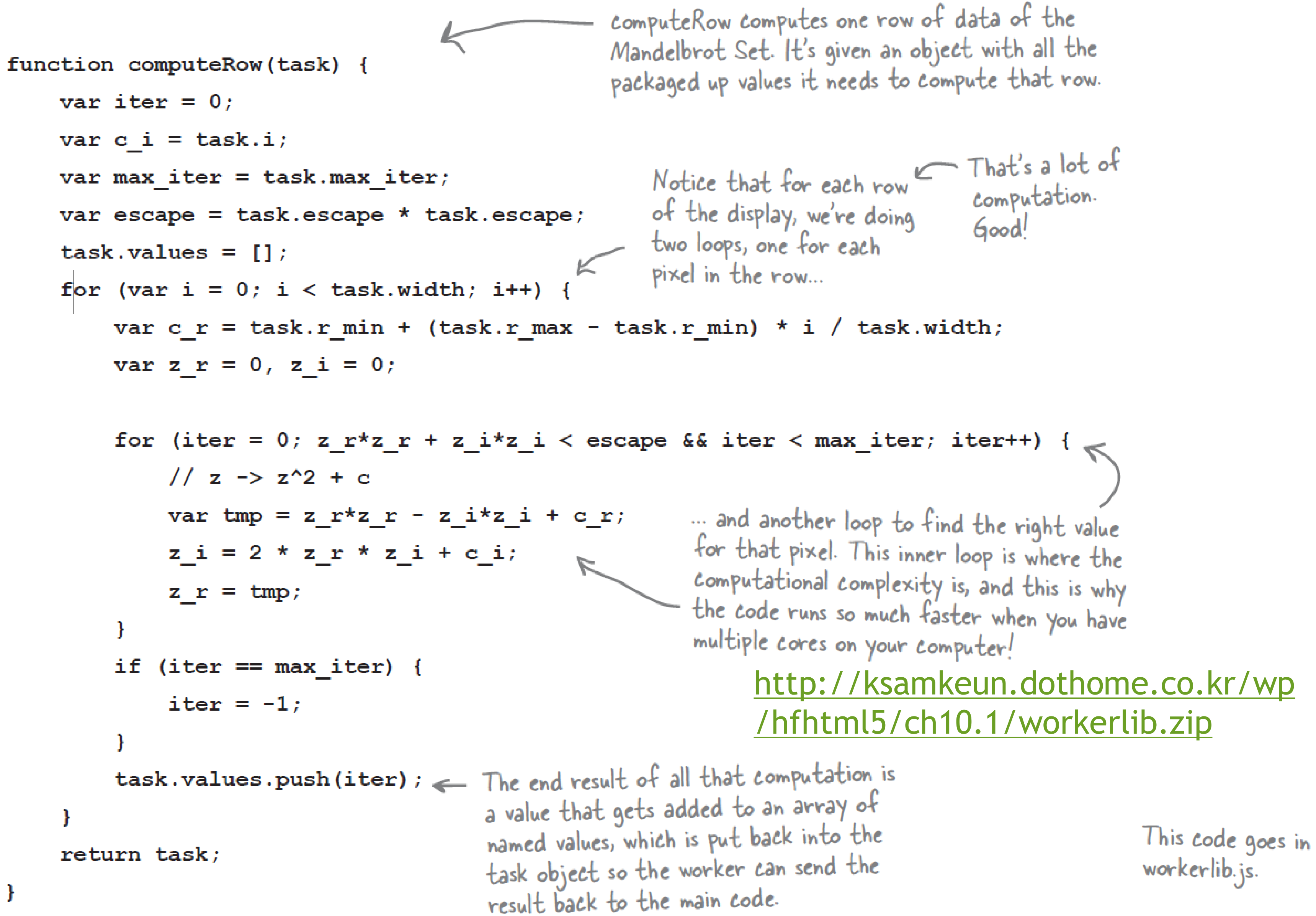
Here's where we grab the canvas and the context and set the initial width and height of the canvas.

These are variables used to compute the Mandelbrot Set.

Here, we're initializing the rowData variable (used to write the pixels to the canvas).

And here we're initializing the palette of colors we're using to draw the set as a fractal image.

Ready Bake Code: 워커가 망델브로 집합의 수학 계산을 하려고 사용한다. 아래 코드를 “**workerlib.js**”에 저장하자:



```
function computeRow(task) {
    var iter = 0;
    var c_i = task.i;
    var max_iter = task.max_iter;
    var escape = task.escape * task.escape;
    task.values = [];
    for (var i = 0; i < task.width; i++) {
        var c_r = task.r_min + (task.r_max - task.r_min) * i / task.width;
        var z_r = 0, z_i = 0;

        for (iter = 0; z_r*z_r + z_i*z_i < escape && iter < max_iter; iter++) {
            // z -> z^2 + c
            var tmp = z_r*z_r - z_i*z_i + c_r;
            z_i = 2 * z_r * z_i + c_i;
            z_r = tmp;
        }
        if (iter == max_iter) {
            iter = -1;
        }
        task.values.push(iter);
    }
    return task;
}
```

computeRow computes one row of data of the Mandelbrot Set. It's given an object with all the packaged up values it needs to compute that row.

Notice that for each row of the display, we're doing two loops, one for each pixel in the row...

That's a lot of computation. Good!

... and another loop to find the right value for that pixel. This inner loop is where the computational complexity is, and this is why the code runs so much faster when you have multiple cores on your computer!

The end result of all that computation is a value that gets added to an array of named values, which is put back into the task object so the worker can send the result back to the main code.

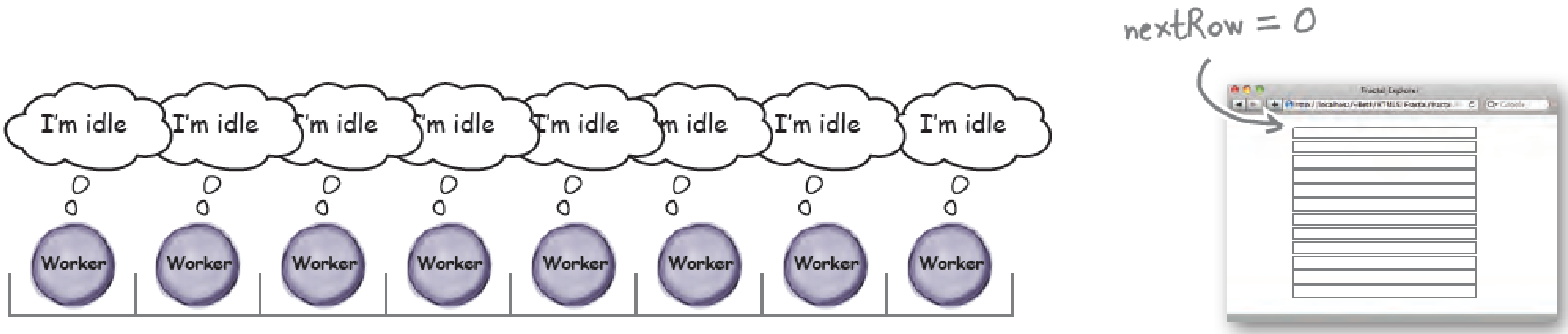
This code goes in workerlib.js.

