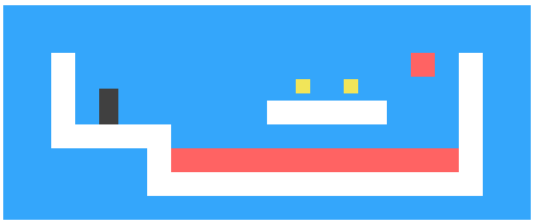
allowed range. Note that sometimes this will set nonsense scroll coordinates that are below zero or beyond the element's scrollable area. This is okay—the DOM will constrain them to acceptable values. Setting `scrollLeft` to -10 will cause it to become 0.

It would have been slightly simpler to always try to scroll the player to the center of the viewport. But this creates a rather jarring effect. As you are jumping, the view will constantly shift up and down. It is more pleasant to have a "neutral" area in the middle of the screen where you can move around without causing any scrolling.

We are now able to display our tiny level.

```
<link rel="stylesheet" href="css/game.css">

<script>
  let simpleLevel = new Level(simpleLevelPlan);
  let display = new DOMDisplay(document.body, simpleLevel);
  display.syncState(State.start(simpleLevel));
</script>
```



The `<link>` tag, when used with `rel="stylesheet"`, is a way to load a CSS file into a page. The file `game.css` contains the styles necessary for our game.

Motion and Collision

Now we're at the point where we can start adding motion—the most interesting aspect of the game. The basic approach, taken by most games like this, is to split time into small steps and, for each step, move the actors by a distance corresponding to their speed multiplied by the size of the time step. We'll measure time in seconds, so speeds are expressed in units per second.

Moving things is easy. The difficult part is dealing with the interactions between the elements. When the player hits a wall or floor, they should not simply move through it. The game must notice when a given motion causes an object to hit another object and respond accordingly. For walls, the motion must be stopped. When hitting a coin, it must be collected. When touching

lava, the game should be lost.

Solving this for the general case is a big task. You can find libraries, usually called *physics engines*, that simulate interaction between physical objects in two or three dimensions. We'll take a more modest approach in this chapter, handling only collisions between rectangular objects and handling them in a rather simplistic way.

Before moving the player or a block of lava, we test whether the motion would take it inside of a wall. If it does, we simply cancel the motion altogether. The response to such a collision depends on the type of actor—the player will stop, whereas a lava block will bounce back.

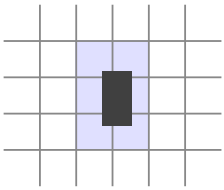
This approach requires our time steps to be rather small since it will cause motion to stop before the objects actually touch. If the time steps (and thus the motion steps) are too big, the player would end up hovering a noticeable distance above the ground. Another approach, arguably better but more complicated, would be to find the exact collision spot and move there. We will take the simple approach and hide its problems by ensuring the animation proceeds in small steps.

This method tells us whether a rectangle (specified by a position and a size) touches a grid element of the given type.

```
Level.prototype.touches = function(pos, size, type) {
  var xStart = Math.floor(pos.x);
  var xEnd = Math.ceil(pos.x + size.x);
  var yStart = Math.floor(pos.y);
  var yEnd = Math.ceil(pos.y + size.y);

  for (var y = yStart; y < yEnd; y++) {
    for (var x = xStart; x < xEnd; x++) {
      let isOutside = x < 0 || x >= this.width ||
                      y < 0 || y >= this.height;
      let here = isOutside ? "wall" : this.rows[y][x];
      if (here == type) return true;
    }
  }
  return false;
};
```

The method computes the set of grid squares that the body overlaps with by using `Math.floor` and `Math.ceil` on its coordinates. Remember that grid squares are 1 by 1 units in size. By rounding the sides of a box up and down, we get the range of background squares that the box touches.



We loop over the block of grid squares found by rounding the coordinates and return `true` when a matching square is found. Squares outside of the level are always treated as `"wall"` to ensure that the player can't leave the world and that we won't accidentally try to read outside of the bounds of our `rows` array.

The state `update` method uses `touches` to figure out whether the player is touching lava.

```
State.prototype.update = function(time, keys) {
  let actors = this.actors
    .map(actor => actor.update(time, this, keys));
  let newState = new State(this.level, actors, this.status);

  if (newState.status != "playing") return newState;

  let player = newState.player;
  if (this.level.touches(player.pos, player.size, "lava")) {
    return new State(this.level, actors, "lost");
  }

  for (let actor of actors) {
    if (actor != player && overlap(actor, player)) {
      newState = actor.collide(newState);
    }
  }
  return newState;
};
```

The method is passed a time step and a data structure that tells it which keys are being held down. The first thing it does is call the `update` method on all actors, producing an array of updated actors. The actors also get the time step, the keys, and the state, so that they can base their update on those. Only the player will actually read keys, since that's the only actor that's controlled by the keyboard.

If the game is already over, no further processing has to be done (the game can't be won after being lost, or vice versa). Otherwise, the method tests whether the player is touching background lava. If so, the game is lost, and

we're done. Finally, if the game really is still going on, it sees whether any other actors overlap the player.

Overlap between actors is detected with the `overlap` function. It takes two actor objects and returns true when they touch—which is the case when they overlap both along the x-axis and along the y-axis.

```
function overlap(actor1, actor2) {
  return actor1.pos.x + actor1.size.x > actor2.pos.x &&
         actor1.pos.x < actor2.pos.x + actor2.size.x &&
         actor1.pos.y + actor1.size.y > actor2.pos.y &&
         actor1.pos.y < actor2.pos.y + actor2.size.y;
}
```

If any actor does overlap, its `collide` method gets a chance to update the state. Touching a lava actor sets the game status to `"lost"`. Coins vanish when you touch them and set the status to `"won"` when they are the last coin of the level.

```
Lava.prototype.collide = function(state) {
  return new State(state.level, state.actors, "lost");
};

Coin.prototype.collide = function(state) {
  let filtered = state.actors.filter(a => a != this);
  let status = state.status;
  if (!filtered.some(a => a.type == "coin")) status = "won";
  return new State(state.level, filtered, status);
};
```

ACTOR UPDATES

Actor objects' `update` methods take as arguments the time step, the state object, and a `keys` object. The one for the `Lava` actor type ignores the `keys` object.

```
Lava.prototype.update = function(time, state) {
  let newPos = this.pos.plus(this.speed.times(time));
  if (!state.level.touches(newPos, this.size, "wall")) {
    return new Lava(newPos, this.speed, this.reset);
  } else if (this.reset) {
    return new Lava(this.reset, this.speed, this.reset);
```

```
  } else {
    return new Lava(this.pos, this.speed.times(-1));
  }
};
```

This `update` method computes a new position by adding the product of the time step and the current speed to its old position. If no obstacle blocks that new position, it moves there. If there is an obstacle, the behavior depends on the type of the lava block—dripping lava has a `reset` position, to which it jumps back when it hits something. Bouncing lava inverts its speed by multiplying it by -1 so that it starts moving in the opposite direction.

Coins use their `update` method to wobble. They ignore collisions with the grid since they are simply wobbling around inside of their own square.

```
const wobbleSpeed = 8, wobbleDist = 0.07;

Coin.prototype.update = function(time) {
  let wobble = this.wobble + time * wobbleSpeed;
  let wobblePos = Math.sin(wobble) * wobbleDist;
  return new Coin(this.basePos.plus(new Vec(0, wobblePos)),
                  this.basePos, wobble);
};
```

The `wobble` property is incremented to track time and then used as an argument to `Math.sin` to find the new position on the wave. The coin's current position is then computed from its base position and an offset based on this wave.

That leaves the player itself. Player motion is handled separately per axis because hitting the floor should not prevent horizontal motion, and hitting a wall should not stop falling or jumping motion.

```
const playerXSpeed = 7;
const gravity = 30;
const jumpSpeed = 17;

Player.prototype.update = function(time, state, keys) {
  let xSpeed = 0;
  if (keys.ArrowLeft) xSpeed -= playerXSpeed;
  if (keys.ArrowRight) xSpeed += playerXSpeed;
  let pos = this.pos;
  let movedX = pos.plus(new Vec(xSpeed * time, 0));
  if (!state.level.touches(movedX, this.size, "wall")) {
```

```
      pos = movedX;
    }

    let ySpeed = this.speed.y + time * gravity;
    let movedY = pos.plus(new Vec(0, ySpeed * time));
    if (!state.level.touches(movedY, this.size, "wall")) {
      pos = movedY;
    } else if (keys.ArrowUp && ySpeed > 0) {
      ySpeed = -jumpSpeed;
    } else {
      ySpeed = 0;
    }
    return new Player(pos, new Vec(xSpeed, ySpeed));
  };
```

The horizontal motion is computed based on the state of the left and right arrow keys. When there's no wall blocking the new position created by this motion, it is used. Otherwise, the old position is kept.

Vertical motion works in a similar way but has to simulate jumping and gravity. The player's vertical speed (`ySpeed`) is first accelerated to account for gravity.

We check for walls again. If we don't hit any, the new position is used. If there *is* a wall, there are two possible outcomes. When the up arrow is pressed *and* we are moving down (meaning the thing we hit is below us), the speed is set to a relatively large, negative value. This causes the player to jump. If that is not the case, the player simply bumped into something, and the speed is set to zero.

The gravity strength, jumping speed, and pretty much all other constants in this game have been set by trial and error. I tested values until I found a combination I liked.

Tracking keys

For a game like this, we do not want keys to take effect once per keypress. Rather, we want their effect (moving the player figure) to stay active as long as they are held.

We need to set up a key handler that stores the current state of the left, right, and up arrow keys. We will also want to call `preventDefault` for those keys so that they don't end up scrolling the page.

The following function, when given an array of key names, will return an object that tracks the current position of those keys. It registers event handlers

for "keydown" and "keyup" events and, when the key code in the event is present in the set of codes that it is tracking, updates the object.

```
function trackKeys(keys) {
  let down = Object.create(null);
  function track(event) {
    if (keys.includes(event.key)) {
      down[event.key] = event.type == "keydown";
      event.preventDefault();
    }
  }
  window.addEventListener("keydown", track);
  window.addEventListener("keyup", track);
  return down;
}

const arrowKeys =
  trackKeys(["ArrowLeft", "ArrowRight", "ArrowUp"]);
```

The same handler function is used for both event types. It looks at the event object's `type` property to determine whether the key state should be updated to true (`"keydown"`) or false (`"keyup"`).

Running the game

The `requestAnimationFrame` function, which we saw in Chapter 14, provides a good way to animate a game. But its interface is quite primitive—using it requires us to track the time at which our function was called the last time around and call `requestAnimationFrame` again after every frame.

Let's define a helper function that wraps those boring parts in a convenient interface and allows us to simply call `runAnimation`, giving it a function that expects a time difference as an argument and draws a single frame. When the frame function returns the value `false`, the animation stops.

```
function runAnimation(frameFunc) {
  let lastTime = null;
  function frame(time) {
    if (lastTime != null) {
      let timeStep = Math.min(time - lastTime, 100) / 1000;
      if (frameFunc(timeStep) === false) return;
    }
    lastTime = time;
    requestAnimationFrame(frame);
```

```
    }
    requestAnimationFrame(frame);
  }
```

I have set a maximum frame step of 100 milliseconds (one-tenth of a second). When the browser tab or window with our page is hidden, `requestAnimationFrame` calls will be suspended until the tab or window is shown again. In this case, the difference between `lastTime` and `time` will be the entire time in which the page was hidden. Advancing the game by that much in a single step would look silly and might cause weird side effects, such as the player falling through the floor.

The function also converts the time steps to seconds, which are an easier quantity to think about than milliseconds.

The `runLevel` function takes a `Level` object and a display constructor and returns a promise. It displays the level (in `document.body`) and lets the user play through it. When the level is finished (lost or won), `runLevel` waits one more second (to let the user see what happens) and then clears the display, stops the animation, and resolves the promise to the game's end status.

```
function runLevel(level, Display) {
  let display = new Display(document.body, level);
  let state = State.start(level);
  let ending = 1;
  return new Promise(resolve => {
    runAnimation(time => {
      state = state.update(time, arrowKeys);
      display.syncState(state);
      if (state.status == "playing") {
        return true;
      } else if (ending > 0) {
        ending -= time;
        return true;
      } else {
        display.clear();
        resolve(state.status);
        return false;
      }
    });
  });
}
```

A game is a sequence of levels. Whenever the player dies, the current level

is restarted. When a level is completed, we move on to the next level. This can be expressed by the following function, which takes an array of level plans (strings) and a display constructor:

```
async function runGame(plans, Display) {
  for (let level = 0; level < plans.length;) {
    let status = await runLevel(new Level(plans[level]),
                                Display);
    if (status == "won") level++;
  }
  console.log("You've won!");
}
```

Because we made `runLevel` return a promise, `runGame` can be written using an `async` function, as shown in Chapter 11. It returns another promise, which resolves when the player finishes the game.

There is a set of level plans available in the `GAME_LEVELS` binding in this chapter's sandbox (*https://eloquentjavascript.net/code#16*). This page feeds them to `runGame`, starting an actual game.

```
<link rel="stylesheet" href="css/game.css">

<body>
  <script>
    runGame(GAME_LEVELS, DOMDisplay);
  </script>
</body>
```

Exercises

Game over

It's traditional for platform games to have the player start with a limited number of *lives* and subtract one life each time they die. When the player is out of lives, the game restarts from the beginning.

Adjust `runGame` to implement lives. Have the player start with three. Output the current number of lives (using `console.log`) every time a level starts.

Pausing the game

Make it possible to pause (suspend) and unpause the game by pressing the Esc key.

This can be done by changing the `runLevel` function to use another keyboard event handler and interrupting or resuming the animation whenever the Esc key is hit.

The `runAnimation` interface may not look like it is suitable for this at first glance, but it is if you rearrange the way `runLevel` calls it.

When you have that working, there is something else you could try. The way we have been registering keyboard event handlers is somewhat problematic. The `arrowKeys` object is currently a global binding, and its event handlers are kept around even when no game is running. You could say they *leak* out of our system. Extend `trackKeys` to provide a way to unregister its handlers and then change `runLevel` to register its handlers when it starts and unregister them again when it is finished.

A monster

It is traditional for platform games to have enemies that you can jump on top of to defeat. This exercise asks you to add such an actor type to the game.

We'll call it a monster. Monsters move only horizontally. You can make them move in the direction of the player, bounce back and forth like horizontal lava, or have any movement pattern you want. The class doesn't have to handle falling, but it should make sure the monster doesn't walk through walls.

When a monster touches the player, the effect depends on whether the player is jumping on top of them or not. You can approximate this by checking whether the player's bottom is near the monster's top. If this is the case, the monster disappears. If not, the game is lost.

"Drawing is deception."

—M.C. Escher, cited by Bruno Ernst in The Magic Mirror of M.C. Escher

CHAPTER 17

Drawing on Canvas

Browsers give us several ways to display graphics. The simplest way is to use styles to position and color regular DOM elements. This can get you quite far, as the game in the previous chapter showed. By adding partially transparent background images to the nodes, we can make them look exactly the way we want. It is even possible to rotate or skew nodes with the `transform` style.

But we'd be using the DOM for something that it wasn't originally designed for. Some tasks, such as drawing a line between arbitrary points, are extremely awkward to do with regular HTML elements.

There are two alternatives. The first is DOM-based but utilizes *Scalable Vector Graphics* (SVG), rather than HTML. Think of SVG as a document-markup dialect that focuses on shapes rather than text. You can embed an SVG document directly in an HTML document or include it with an `<img>` tag.

The second alternative is called a *canvas*. A canvas is a single DOM element that encapsulates a picture. It provides a programming interface for drawing shapes onto the space taken up by the node. The main difference between a canvas and an SVG picture is that in SVG the original description of the shapes is preserved so that they can be moved or resized at any time. A canvas, on the other hand, converts the shapes to pixels (colored dots on a raster) as soon as they are drawn and does not remember what these pixels represent. The only way to move a shape on a canvas is to clear the canvas (or the part of the canvas around the shape) and redraw it with the shape in a new position.

SVG

This book will not go into SVG in detail, but I will briefly explain how it works. At the end of the chapter, I'll come back to the trade-offs that you must consider when deciding which drawing mechanism is appropriate for a given application.

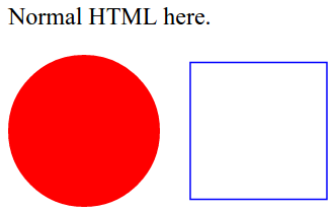
This is an HTML document with a simple SVG picture in it:

```
<p>Normal HTML here.</p>
```

```
<svg xmlns="http://www.w3.org/2000/svg">
  <circle r="50" cx="50" cy="50" fill="red"/>
  <rect x="120" y="5" width="90" height="90"
        stroke="blue" fill="none"/>
</svg>
```

The `xmlns` attribute changes an element (and its children) to a different *XML namespace*. This namespace, identified by a URL, specifies the dialect that we are currently speaking. The `<circle>` and `<rect>` tags, which do not exist in HTML, do have a meaning in SVG—they draw shapes using the style and position specified by their attributes.