http://ksamkeun.dothome.co.kr/wp/hfhtml5/ch10.1/workerlib.zip

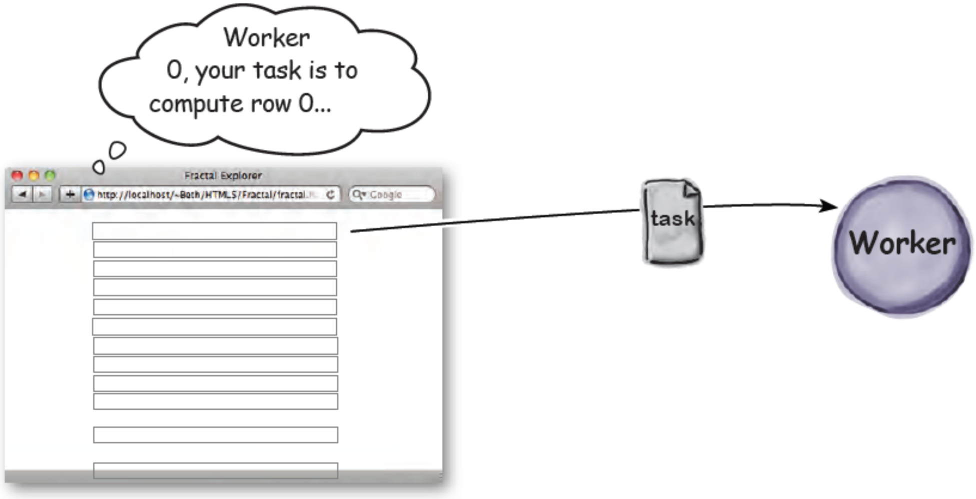
Creating workers, and giving them tasks…

워커를 생성해서 작업을 시켜보자:

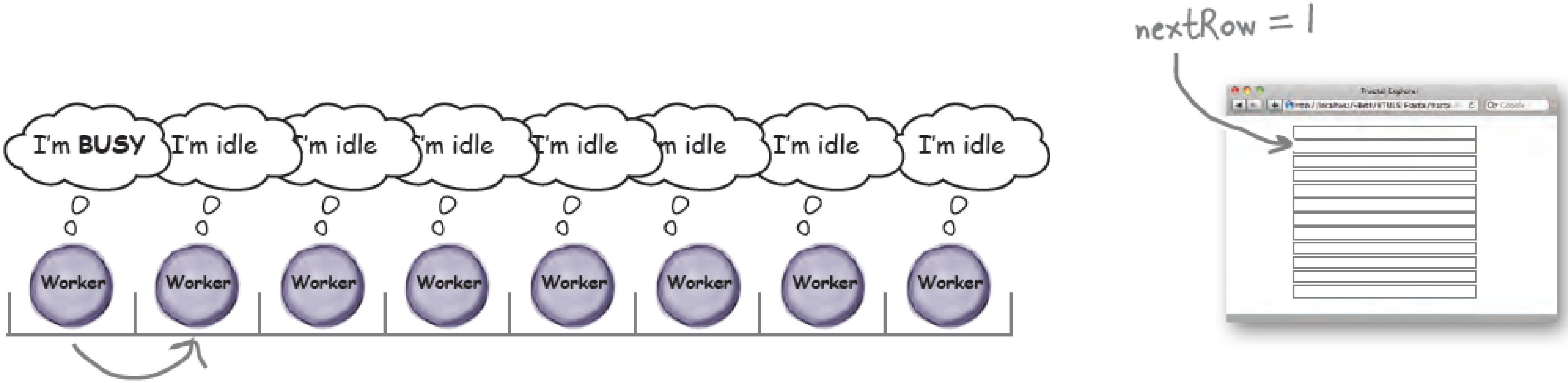
1. 워커 배열을 생성한다: 처음에는 모두 **idle 상태**이다 (nextRow = 0).



2. 배열을 통해 각 **idle 워커**에 대한 작업(task)을 생성한다.

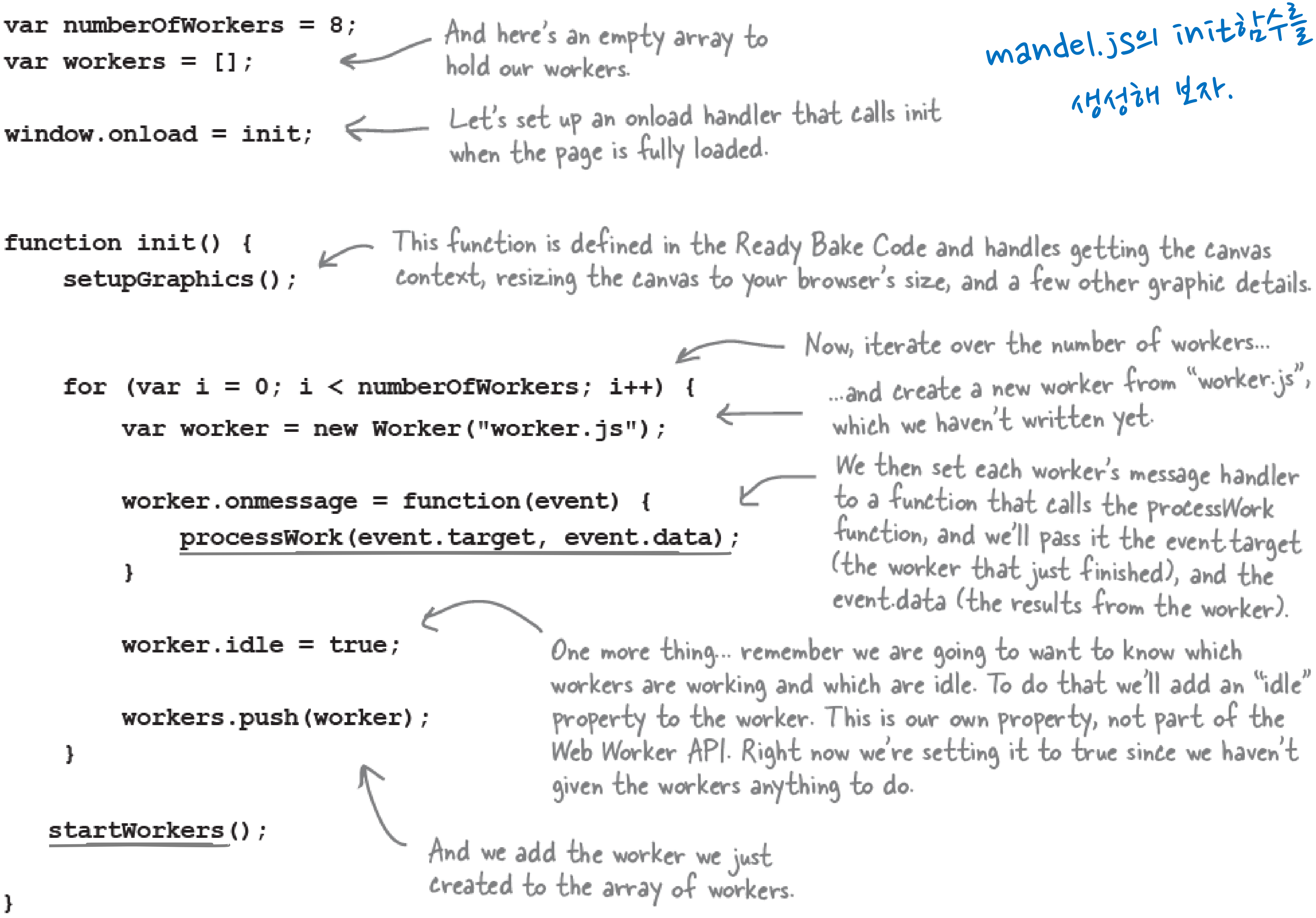


3. 다음 idle 워커에 할당할 작업을 찾아서 계속 반복한다: nextRow = 1. . .