The document is displayed like this:



These tags create DOM elements, just like HTML tags, that scripts can interact with. For example, this changes the `<circle>` element to be colored cyan instead:

```
let circle = document.querySelector("circle");
circle.setAttribute("fill", "cyan");
```

The canvas element

Canvas graphics can be drawn onto a `<canvas>` element. You can give such an element `width` and `height` attributes to determine its size in pixels.

A new canvas is empty, meaning it is entirely transparent and thus shows up as empty space in the document.

The `<canvas>` tag is intended to allow different styles of drawing. To get access to an actual drawing interface, we first need to create a *context*, an object whose methods provide the drawing interface. There are currently two widely supported drawing styles: `"2d"` for two-dimensional graphics and `"webgl"` for three-dimensional graphics through the OpenGL interface.

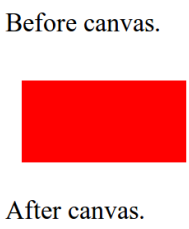
This book won't discuss WebGL—we'll stick to two dimensions. But if you are interested in three-dimensional graphics, I do encourage you to look into

WebGL. It provides a direct interface to graphics hardware and allows you to render even complicated scenes efficiently, using JavaScript.

You create a context with the `getContext` method on the `<canvas>` DOM element.

```
<p>Before canvas.</p>
<canvas width="120" height="60"></canvas>
<p>After canvas.</p>
<script>
  let canvas = document.querySelector("canvas");
  let context = canvas.getContext("2d");
  context.fillStyle = "red";
  context.fillRect(10, 10, 100, 50);
</script>
```

After creating the context object, the example draws a red rectangle 100 pixels wide and 50 pixels high, with its top-left corner at coordinates (10,10).



Just like in HTML (and SVG), the coordinate system that the canvas uses puts (0,0) at the top-left corner, and the positive y-axis goes down from there. So (10,10) is 10 pixels below and to the right of the top-left corner.

Lines and surfaces

In the canvas interface, a shape can be *filled*, meaning its area is given a certain color or pattern, or it can be *stroked*, which means a line is drawn along its edge. The same terminology is used by SVG.

The `fillRect` method fills a rectangle. It takes first the x- and y-coordinates of the rectangle's top-left corner, then its width, and then its height. A similar method, `strokeRect`, draws the outline of a rectangle.

Neither method takes any further parameters. The color of the fill, thickness of the stroke, and so on, are not determined by an argument to the method (as you might reasonably expect) but rather by properties of the context object.

The `fillStyle` property controls the way shapes are filled. It can be set to a string that specifies a color, using the color notation used by CSS.

The `strokeStyle` property works similarly but determines the color used for a stroked line. The width of that line is determined by the `lineWidth` property, which may contain any positive number.

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.strokeStyle = "blue";
  cx.strokeRect(5, 5, 50, 50);
  cx.lineWidth = 5;
  cx.strokeRect(135, 5, 50, 50);
</script>
```

This code draws two blue squares, using a thicker line for the second one.

When no `width` or `height` attribute is specified, as in the example, a canvas element gets a default width of 300 pixels and height of 150 pixels.

Paths

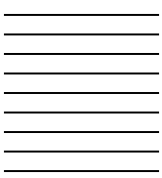
A path is a sequence of lines. The 2D canvas interface takes a peculiar approach to describing such a path. It is done entirely through side effects. Paths are not values that can be stored and passed around. Instead, if you want to do something with a path, you make a sequence of method calls to describe its shape.

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.beginPath();
  for (let y = 10; y < 100; y += 10) {
    cx.moveTo(10, y);
    cx.lineTo(90, y);
  }
  cx.stroke();
</script>
```

This example creates a path with a number of horizontal line segments and

then strokes it using the `stroke` method. Each segment created with `lineTo` starts at the path's *current* position. That position is usually the end of the last segment, unless `moveTo` was called. In that case, the next segment would start at the position passed to `moveTo`.

The path described by the previous program looks like this:



When filling a path (using the `fill` method), each shape is filled separately. A path can contain multiple shapes—each `moveTo` motion starts a new one. But the path needs to be *closed* (meaning its start and end are in the same position) before it can be filled. If the path is not already closed, a line is added from its end to its start, and the shape enclosed by the completed path is filled.

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.beginPath();
  cx.moveTo(50, 10);
  cx.lineTo(10, 70);
  cx.lineTo(90, 70);
  cx.fill();
</script>
```

This example draws a filled triangle. Note that only two of the triangle's sides are explicitly drawn. The third, from the bottom-right corner back to the top, is implied and wouldn't be there when you stroke the path.

You could also use the `closePath` method to explicitly close a path by adding an actual line segment back to the path's start. This segment *is* drawn when stroking the path.

Curves

A path may also contain curved lines. These are unfortunately a bit more involved to draw.

The `quadraticCurveTo` method draws a curve to a given point. To determine the curvature of the line, the method is given a control point as well as a destination point. Imagine this control point as *attracting* the line, giving it its curve. The line won't go through the control point, but its direction at the start and end points will be such that a straight line in that direction would point toward the control point. The following example illustrates this:

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.beginPath();
  cx.moveTo(10, 90);
  // control=(60,10) goal=(90,90)
  cx.quadraticCurveTo(60, 10, 90, 90);
  cx.lineTo(60, 10);
  cx.closePath();
  cx.stroke();
</script>
```

It produces a path that looks like this:

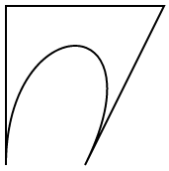
We draw a quadratic curve from the left to the right, with (60,10) as control point, and then draw two line segments going through that control point and back to the start of the line. The result somewhat resembles a *Star Trek* insignia. You can see the effect of the control point: the lines leaving the lower corners start off in the direction of the control point and then curve toward their target.

The `bezierCurveTo` method draws a similar kind of curve. Instead of a single control point, this one has two—one for each of the line's endpoints. Here is a similar sketch to illustrate the behavior of such a curve:

```
<canvas></canvas>
<script>
```

```
  let cx = document.querySelector("canvas").getContext("2d");
  cx.beginPath();
  cx.moveTo(10, 90);
  // control1=(10,10) control2=(90,10) goal=(50,90)
  cx.bezierCurveTo(10, 10, 90, 10, 50, 90);
  cx.lineTo(90, 10);
  cx.lineTo(10, 10);
  cx.closePath();
  cx.stroke();
</script>
```

The two control points specify the direction at both ends of the curve. The farther they are away from their corresponding point, the more the curve will "bulge" in that direction.



Such curves can be hard to work with—it's not always clear how to find the control points that provide the shape you are looking for. Sometimes you can compute them, and sometimes you'll just have to find a suitable value by trial and error.

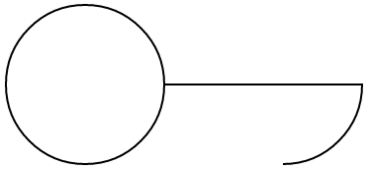
The `arc` method is a way to draw a line that curves along the edge of a circle. It takes a pair of coordinates for the arc's center, a radius, and then a start angle and end angle.

Those last two parameters make it possible to draw only part of the circle. The angles are measured in radians, not degrees. This means a full circle has an angle of 2π, or `2 * Math.PI`, which is about 6.28. The angle starts counting at the point to the right of the circle's center and goes clockwise from there. You can use a start of 0 and an end bigger than 2π (say, 7) to draw a full circle.

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.beginPath();
  // center=(50,50) radius=40 angle=0 to 7
  cx.arc(50, 50, 40, 0, 7);
  // center=(150,50) radius=40 angle=0 to π½
  cx.arc(150, 50, 40, 0, 0.5 * Math.PI);
  cx.stroke();
```

```
</script>
```

The resulting picture contains a line from the right of the full circle (first call to `arc`) to the right of the quarter-circle (second call). Like other path-drawing methods, a line drawn with `arc` is connected to the previous path segment. You can call `moveTo` or start a new path to avoid this.



Drawing a pie chart

Imagine you've just taken a job at EconomiCorp, Inc., and your first assignment is to draw a pie chart of its customer satisfaction survey results.

The `results` binding contains an array of objects that represent the survey responses.

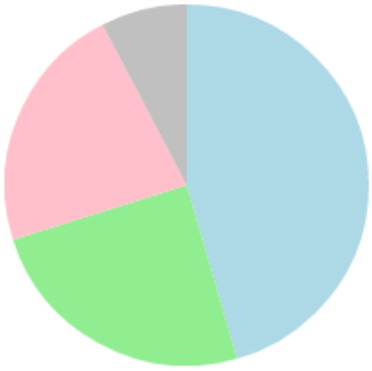
```
const results = [
  {name: "Satisfied", count: 1043, color: "lightblue"},
  {name: "Neutral", count: 563, color: "lightgreen"},
  {name: "Unsatisfied", count: 510, color: "pink"},
  {name: "No comment", count: 175, color: "silver"}
];
```

To draw a pie chart, we draw a number of pie slices, each made up of an arc and a pair of lines to the center of that arc. We can compute the angle taken up by each arc by dividing a full circle (2π) by the total number of responses and then multiplying that number (the angle per response) by the number of people who picked a given choice.

```
<canvas width="200" height="200"></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  let total = results
    .reduce((sum, {count}) => sum + count, 0);
  // Start at the top
  let currentAngle = -0.5 * Math.PI;
  for (let result of results) {
```

```
    let sliceAngle = (result.count / total) * 2 * Math.PI;
    cx.beginPath();
    // center=100,100, radius=100
    // from current angle, clockwise by slice's angle
    cx.arc(100, 100, 100,
           currentAngle, currentAngle + sliceAngle);
    currentAngle += sliceAngle;
    cx.lineTo(100, 100);
    cx.fillStyle = result.color;
    cx.fill();
  }
</script>
```

This draws the following chart:



But a chart that doesn't tell us what the slices mean isn't very helpful. We need a way to draw text to the canvas.

Text

A 2D canvas drawing context provides the methods `fillText` and `strokeText`. The latter can be useful for outlining letters, but usually `fillText` is what you need. It will fill the outline of the given text with the current `fillStyle`.

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.font = "28px Georgia";
  cx.fillStyle = "fuchsia";
  cx.fillText("I can draw text, too!", 10, 50);
</script>
```

You can specify the size, style, and font of the text with the `font` property. This example just gives a font size and family name. It is also possible to add `italic` or `bold` to the start of the string to select a style.

The last two arguments to `fillText` and `strokeText` provide the position at which the font is drawn. By default, they indicate the position of the start of the text's alphabetic baseline, which is the line that letters "stand" on, not counting hanging parts in letters such as *j* or *p*. You can change the horizontal position by setting the `textAlign` property to `"end"` or `"center"` and the vertical position by setting `textBaseline` to `"top"`, `"middle"`, or `"bottom"`.

We'll come back to our pie chart, and the problem of labeling the slices, in the exercises at the end of the chapter.

Images

In computer graphics, a distinction is often made between *vector* graphics and *bitmap* graphics. The first is what we have been doing so far in this chapter—specifying a picture by giving a logical description of shapes. Bitmap graphics, on the other hand, don't specify actual shapes but rather work with pixel data (rasters of colored dots).

The `drawImage` method allows us to draw pixel data onto a canvas. This pixel data can originate from an `<img>` element or from another canvas. The following example creates a detached `<img>` element and loads an image file into it. But it cannot immediately start drawing from this picture because the browser may not have loaded it yet. To deal with this, we register a `"load"` event handler and do the drawing after the image has loaded.

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  let img = document.createElement("img");
  img.src = "img/hat.png";
  img.addEventListener("load", () => {
    for (let x = 10; x < 200; x += 30) {
      cx.drawImage(img, x, 10);
    }
  });
</script>
```

By default, `drawImage` will draw the image at its original size. You can also give it two additional arguments to set a different width and height.

When `drawImage` is given *nine* arguments, it can be used to draw only a fragment of an image. The second through fifth arguments indicate the rectangle (x, y, width, and height) in the source image that should be copied, and the sixth to ninth arguments give the rectangle (on the canvas) into which it should be copied.

This can be used to pack multiple *sprites* (image elements) into a single image file and then draw only the part you need. For example, we have this picture containing a game character in multiple poses:

By alternating which pose we draw, we can show an animation that looks like a walking character.

To animate a picture on a canvas, the `clearRect` method is useful. It resembles `fillRect`, but instead of coloring the rectangle, it makes it transparent, removing the previously drawn pixels.

We know that each *sprite*, each subpicture, is 24 pixels wide and 30 pixels high. The following code loads the image and then sets up an interval (repeated timer) to draw the next frame:

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  let img = document.createElement("img");
  img.src = "img/player.png";
  let spriteW = 24, spriteH = 30;
  img.addEventListener("load", () => {
    let cycle = 0;
    setInterval(() => {
      cx.clearRect(0, 0, spriteW, spriteH);
      cx.drawImage(img,
                   // source rectangle
                   cycle * spriteW, 0, spriteW, spriteH,
                   // destination rectangle
                   0,               0, spriteW, spriteH);
      cycle = (cycle + 1) % 8;
    }, 120);
  });
</script>
```

The `cycle` binding tracks our position in the animation. For each frame, it is incremented and then clipped back to the 0 to 7 range by using the remainder

operator. This binding is then used to compute the x-coordinate that the sprite for the current pose has in the picture.

Transformation

But what if we want our character to walk to the left instead of to the right? We could draw another set of sprites, of course. But we can also instruct the canvas to draw the picture the other way round.

Calling the `scale` method will cause anything drawn after it to be scaled. This method takes two parameters, one to set a horizontal scale and one to set a vertical scale.

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  cx.scale(3, .5);
  cx.beginPath();
  cx.arc(50, 50, 40, 0, 7);
  cx.lineWidth = 3;
  cx.stroke();
</script>
```

Because of the call to `scale`, the circle is drawn three times as wide and half as high.

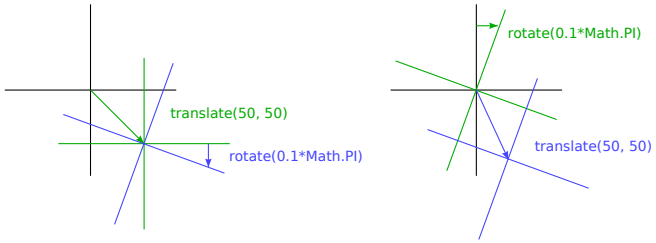
Scaling will cause everything about the drawn image, including the line width, to be stretched out or squeezed together as specified. Scaling by a negative amount will flip the picture around. The flipping happens around point (0,0), which means it will also flip the direction of the coordinate system. When a horizontal scaling of -1 is applied, a shape drawn at x position 100 will end up at what used to be position -100.

So to turn a picture around, we can't simply add `cx.scale(-1, 1)` before the call to `drawImage` because that would move our picture outside of the canvas, where it won't be visible. You could adjust the coordinates given to `drawImage` to compensate for this by drawing the image at x position -50 instead of 0. Another solution, which doesn't require the code that does the drawing to know about the scale change, is to adjust the axis around which the scaling happens.

There are several other methods besides `scale` that influence the coordinate

system for a canvas. You can rotate subsequently drawn shapes with the `rotate` method and move them with the `translate` method. The interesting—and confusing—thing is that these transformations *stack*, meaning that each one happens relative to the previous transformations.

So if we translate by 10 horizontal pixels twice, everything will be drawn 20 pixels to the right. If we first move the center of the coordinate system to (50,50) and then rotate by 20 degrees (about 0.1π radians), that rotation will happen *around* point (50,50).

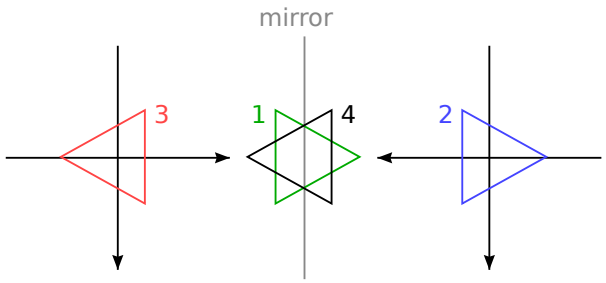


But if we *first* rotate by 20 degrees and *then* translate by (50,50), the translation will happen in the rotated coordinate system and thus produce a different orientation. The order in which transformations are applied matters.

To flip a picture around the vertical line at a given x position, we can do the following:

```
function flipHorizontally(context, around) {
  context.translate(around, 0);
  context.scale(-1, 1);
  context.translate(-around, 0);
}
```

We move the y-axis to where we want our mirror to be, apply the mirroring, and finally move the y-axis back to its proper place in the mirrored universe. The following picture explains why this works:



This shows the coordinate systems before and after mirroring across the central line. The triangles are numbered to illustrate each step. If we draw a

triangle at a positive x position, it would, by default, be in the place where triangle 1 is. A call to `flipHorizontally` first does a translation to the right, which gets us to triangle 2. It then scales, flipping the triangle over to position 3. This is not where it should be, if it were mirrored in the given line. The second `translate` call fixes this—it "cancels" the initial translation and makes triangle 4 appear exactly where it should.

We can now draw a mirrored character at position (100,0) by flipping the world around the character's vertical center.