Writing the code

mandel.js의 init함수를 생성해 보자.



```
var numberOfWorkers = 8;
var workers = [];

window.onload = init;


function init() {
    setupGraphics();


    for (var i = 0; i < numberOfWorkers; i++) {
        var worker = new Worker("worker.js");

        worker.onmessage = function(event) {
            processWork(event.target, event.data);
        }

        worker.idle = true;

        workers.push(worker);
    }

   startWorkers();

}
```

And here's an empty array to hold our workers.

Let's set up an onload handler that calls init when the page is fully loaded.

This function is defined in the Ready Bake Code and handles getting the canvas context, resizing the canvas to your browser's size, and a few other graphic details.

Now, iterate over the number of workers...

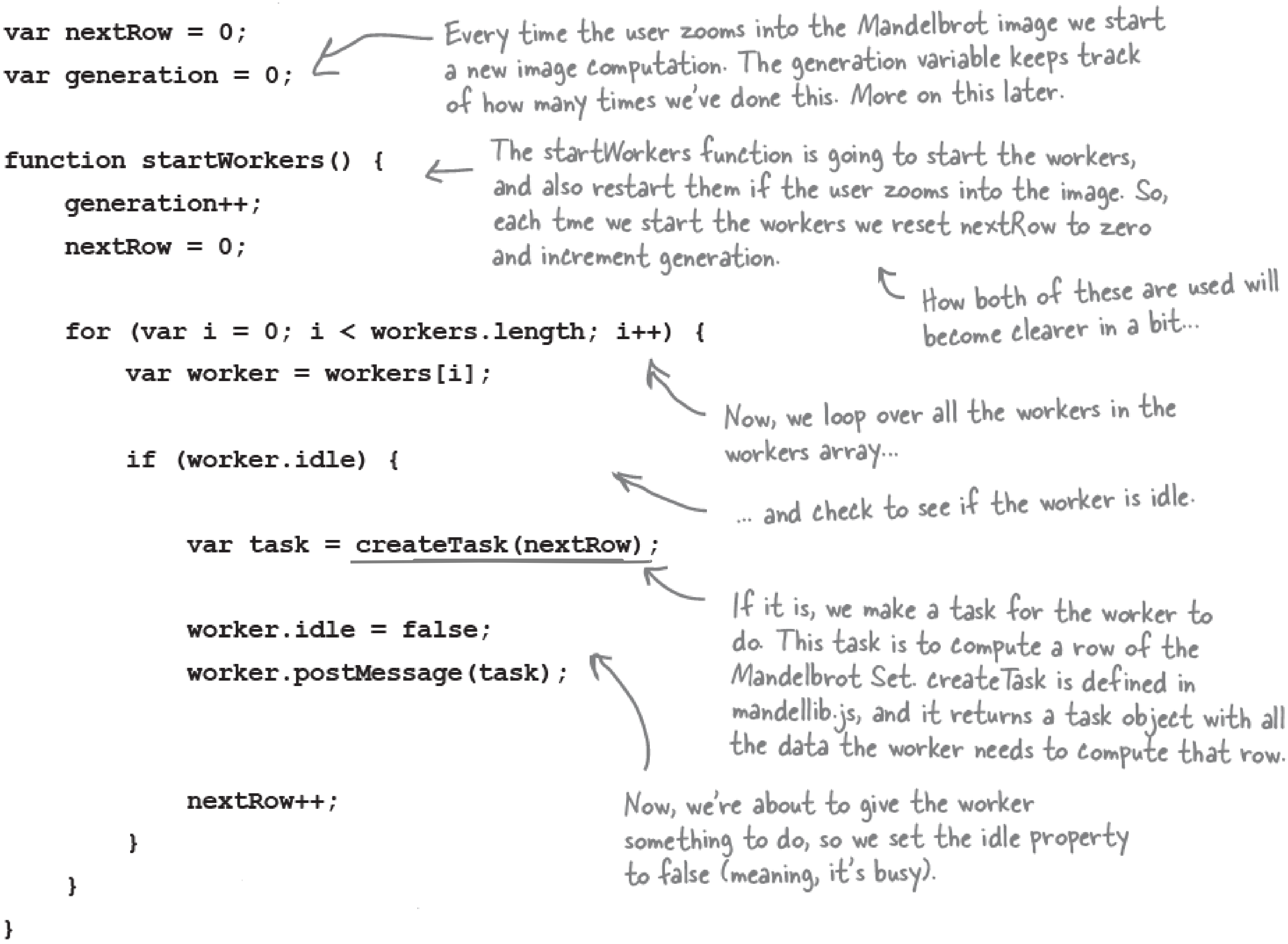
...and create a new worker from "worker.js", which we haven't written yet.

We then set each worker's message handler to a function that calls the processWork function, and we'll pass it the event.target (the worker that just finished), and the event.data (the results from the worker).

One more thing... remember we are going to want to know which workers are working and which are idle. To do that we'll add an "idle" property to the worker. This is our own property, not part of the Web Worker API. Right now we're setting it to true since we haven't given the workers anything to do.

And we add the worker we just created to the array of workers.

Getting the workers started



```
var nextRow = 0;
var generation = 0;

function startWorkers() {
    generation++;
    nextRow = 0;

    for (var i = 0; i < workers.length; i++) {
        var worker = workers[i];

        if (worker.idle) {

            var task = createTask(nextRow);

            worker.idle = false;
            worker.postMessage(task);


            nextRow++;
        }
    }
}
```

Every time the user zooms into the Mandelbrot image we start a new image computation. The generation variable keeps track of how many times we've done this. More on this later.

The startWorkers function is going to start the workers, and also restart them if the user zooms into the image. So, each tme we start the workers we reset nextRow to zero and increment generation.

How both of these are used will become clearer in a bit...

Now, we loop over all the workers in the workers array...