```
<canvas></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  let img = document.createElement("img");
  img.src = "img/player.png";
  let spriteW = 24, spriteH = 30;
  img.addEventListener("load", () => {
    flipHorizontally(cx, 100 + spriteW / 2);
    cx.drawImage(img, 0, 0, spriteW, spriteH,
                 100, 0, spriteW, spriteH);
  });
</script>
```

Storing and clearing transformations

Transformations stick around. Everything else we draw after drawing that mirrored character would also be mirrored. That might be inconvenient.

It is possible to save the current transformation, do some drawing and transforming, and then restore the old transformation. This is usually the proper thing to do for a function that needs to temporarily transform the coordinate system. First, we save whatever transformation the code that called the function was using. Then the function does its thing, adding more transformations on top of the current transformation. Finally, we revert to the transformation we started with.

The `save` and `restore` methods on the 2D canvas context do this transformation management. They conceptually keep a stack of transformation states. When you call `save`, the current state is pushed onto the stack, and when you call `restore`, the state on top of the stack is taken off and used as the context's current transformation. You can also call `resetTransform` to fully reset the transformation.

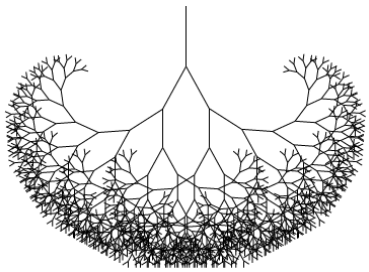
The `branch` function in the following example illustrates what you can do

with a function that changes the transformation and then calls a function (in this case itself), which continues drawing with the given transformation.

This function draws a treelike shape by drawing a line, moving the center of the coordinate system to the end of the line, and calling itself twice—first rotated to the left and then rotated to the right. Every call reduces the length of the branch drawn, and the recursion stops when the length drops below 8.

```
<canvas width="600" height="300"></canvas>
<script>
  let cx = document.querySelector("canvas").getContext("2d");
  function branch(length, angle, scale) {
    cx.fillRect(0, 0, 1, length);
    if (length < 8) return;
    cx.save();
    cx.translate(0, length);
    cx.rotate(-angle);
    branch(length * scale, angle, scale);
    cx.rotate(2 * angle);
    branch(length * scale, angle, scale);
    cx.restore();
  }
  cx.translate(300, 0);
  branch(60, 0.5, 0.8);
</script>
```

The result is a simple fractal.



If the calls to `save` and `restore` were not there, the second recursive call to `branch` would end up with the position and rotation created by the first call. It wouldn't be connected to the current branch but rather to the innermost, rightmost branch drawn by the first call. The resulting shape might also be interesting, but it is definitely not a tree.

Back to the game

We now know enough about canvas drawing to start working on a canvas-based display system for the game from the previous chapter. The new display will no longer be showing just colored boxes. Instead, we'll use `drawImage` to draw pictures that represent the game's elements.

We define another display object type called `CanvasDisplay`, supporting the same interface as `DOMDisplay` from Chapter 16, namely, the methods `syncState` and `clear`.

This object keeps a little more information than `DOMDisplay`. Rather than using the scroll position of its DOM element, it tracks its own viewport, which tells us what part of the level we are currently looking at. Finally, it keeps a `flipPlayer` property so that even when the player is standing still, it keeps facing the direction it last moved in.

```
class CanvasDisplay {
  constructor(parent, level) {
    this.canvas = document.createElement("canvas");
    this.canvas.width = Math.min(600, level.width * scale);
    this.canvas.height = Math.min(450, level.height * scale);
    parent.appendChild(this.canvas);
    this.cx = this.canvas.getContext("2d");

    this.flipPlayer = false;

    this.viewport = {
      left: 0,
      top: 0,
      width: this.canvas.width / scale,
      height: this.canvas.height / scale
    };
  }

  clear() {
    this.canvas.remove();
  }
}
```

The `syncState` method first computes a new viewport and then draws the game scene at the appropriate position.

```
CanvasDisplay.prototype.syncState = function(state) {
  this.updateViewport(state);
```

```
  this.clearDisplay(state.status);
  this.drawBackground(state.level);
  this.drawActors(state.actors);
};
```

Contrary to `DOMDisplay`, this display style *does* have to redraw the background on every update. Because shapes on a canvas are just pixels, after we draw them there is no good way to move them (or remove them). The only way to update the canvas display is to clear it and redraw the scene. We may also have scrolled, which requires the background to be in a different position.

The `updateViewport` method is similar to `DOMDisplay`'s `scrollPlayerIntoView` method. It checks whether the player is too close to the edge of the screen and moves the viewport when this is the case.

```
CanvasDisplay.prototype.updateViewport = function(state) {
  let view = this.viewport, margin = view.width / 3;
  let player = state.player;
  let center = player.pos.plus(player.size.times(0.5));

  if (center.x < view.left + margin) {
    view.left = Math.max(center.x - margin, 0);
  } else if (center.x > view.left + view.width - margin) {
    view.left = Math.min(center.x + margin - view.width,
                         state.level.width - view.width);
  }
  if (center.y < view.top + margin) {
    view.top = Math.max(center.y - margin, 0);
  } else if (center.y > view.top + view.height - margin) {
    view.top = Math.min(center.y + margin - view.height,
                        state.level.height - view.height);
  }
};
```

The calls to `Math.max` and `Math.min` ensure that the viewport does not end up showing space outside of the level. `Math.max(x, 0)` makes sure the resulting number is not less than zero. `Math.min` similarly guarantees that a value stays below a given bound.

When clearing the display, we'll use a slightly different color depending on whether the game is won (brighter) or lost (darker).

```
CanvasDisplay.prototype.clearDisplay = function(status) {
  if (status == "won") {
```

```
    this.cx.fillStyle = "rgb(68, 191, 255)";
  } else if (status == "lost") {
    this.cx.fillStyle = "rgb(44, 136, 214)";
  } else {
    this.cx.fillStyle = "rgb(52, 166, 251)";
  }
  this.cx.fillRect(0, 0,
                   this.canvas.width, this.canvas.height);
};
```

To draw the background, we run through the tiles that are visible in the current viewport, using the same trick used in the `touches` method from the previous chapter.

```
let otherSprites = document.createElement("img");
otherSprites.src = "img/sprites.png";

CanvasDisplay.prototype.drawBackground = function(level) {
  let {left, top, width, height} = this.viewport;
  let xStart = Math.floor(left);
  let xEnd = Math.ceil(left + width);
  let yStart = Math.floor(top);
  let yEnd = Math.ceil(top + height);

  for (let y = yStart; y < yEnd; y++) {
    for (let x = xStart; x < xEnd; x++) {
      let tile = level.rows[y][x];
      if (tile == "empty") continue;
      let screenX = (x - left) * scale;
      let screenY = (y - top) * scale;
      let tileX = tile == "lava" ? scale : 0;
      this.cx.drawImage(otherSprites,
                        tileX,         0, scale, scale,
                        screenX, screenY, scale, scale);
    }
  }
};
```

Tiles that are not empty are drawn with `drawImage`. The `otherSprites` image contains the pictures used for elements other than the player. It contains, from left to right, the wall tile, the lava tile, and the sprite for a coin.

Background tiles are 20 by 20 pixels since we will use the same scale that we used in `DOMDisplay`. Thus, the offset for lava tiles is 20 (the value of the `scale` binding), and the offset for walls is 0.

We don't bother waiting for the sprite image to load. Calling `drawImage` with an image that hasn't been loaded yet will simply do nothing. Thus, we might fail to draw the game properly for the first few frames, while the image is still loading, but that is not a serious problem. Since we keep updating the screen, the correct scene will appear as soon as the loading finishes.

The walking character shown earlier will be used to represent the player. The code that draws it needs to pick the right sprite and direction based on the player's current motion. The first eight sprites contain a walking animation. When the player is moving along a floor, we cycle through them based on the current time. We want to switch frames every 60 milliseconds, so the time is divided by 60 first. When the player is standing still, we draw the ninth sprite. During jumps, which are recognized by the fact that the vertical speed is not zero, we use the tenth, rightmost sprite.

Because the sprites are slightly wider than the player object—24 instead of 16 pixels to allow some space for feet and arms—the method has to adjust the x-coordinate and width by a given amount (`playerXOverlap`).

```
let playerSprites = document.createElement("img");
playerSprites.src = "img/player.png";
const playerXOverlap = 4;

CanvasDisplay.prototype.drawPlayer = function(player, x, y,
                                              width, height){
  width += playerXOverlap * 2;
  x -= playerXOverlap;
  if (player.speed.x != 0) {
    this.flipPlayer = player.speed.x < 0;
  }

  let tile = 8;
  if (player.speed.y != 0) {
    tile = 9;
  } else if (player.speed.x != 0) {
    tile = Math.floor(Date.now() / 60) % 8;
  }

  this.cx.save();
  if (this.flipPlayer) {
    flipHorizontally(this.cx, x + width / 2);
  }
```

```
  let tileX = tile * width;
  this.cx.drawImage(playerSprites, tileX, 0, width, height,
                                   x,     y, width, height);
  this.cx.restore();
};
```

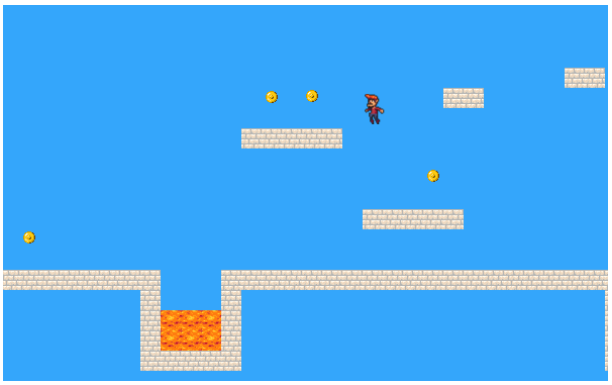
The `drawPlayer` method is called by `drawActors`, which is responsible for drawing all the actors in the game.

```
CanvasDisplay.prototype.drawActors = function(actors) {
  for (let actor of actors) {
    let width = actor.size.x * scale;
    let height = actor.size.y * scale;
    let x = (actor.pos.x - this.viewport.left) * scale;
    let y = (actor.pos.y - this.viewport.top) * scale;
    if (actor.type == "player") {
      this.drawPlayer(actor, x, y, width, height);
    } else {
      let tileX = (actor.type == "coin" ? 2 : 1) * scale;
      this.cx.drawImage(otherSprites,
                        tileX, 0, width, height,
                        x,     y, width, height);
    }
  }
};
```

When drawing something that is not the player, we look at its type to find the offset of the correct sprite. The lava tile is found at offset 20, and the coin sprite is found at 40 (two times `scale`).

We have to subtract the viewport's position when computing the actor's position since (0,0) on our canvas corresponds to the top left of the viewport, not the top left of the level. We could also have used `translate` for this. Either way works.

That concludes the new display system. The resulting game looks something like this:



Choosing a graphics interface

So when you need to generate graphics in the browser, you can choose between plain HTML, SVG, and canvas. There is no single *best* approach that works in all situations. Each option has strengths and weaknesses.

Plain HTML has the advantage of being simple. It also integrates well with text. Both SVG and canvas allow you to draw text, but they won't help you position that text or wrap it when it takes up more than one line. In an HTML-based picture, it is much easier to include blocks of text.

SVG can be used to produce crisp graphics that look good at any zoom level. Unlike HTML, it is designed for drawing and is thus more suitable for that purpose.

Both SVG and HTML build up a data structure (the DOM) that represents your picture. This makes it possible to modify elements after they are drawn. If you need to repeatedly change a small part of a big picture in response to what the user is doing or as part of an animation, doing it in a canvas can be needlessly expensive. The DOM also allows us to register mouse event handlers on every element in the picture (even on shapes drawn with SVG). You can't do that with canvas.

But canvas's pixel-oriented approach can be an advantage when drawing a huge number of tiny elements. The fact that it does not build up a data structure but only repeatedly draws onto the same pixel surface gives canvas a lower cost per shape.

There are also effects, such as rendering a scene one pixel at a time (for example, using a ray tracer) or postprocessing an image with JavaScript (blurring or distorting it), that can be realistically handled only by a pixel-based approach.

In some cases, you may want to combine several of these techniques. For

example, you might draw a graph with SVG or canvas but show textual information by positioning an HTML element on top of the picture.

For nondemanding applications, it really doesn't matter much which interface you choose. The display we built for our game in this chapter could have been implemented using any of these three graphics technologies since it does not need to draw text, handle mouse interaction, or work with an extraordinarily large number of elements.

Summary

In this chapter we discussed techniques for drawing graphics in the browser, focusing on the `<canvas>` element.

A canvas node represents an area in a document that our program may draw on. This drawing is done through a drawing context object, created with the `getContext` method.

The 2D drawing interface allows us to fill and stroke various shapes. The context's `fillStyle` property determines how shapes are filled. The `strokeStyle` and `lineWidth` properties control the way lines are drawn.

Rectangles and pieces of text can be drawn with a single method call. The `fillRect` and `strokeRect` methods draw rectangles, and the `fillText` and `strokeText` methods draw text. To create custom shapes, we must first build up a path.

Calling `beginPath` starts a new path. A number of other methods add lines and curves to the current path. For example, `lineTo` can add a straight line. When a path is finished, it can be filled with the `fill` method or stroked with the `stroke` method.

Moving pixels from an image or another canvas onto our canvas is done with the `drawImage` method. By default, this method draws the whole source image, but by giving it more parameters, you can copy a specific area of the image. We used this for our game by copying individual poses of the game character out of an image that contained many such poses.

Transformations allow you to draw a shape in multiple orientations. A 2D drawing context has a current transformation that can be changed with the `translate`, `scale`, and `rotate` methods. These will affect all subsequent drawing operations. A transformation state can be saved with the `save` method and restored with the `restore` method.

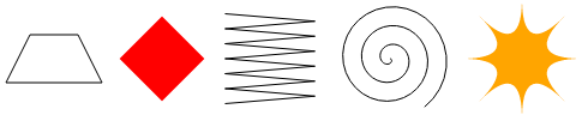
When showing an animation on a canvas, the `clearRect` method can be used to clear part of the canvas before redrawing it.

Exercises

Shapes

Write a program that draws the following shapes on a canvas:

1. A trapezoid (a rectangle that is wider on one side)
2. A red diamond (a rectangle rotated 45 degrees or ¼π radians)
3. A zigzagging line
4. A spiral made up of 100 straight line segments
5. A yellow star



When drawing the last two, you may want to refer to the explanation of `Math.cos` and `Math.sin` in Chapter 14, which describes how to get coordinates on a circle using these functions.

I recommend creating a function for each shape. Pass the position, and optionally other properties such as the size or the number of points, as parameters. The alternative, which is to hard-code numbers all over your code, tends to make the code needlessly hard to read and modify.

The pie chart

Earlier in the chapter, we saw an example program that drew a pie chart. Modify this program so that the name of each category is shown next to the slice that represents it. Try to find a pleasing-looking way to automatically position this text that would work for other data sets as well. You may assume that categories are big enough to leave ample room for their labels.

You might need `Math.sin` and `Math.cos` again, which are described in Chapter 14.

A bouncing ball

Use the `requestAnimationFrame` technique that we saw in Chapter 14 and Chapter 16 to draw a box with a bouncing ball in it. The ball moves at a constant speed and bounces off the box's sides when it hits them.

Precomputed mirroring

One unfortunate thing about transformations is that they slow down the drawing of bitmaps. The position and size of each pixel has to be transformed, and though it is possible that browsers will get cleverer about transformation in the future, they currently cause a measurable increase in the time it takes to draw a bitmap.

In a game like ours, where we are drawing only a single transformed sprite, this is a nonissue. But imagine that we need to draw hundreds of characters or thousands of rotating particles from an explosion.

Think of a way to allow us to draw an inverted character without loading additional image files and without having to make transformed `drawImage` calls every frame.

> *"Communication must be stateless in nature [...] such that each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server."*
>
> —Roy Fielding, Architectural Styles and the Design of Network-based Software Architectures

CHAPTER 18

HTTP and Forms

The *Hypertext Transfer Protocol*, already mentioned in Chapter 13, is the mechanism through which data is requested and provided on the World Wide Web. This chapter describes the protocol in more detail and explains the way browser JavaScript has access to it.

The protocol

If you type *eloquentjavascript.net/18_http.html* into your browser's address bar, the browser first looks up the address of the server associated with *eloquentjavascript.net* and tries to open a TCP connection to it on port 80, the default port for HTTP traffic. If the server exists and accepts the connection, the browser might send something like this:

```
GET /18_http.html HTTP/1.1
Host: eloquentjavascript.net
User-Agent: Your browser's name
```

Then the server responds, through that same connection.