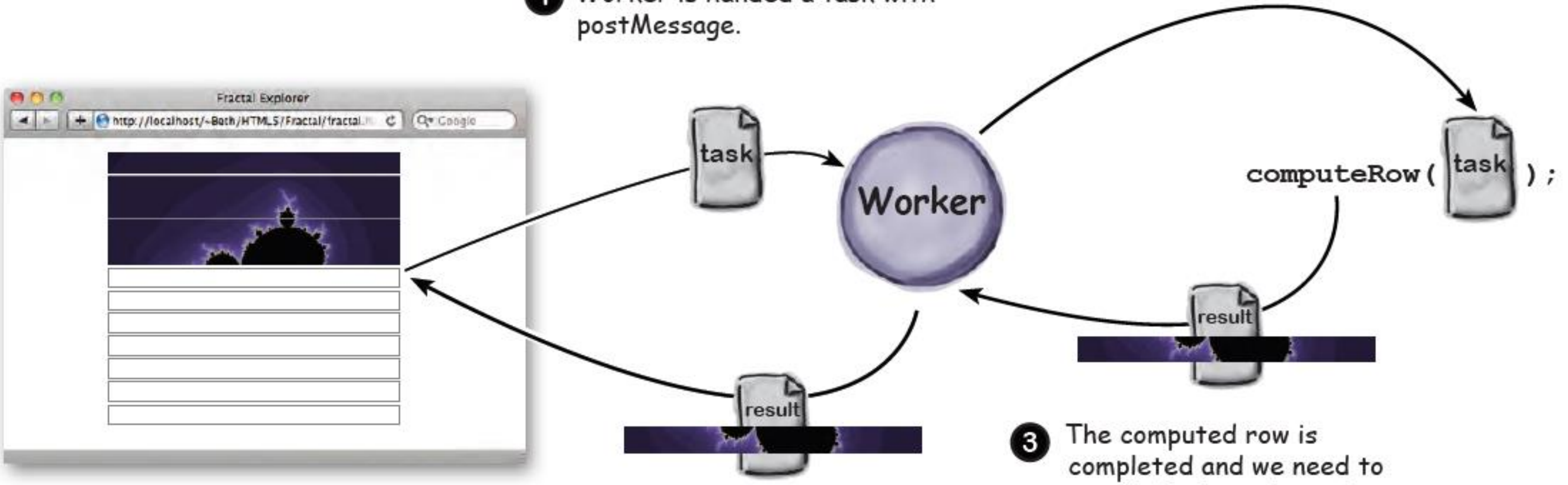
... and check to see if the worker is idle.

If it is, we make a task for the worker to do. This task is to compute a row of the Mandelbrot Set. createTask is defined in mandellib.js, and it returns a task object with all the data the worker needs to compute that row.

Now, we're about to give the worker something to do, so we set the idle property to false (meaning, it's busy).

Implementing the worker

Let’s quickly review **how it should work:**



1 Worker is handed a task with postMessage.

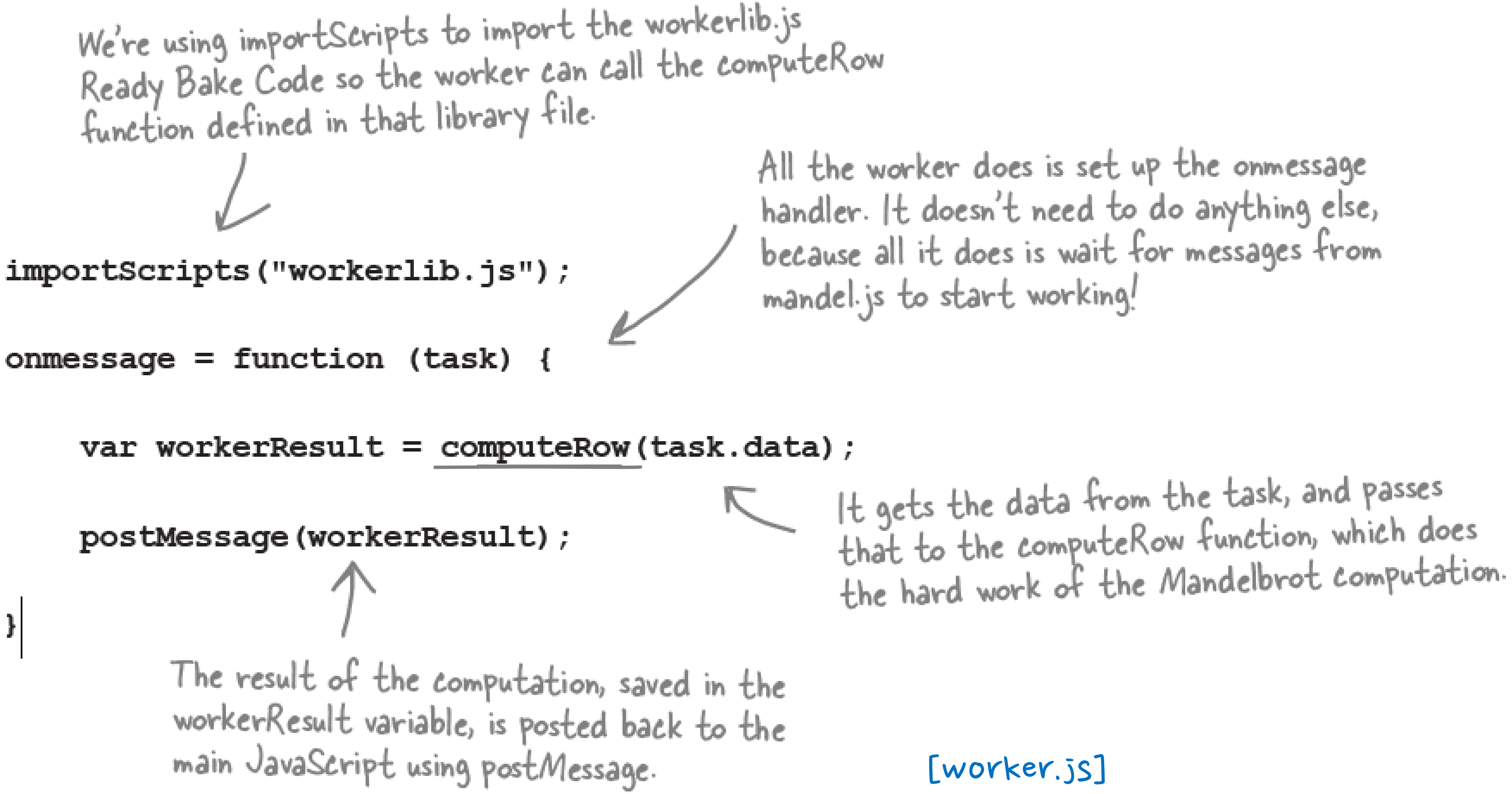
2 The worker takes the task and passes it to a Ready Bake function to compute the row.

3 The computed row is completed and we need to send it back to the main page code.

4 Result is sent back from worker with another postMessage.

Implementing the worker

So let's implement this: go head and type the following code into your **worker.js** file.



We're using importScripts to import the workerlib.js Ready Bake Code so the worker can call the computeRow function defined in that library file.

All the worker does is set up the onmessage handler. It doesn't need to do anything else, because all it does is wait for messages from mandel.js to start working!

```
importScripts("workerlib.js");

onmessage = function (task) {

    var workerResult = computeRow(task.data);

    postMessage(workerResult);

}
```

It gets the data from the task, and passes that to the computeRow function, which does the hard work of the Mandelbrot computation.

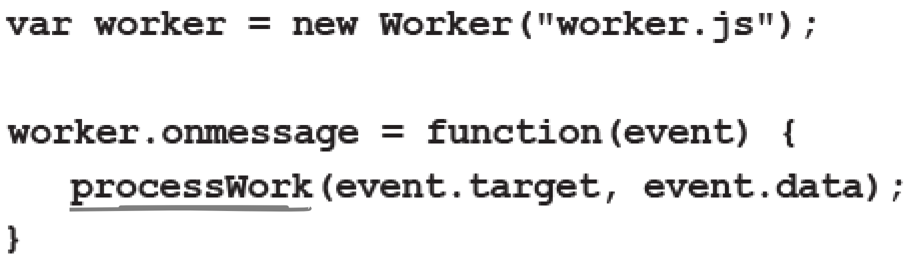
The result of the computation, saved in the workerResult variable, is posted back to the main JavaScript using postMessage.

[worker.js]

How to process the worker's results

워커로부터 작업 결과를 돌려 받을 때의 상황을 살펴보자:

워커를 생성할 때 **processWork**라는 메시지 핸들러를 할당했었다.

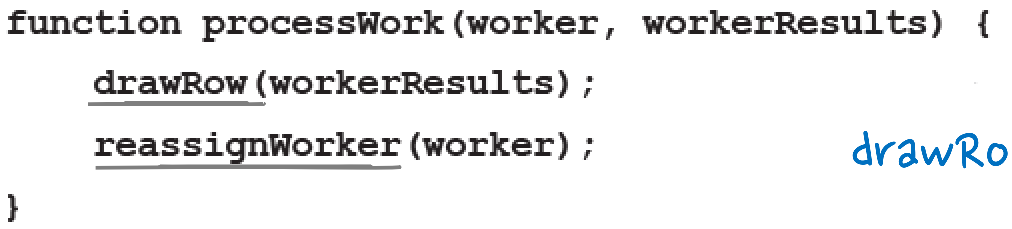


```
var worker = new Worker("worker.js");

worker.onmessage = function(event) {
   processWork(event.target, event.data);
}
```

워커가 결과와 함께 메시지를 돌려주면 그것을 제어하는 것은 **processWork** 함수이다.

다음처럼 두 가지 것을 돌려준다: 메시지의 **target**과 메시지의 **data**.

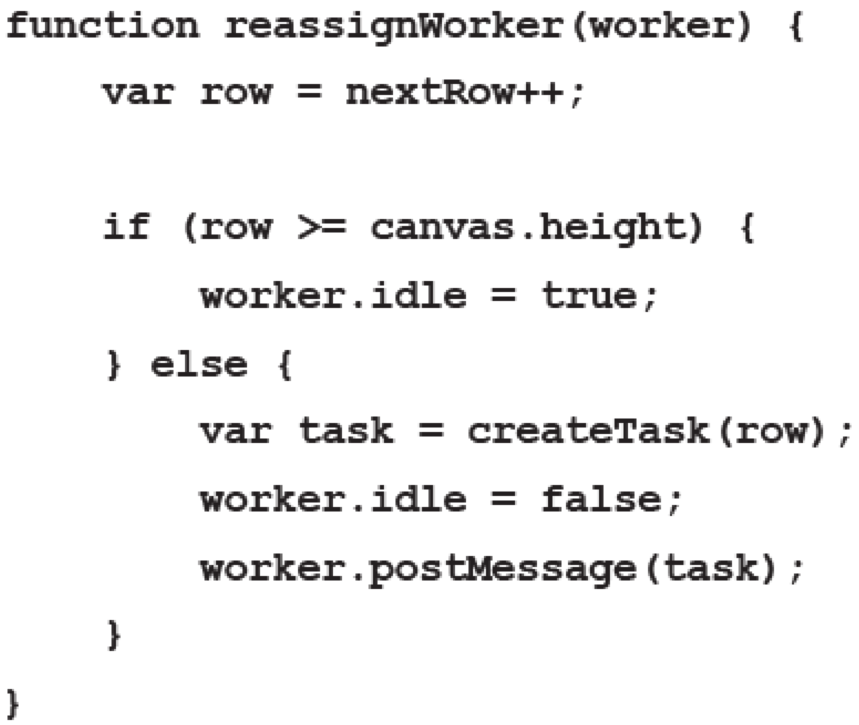
processWork를 작성하여 **mandel.js**에 추가하자:



```
function processWork(worker, workerResults) {
     drawRow(workerResults);
     reassignWorker(worker);
}
```

drawRow: mandellib.js의 함수

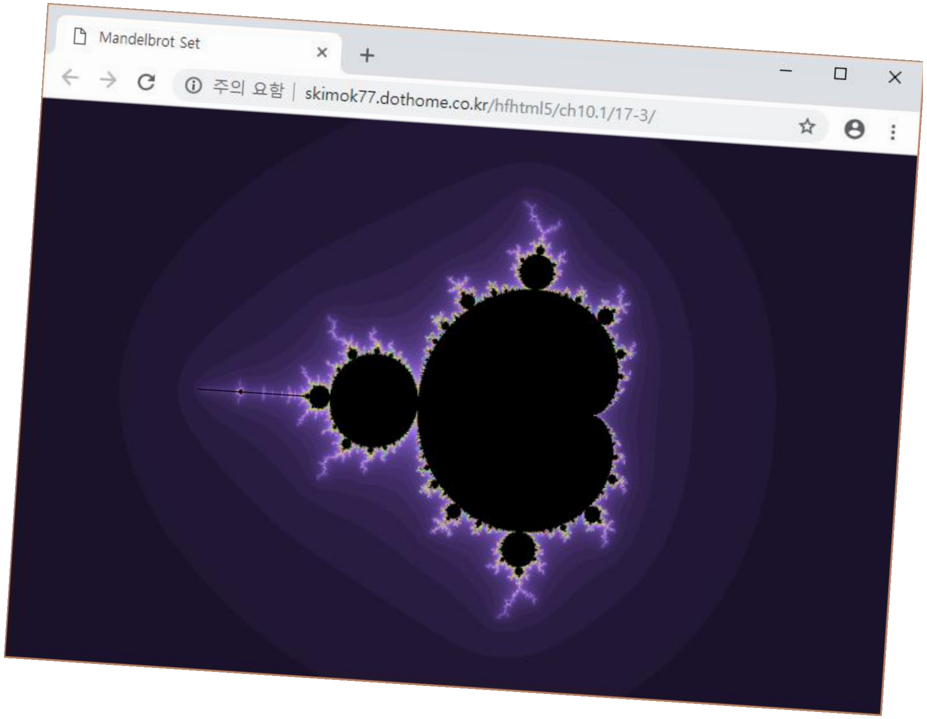
다음 코드를 **mandel.js**에 추가한다:



```
function reassignWorker(worker) {
    var row = nextRow++;

    if (row >= canvas.height) {
        worker.idle = true;
    } else {
        var task = createTask(row);
        worker.idle = false;
        worker.postMessage(task);
    }
}
```

실습과제 16-3 Psychedelic test drive

Enough code already! Let’s road test this thing. Load the fractal.html file into your browser and see your workers going to work. Depending on your machine, your Fractal Explorer should run a little faster than before.



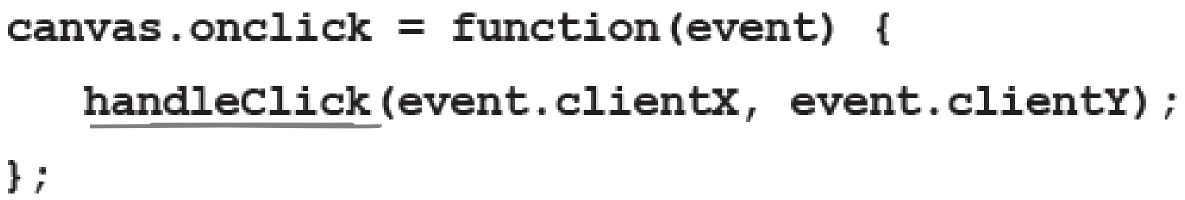
http://ksamkeun.dothome.co.kr/wp/hfhtml5/ch10.1/16-3/

Psychedelic test drive

Handling a click event

마우스 클릭을 제어하기 위한 **핸들러**를 추가하자.

- 클릭은 **canvas** 엘리먼트에서 발생한다:
- 따라서 캔버스의 **onclick** 프로퍼티에 대한 핸들러를 추가하면 된다:



```
canvas.onclick = function(event) {
   handleClick(event.clientX, event.clientY);
};
```

위의 코드를 “**mandel.js**”에서 init 함수의 **setUpGraphics** 호출 아래에 추가하자.

handleClick is called when the user clicks on the canvas to zoom into the fractal.

We pass in the x, y position of the click so we know where they clicked on the screen.

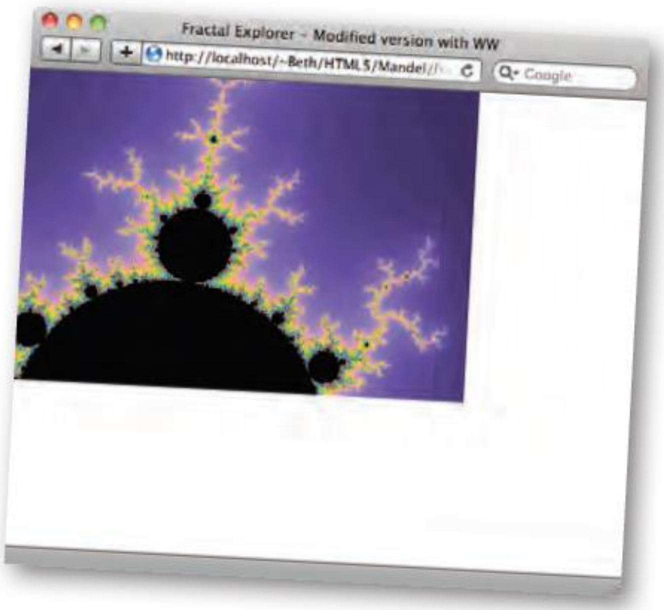
```
function handleClick(x, y) {
    var width = r_max - r_min;
    var height = i_min - i_max;
    var click_r = r_min + width * x / canvas.width;
    var click_i = i_max + height * y / canvas.height;

    var zoom = 8;

    r_min = click_r - width/zoom;
    r_max = click_r + width/zoom;
    i_max = click_i - height/zoom;
    i_min = click_i + height/zoom;

    startWorkers();
}
```

Now, we're ready to restart the workers.

실습과제 16-4 We can zoom!

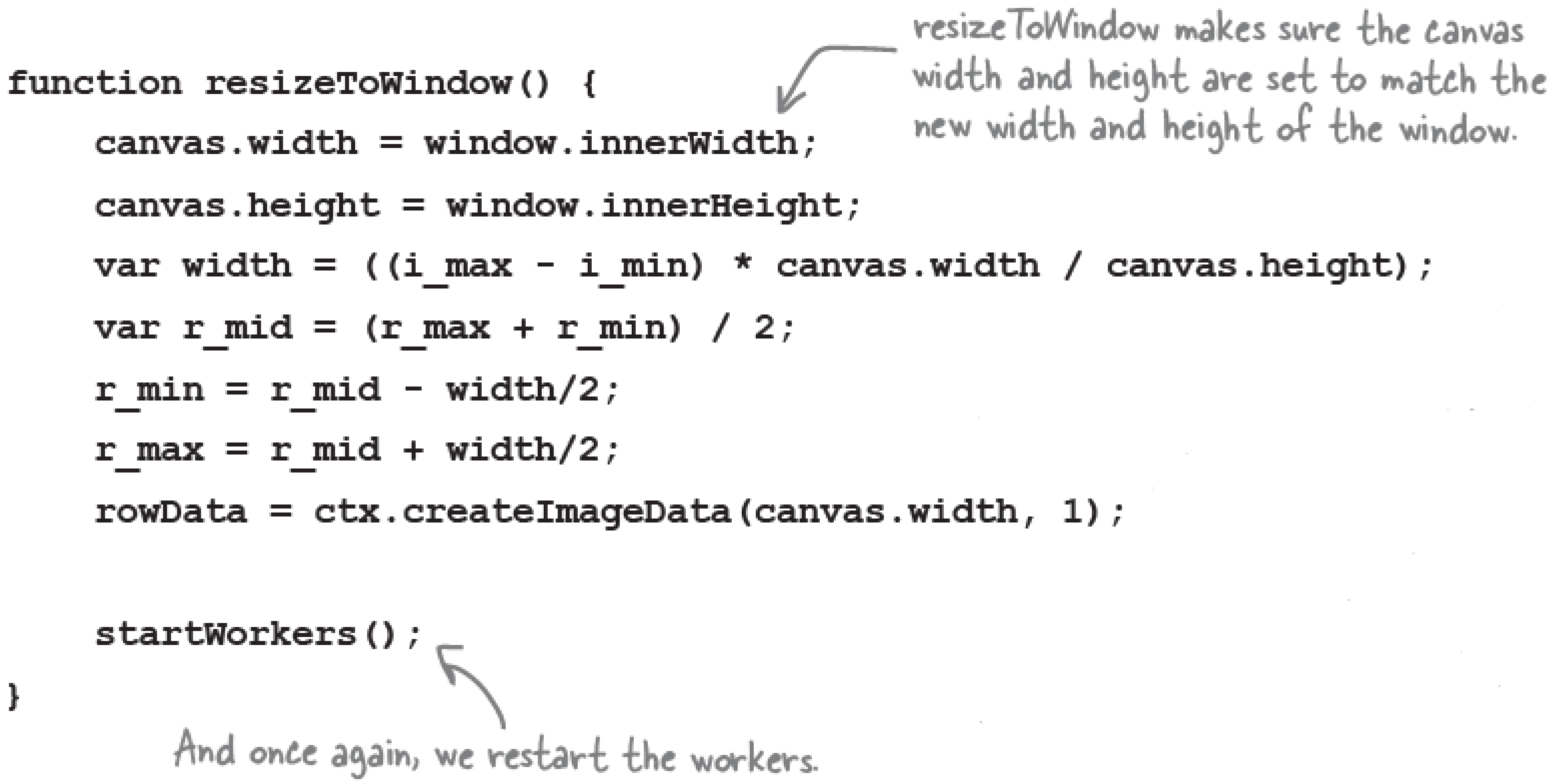
Let’s give those code changes a try. Reload **fractal.html** in your browser and this time click somewhere in the canvas. When you do you’ll see the workers start working on the zoomed-in view.

Nice! We can zoom, but we still need to resize the canvas to fit our window fully.


Fractal Explorer - Modified version with WW
http://localhost/~Beth/HTML5/Mandel/f
Google

http://ksamkeun.dothome.co.kr/wp/hfhtml5/ch10.1/16-4/

Fitting the canvas to the browser window

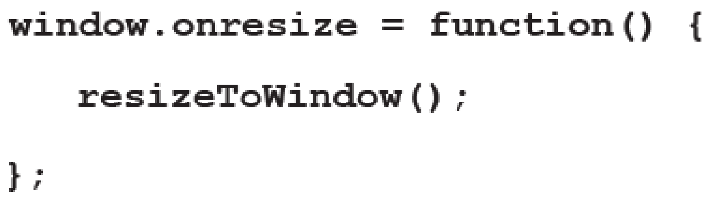


```
function resizeToWindow() {
    canvas.width = window.innerWidth;
    canvas.height = window.innerHeight;
    var width = ((i_max - i_min) * canvas.width / canvas.height);
    var r_mid = (r_max + r_min) / 2;
    r_min = r_mid - width/2;
    r_max = r_mid + width/2;
    rowData = ctx.createImageData(canvas.width, 1);

    startWorkers();
}
```

resizeToWindow makes sure the canvas width and height are set to match the new width and height of the window.

And once again, we restart the workers.

아래 코드를 “**mandel.js**”에서 init 함수의 **setUpGraphics** 호출 아래에 추가하자.

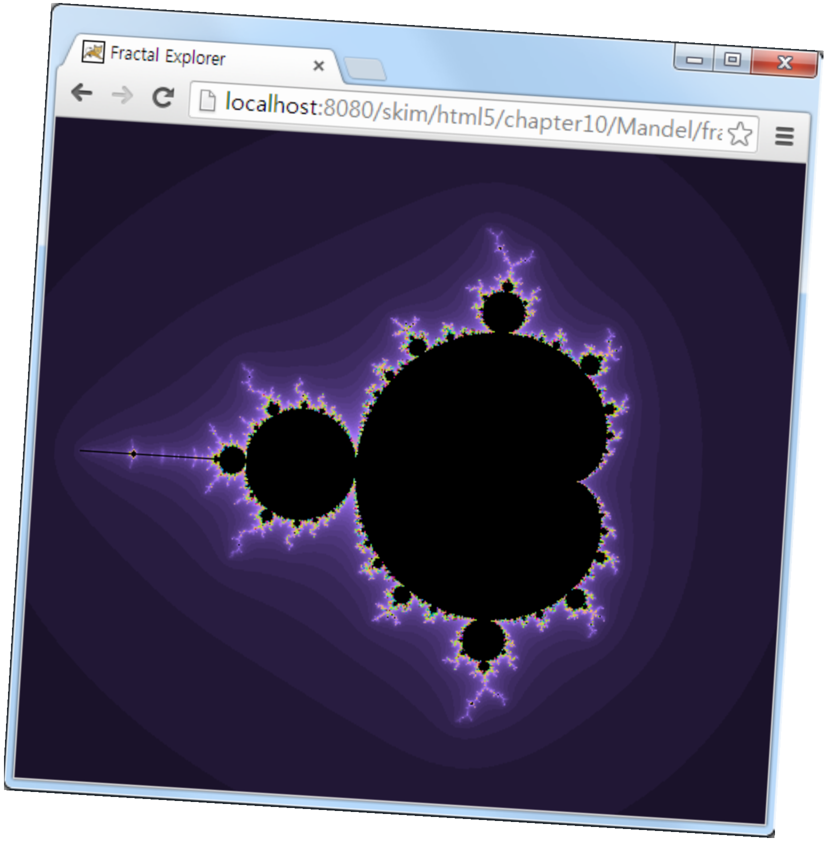


```
window.onresize = function() {
    resizeToWindow();
};
```

실습과제 16-5

Fitting the canvas to the browser window

Let's write the code to resize the canvas to the size of the browser window.



http://ksamkeun.dothome.co.kr/wp/hfhtml5/ch10.1/16-5/

The retentive ~~chef~~ coder

워커들이 자기에게 주어진 작업을 열심히 수행하고 있을 때, 사용자가 갑자기 스크린의 이미지를 확대할 수 있다.

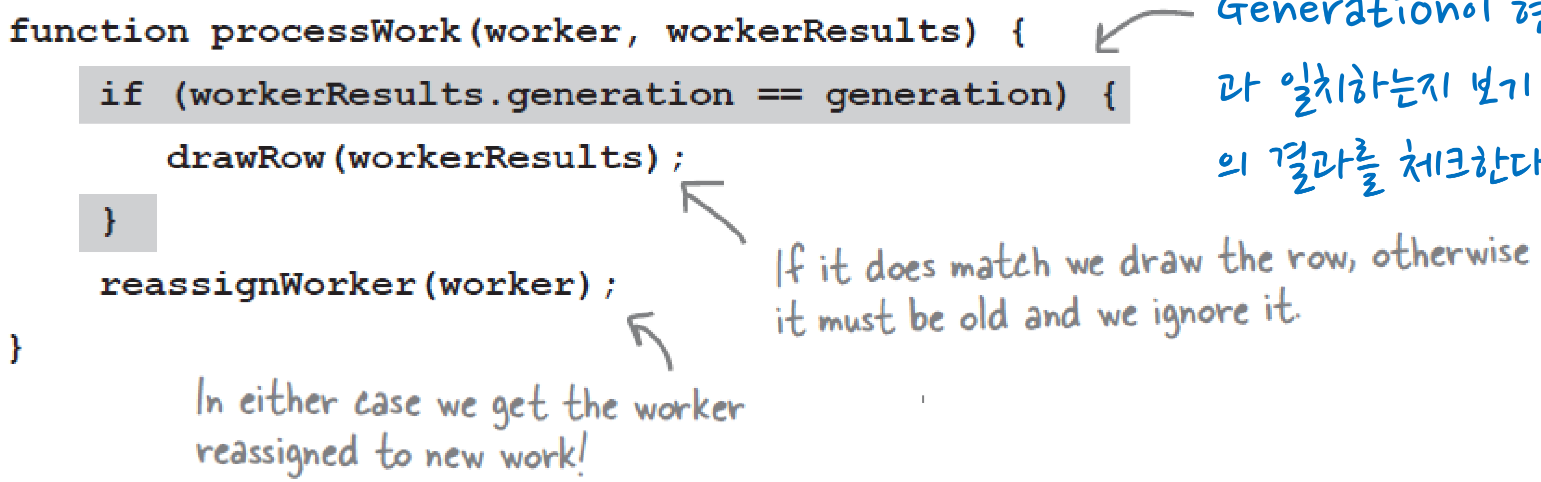
⇒ 이것은 워커들이 지금까지 수행한 작업들을 쓸모 없게 만들어 버린다

⇒ 설상가상으로 워커들은 사용자가 클릭을 했다는 사실조차도 모른다

⇒ 더욱 더 심각한 문제는 메인 페이지의 코드는 기꺼이 워커가 작업한 행을 받아서 디스플레이 한다는 것이다

이 모든 문제는 **사용자가 윈도우 크기를 재조정하면 발생한다.**

코드 수정:

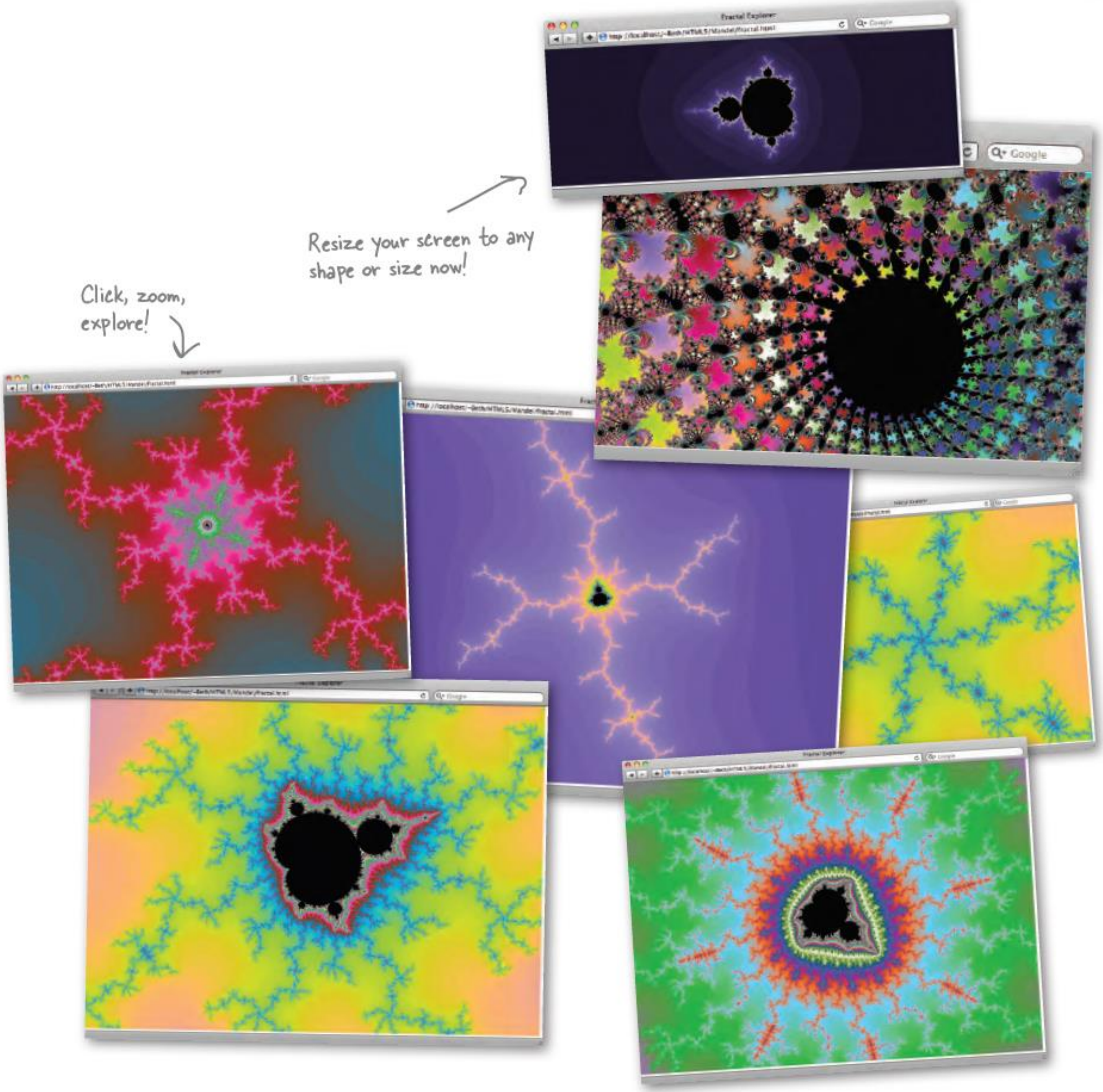


```
function processWork(worker, workerResults) {
    if (workerResults.generation == generation) {
        drawRow(workerResults);
    }
    reassignWorker(worker);
}
```

If it does match we draw the row, otherwise it must be old and we ignore it.

In either case we get the worker reassigned to new work!

Generation이 현재의 것과 일치하는지 보기 위해 워커의 결과를 체크한다!



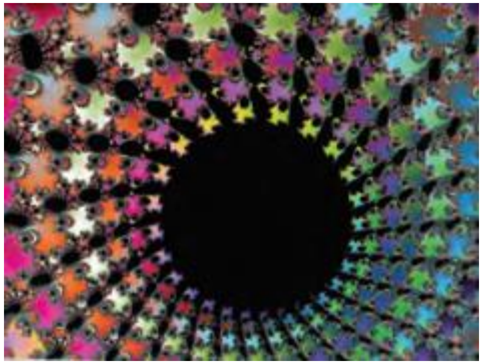
Resize your screen to any shape or size now!

Click, zoom, explore!

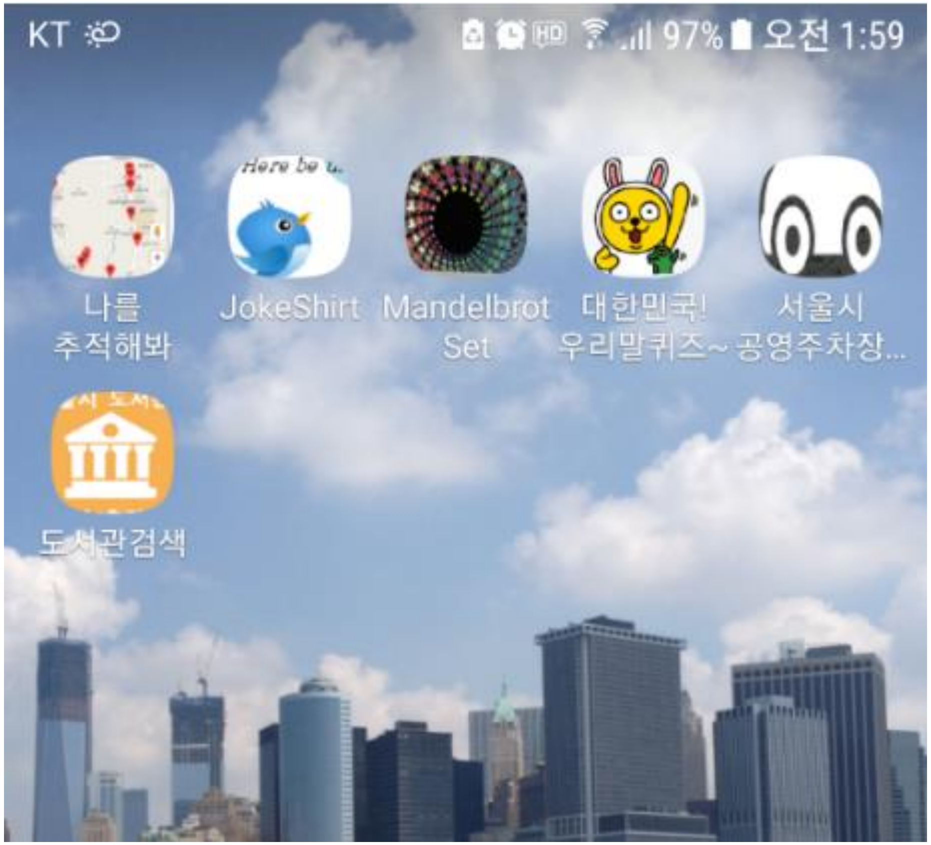
실습과제 16-6 모바일 바탕화면에 바로가기 만들기

지금까지 만들었던 웹앱 중 3개 이상의 앱에 대해 모바일 바탕화면에 바로가기를 만드시오.

```
<link rel="apple-touch-icon" sizes="180x180" href="/images/logo.png />
<link rel="icon" type="image/png" href="/images/logo.png sizes="192x192"/>
```



192x192



Q & A