```
HTTP/1.1 200 OK
Content-Length: 65585
Content-Type: text/html
Last-Modified: Mon, 08 Jan 2018 10:29:45 GMT

<!doctype html>
... the rest of the document
```

The browser takes the part of the response after the blank line, its *body* (not to be confused with the HTML `<body>` tag), and displays it as an HTML document.

The information sent by the client is called the *request*. It starts with this

line:

```
GET /18_http.html HTTP/1.1
```

The first word is the *method* of the request. `GET` means that we want to *get* the specified resource. Other common methods are `DELETE` to delete a resource, `PUT` to create or replace it, and `POST` to send information to it. Note that the server is not obliged to carry out every request it gets. If you walk up to a random website and tell it to `DELETE` its main page, it'll probably refuse.

The part after the method name is the path of the *resource* the request applies to. In the simplest case, a resource is simply a file on the server, but the protocol doesn't require it to be. A resource may be anything that can be transferred *as if* it is a file. Many servers generate the responses they produce on the fly. For example, if you open *https://github.com/marijnh*, the server looks in its database for a user named "marijnh", and if it finds one, it will generate a profile page for that user.

After the resource path, the first line of the request mentions `HTTP/1.1` to indicate the version of the HTTP protocol it is using.

In practice, many sites use HTTP version 2, which supports the same concepts as version 1.1 but is a lot more complicated so that it can be faster. Browsers will automatically switch to the appropriate protocol version when talking to a given server, and the outcome of a request is the same regardless of which version is used. Because version 1.1 is more straightforward and easier to play around with, we'll focus on that.

The server's response will start with a version as well, followed by the status of the response, first as a three-digit status code and then as a human-readable string.

```
HTTP/1.1 200 OK
```

Status codes starting with a 2 indicate that the request succeeded. Codes starting with 4 mean there was something wrong with the request. 404 is probably the most famous HTTP status code—it means that the resource could not be found. Codes that start with 5 mean an error happened on the server and the request is not to blame.

The first line of a request or response may be followed by any number of *headers*. These are lines in the form `name: value` that specify extra information about the request or response. These headers were part of the example response:

```
Content-Length: 65585
Content-Type: text/html
Last-Modified: Thu, 04 Jan 2018 14:05:30 GMT
```

This tells us the size and type of the response document. In this case, it is an HTML document of 65,585 bytes. It also tells us when that document was last modified.

For most headers, the client and server are free to decide whether to include them in a request or response. But a few are required. For example, the `Host` header, which specifies the hostname, should be included in a request because a server might be serving multiple hostnames on a single IP address, and without that header, the server won't know which hostname the client is trying to talk to.

After the headers, both requests and responses may include a blank line followed by a body, which contains the data being sent. `GET` and `DELETE` requests don't send along any data, but `PUT` and `POST` requests do. Similarly, some response types, such as error responses, do not require a body.

Browsers and HTTP

As we saw in the example, a browser will make a request when we enter a URL in its address bar. When the resulting HTML page references other files, such as images and JavaScript files, those are also retrieved.

A moderately complicated website can easily include anywhere from 10 to 200 resources. To be able to fetch those quickly, browsers will make several `GET` requests simultaneously, rather than waiting for the responses one at a time.

HTML pages may include *forms*, which allow the user to fill out information and send it to the server. This is an example of a form:

```
<form method="GET" action="example/message.html">
  <p>Name: <input type="text" name="name"></p>
  <p>Message:<br><textarea name="message"></textarea></p>
  <p><button type="submit">Send</button></p>
</form>
```

This code describes a form with two fields: a small one asking for a name and a larger one to write a message in. When you click the Send button, the form is *submitted*, meaning that the content of its field is packed into an HTTP request and the browser navigates to the result of that request.

When the `<form>` element's `method` attribute is `GET` (or is omitted), the information in the form is added to the end of the `action` URL as a *query string*. The browser might make a request to this URL:

```
GET /example/message.html?name=Jean&message=Yes%3F HTTP/1.1
```

The question mark indicates the end of the path part of the URL and the start of the query. It is followed by pairs of names and values, corresponding to the `name` attribute on the form field elements and the content of those elements, respectively. An ampersand character (`&`) is used to separate the pairs.

The actual message encoded in the URL is "Yes?", but the question mark is replaced by a strange code. Some characters in query strings must be escaped. The question mark, represented as `%3F`, is one of those. There seems to be an unwritten rule that every format needs its own way of escaping characters. This one, called *URL encoding*, uses a percent sign followed by two hexadecimal (base 16) digits that encode the character code. In this case, 3F, which is 63 in decimal notation, is the code of a question mark character. JavaScript provides the `encodeURIComponent` and `decodeURIComponent` functions to encode and decode this format.

```
console.log(encodeURIComponent("Yes?"));
// → Yes%3F
console.log(decodeURIComponent("Yes%3F"));
// → Yes?
```

If we change the `method` attribute of the HTML form in the example we saw earlier to `POST`, the HTTP request made to submit the form will use the `POST` method and put the query string in the body of the request, rather than adding it to the URL.

```
POST /example/message.html HTTP/1.1
Content-length: 24
Content-type: application/x-www-form-urlencoded

name=Jean&message=Yes%3F
```

`GET` requests should be used for requests that do not have side effects but simply ask for information. Requests that change something on the server, for example creating a new account or posting a message, should be expressed with other methods, such as `POST`. Client-side software such as a browser knows

that it shouldn't blindly make `POST` requests but will often implicitly make `GET` requests—for example to prefetch a resource it believes the user will soon need.

We'll come back to forms and how to interact with them from JavaScript later in the chapter.

Fetch

The interface through which browser JavaScript can make HTTP requests is called `fetch`. Since it is relatively new, it conveniently uses promises (which is rare for browser interfaces).

```
fetch("example/data.txt").then(response => {
  console.log(response.status);
  // → 200
  console.log(response.headers.get("Content-Type"));
  // → text/plain
});
```

Calling `fetch` returns a promise that resolves to a `Response` object holding information about the server's response, such as its status code and its headers. The headers are wrapped in a `Map`-like object that treats its keys (the header names) as case insensitive because header names are not supposed to be case sensitive. This means `headers.get("Content-Type")` and `headers.get("content-TYPE")` will return the same value.

Note that the promise returned by `fetch` resolves successfully even if the server responded with an error code. It *might* also be rejected if there is a network error or if the server that the request is addressed to can't be found.

The first argument to `fetch` is the URL that should be requested. When that URL doesn't start with a protocol name (such as *http:*), it is treated as *relative*, which means it is interpreted relative to the current document. When it starts with a slash (/), it replaces the current path, which is the part after the server name. When it does not, the part of the current path up to and including its last slash character is put in front of the relative URL.

To get at the actual content of a response, you can use its `text` method. Because the initial promise is resolved as soon as the response's headers have been received and because reading the response body might take a while longer, this again returns a promise.

```
fetch("example/data.txt")
  .then(resp => resp.text())
  .then(text => console.log(text));
```

```
// → This is the content of data.txt
```

A similar method, called `json`, returns a promise that resolves to the value you get when parsing the body as JSON or rejects if it's not valid JSON.

By default, `fetch` uses the `GET` method to make its request and does not include a request body. You can configure it differently by passing an object with extra options as a second argument. For example, this request tries to delete `example/data.txt`:

```
fetch("example/data.txt", {method: "DELETE"}).then(resp => {
  console.log(resp.status);
  // → 405
});
```

The 405 status code means "method not allowed", an HTTP server's way of saying "I can't do that".

To add a request body, you can include a `body` option. To set headers, there's the `headers` option. For example, this request includes a `Range` header, which instructs the server to return only part of a response.

```
fetch("example/data.txt", {headers: {Range: "bytes=8-19"}})
  .then(resp => resp.text())
  .then(console.log);
// → the content
```

The browser will automatically add some request headers, such as "Host" and those needed for the server to figure out the size of the body. But adding your own headers is often useful to include things such as authentication information or to tell the server which file format you'd like to receive.

HTTP SANDBOXING

Making HTTP requests in web page scripts once again raises concerns about security. The person who controls the script might not have the same interests as the person on whose computer it is running. More specifically, if I visit *themafia.org*, I do not want its scripts to be able to make a request to *mybank.com*, using identifying information from my browser, with instructions to transfer all my money to some random account.

For this reason, browsers protect us by disallowing scripts to make HTTP

requests to other domains (names such as *themafia.org* and *mybank.com*).

This can be an annoying problem when building systems that want to access several domains for legitimate reasons. Fortunately, servers can include a header like this in their response to explicitly indicate to the browser that it is okay for the request to come from another domain:

```
Access-Control-Allow-Origin: *
```

Appreciating HTTP

When building a system that requires communication between a JavaScript program running in the browser (client-side) and a program on a server (server-side), there are several different ways to model this communication.

A commonly used model is that of *remote procedure calls*. In this model, communication follows the patterns of normal function calls, except that the function is actually running on another machine. Calling it involves making a request to the server that includes the function's name and arguments. The response to that request contains the returned value.

When thinking in terms of remote procedure calls, HTTP is just a vehicle for communication, and you will most likely write an abstraction layer that hides it entirely.

Another approach is to build your communication around the concept of resources and HTTP methods. Instead of a remote procedure called `addUser`, you use a `PUT` request to `/users/larry`. Instead of encoding that user's properties in function arguments, you define a JSON document format (or use an existing format) that represents a user. The body of the `PUT` request to create a new resource is then such a document. A resource is fetched by making a `GET` request to the resource's URL (for example, `/user/larry`), which again returns the document representing the resource.

This second approach makes it easier to use some of the features that HTTP provides, such as support for caching resources (keeping a copy on the client for fast access). The concepts used in HTTP, which are well designed, can provide a helpful set of principles to design your server interface around.

Security and HTTPS

Data traveling over the Internet tends to follow a long, dangerous road. To get to its destination, it must hop through anything from coffee shop Wi-Fi

hotspots to networks controlled by various companies and states. At any point along its route it may be inspected or even modified.

If it is important that something remain secret, such as the password to your email account, or that it arrive at its destination unmodified, such as the account number you transfer money to via your bank's website, plain HTTP is not good enough.

The secure HTTP protocol, used for URLs starting with *https://*, wraps HTTP traffic in a way that makes it harder to read and tamper with. Before exchanging data, the client verifies that the server is who it claims to be by asking it to prove that it has a cryptographic certificate issued by a certificate authority that the browser recognizes. Next, all data going over the connection is encrypted in a way that should prevent eavesdropping and tampering.

Thus, when it works right, HTTPS prevents other people from impersonating the website you are trying to talk to and from snooping on your communication. It is not perfect, and there have been various incidents where HTTPS failed because of forged or stolen certificates and broken software, but it is a *lot* safer than plain HTTP.

Form fields

Forms were originally designed for the pre-JavaScript Web to allow web sites to send user-submitted information in an HTTP request. This design assumes that interaction with the server always happens by navigating to a new page.

But their elements are part of the DOM like the rest of the page, and the DOM elements that represent form fields support a number of properties and events that are not present on other elements. These make it possible to inspect and control such input fields with JavaScript programs and do things such as adding new functionality to a form or using forms and fields as building blocks in a JavaScript application.

A web form consists of any number of input fields grouped in a `<form>` tag. HTML allows several different styles of fields, ranging from simple on/off checkboxes to drop-down menus and fields for text input. This book won't try to comprehensively discuss all field types, but we'll start with a rough overview.

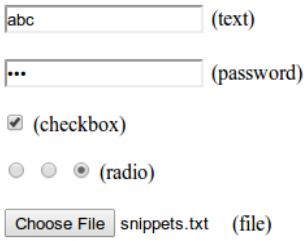
A lot of field types use the `<input>` tag. This tag's `type` attribute is used to select the field's style. These are some commonly used `<input>` types:

`text`	A single-line text field
`password`	Same as `text` but hides the text that is typed
`checkbox`	An on/off switch
`radio`	(Part of) a multiple-choice field
`file`	Allows the user to choose a file from their computer

Form fields do not necessarily have to appear in a `<form>` tag. You can put them anywhere in a page. Such form-less fields cannot be submitted (only a form as a whole can), but when responding to input with JavaScript, we often don't want to submit our fields normally anyway.

```
<p><input type="text" value="abc"> (text)</p>
<p><input type="password" value="abc"> (password)</p>
<p><input type="checkbox" checked> (checkbox)</p>
<p><input type="radio" value="A" name="choice">
   <input type="radio" value="B" name="choice" checked>
   <input type="radio" value="C" name="choice"> (radio)</p>
<p><input type="file"> (file)</p>
```

The fields created with this HTML code look like this:



The JavaScript interface for such elements differs with the type of the element.

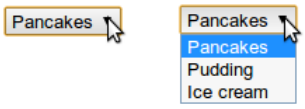
Multiline text fields have their own tag, `<textarea>`, mostly because using an attribute to specify a multiline starting value would be awkward. The `<textarea>` tag requires a matching `</textarea>` closing tag and uses the text between those two, instead of the `value` attribute, as starting text.

```
<textarea>
one
two
three
</textarea>
```

Finally, the `<select>` tag is used to create a field that allows the user to select from a number of predefined options.

```
<select>
  <option>Pancakes</option>
  <option>Pudding</option>
  <option>Ice cream</option>
</select>
```

Such a field looks like this:



Whenever the value of a form field changes, it will fire a `"change"` event.

Focus

Unlike most elements in HTML documents, form fields can get *keyboard focus*. When clicked or activated in some other way, they become the currently active element and the recipient of keyboard input.

Thus, you can type into a text field only when it is focused. Other fields respond differently to keyboard events. For example, a `<select>` menu tries to move to the option that contains the text the user typed and responds to the arrow keys by moving its selection up and down.

We can control focus from JavaScript with the `focus` and `blur` methods. The first moves focus to the DOM element it is called on, and the second removes focus. The value in `document.activeElement` corresponds to the currently focused element.

```
<input type="text">
<script>
  document.querySelector("input").focus();
  console.log(document.activeElement.tagName);
  // → INPUT
  document.querySelector("input").blur();
  console.log(document.activeElement.tagName);
  // → BODY
</script>
```

For some pages, the user is expected to want to interact with a form field immediately. JavaScript can be used to focus this field when the document is loaded, but HTML also provides the `autofocus` attribute, which produces the same effect while letting the browser know what we are trying to achieve. This

gives the browser the option to disable the behavior when it is not appropriate, such as when the user has put the focus on something else.

Browsers traditionally also allow the user to move the focus through the document by pressing the TAB key. We can influence the order in which elements receive focus with the `tabindex` attribute. The following example document will let the focus jump from the text input to the OK button, rather than going through the help link first:

```
<input type="text" tabindex=1> <a href=".">(help)</a>
<button onclick="console.log('ok')" tabindex=2>OK</button>
```

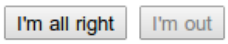
By default, most types of HTML elements cannot be focused. But you can add a `tabindex` attribute to any element that will make it focusable. A `tabindex` of -1 makes tabbing skip over an element, even if it is normally focusable.

Disabled fields

All form fields can be *disabled* through their `disabled` attribute. It is an attribute that can be specified without value—the fact that it is present at all disables the element.

```
<button>I'm all right</button>
<button disabled>I'm out</button>
```

Disabled fields cannot be focused or changed, and browsers make them look gray and faded.



When a program is in the process of handling an action caused by some button or other control that might require communication with the server and thus take a while, it can be a good idea to disable the control until the action finishes. That way, when the user gets impatient and clicks it again, they don't accidentally repeat their action.

The form as a whole

When a field is contained in a `<form>` element, its DOM element will have a `form` property linking back to the form's DOM element. The `<form>` element,

in turn, has a property called `elements` that contains an array-like collection of the fields inside it.

The `name` attribute of a form field determines the way its value will be identified when the form is submitted. It can also be used as a property name when accessing the form's `elements` property, which acts both as an array-like object (accessible by number) and a map (accessible by name).

```
<form action="example/submit.html">
  Name: <input type="text" name="name"><br>
  Password: <input type="password" name="password"><br>
  <button type="submit">Log in</button>
</form>
<script>
  let form = document.querySelector("form");
  console.log(form.elements[1].type);
  // → password
  console.log(form.elements.password.type);
  // → password
  console.log(form.elements.name.form == form);
  // → true
</script>
```

A button with a `type` attribute of `submit` will, when pressed, cause the form to be submitted. Pressing ENTER when a form field is focused has the same effect.

Submitting a form normally means that the browser navigates to the page indicated by the form's `action` attribute, using either a `GET` or a `POST` request. But before that happens, a `"submit"` event is fired. You can handle this event with JavaScript and prevent this default behavior by calling `preventDefault` on the event object.

```
<form action="example/submit.html">
  Value: <input type="text" name="value">
  <button type="submit">Save</button>
</form>
<script>
  let form = document.querySelector("form");
  form.addEventListener("submit", event => {
    console.log("Saving value", form.elements.value.value);
    event.preventDefault();
  });
</script>
```

Intercepting `"submit"` events in JavaScript has various uses. We can write code to verify that the values the user entered make sense and immediately show an error message instead of submitting the form. Or we can disable the regular way of submitting the form entirely, as in the example, and have our program handle the input, possibly using `fetch` to send it to a server without reloading the page.

Text fields

Fields created by `<textarea>` tags, or `<input>` tags with a type of `text` or `password`, share a common interface. Their DOM elements have a `value` property that holds their current content as a string value. Setting this property to another string changes the field's content.

The `selectionStart` and `selectionEnd` properties of text fields give us information about the cursor and selection in the text. When nothing is selected, these two properties hold the same number, indicating the position of the cursor. For example, 0 indicates the start of the text, and 10 indicates the cursor is after the 10^{th} character. When part of the field is selected, the two properties will differ, giving us the start and end of the selected text. Like `value`, these properties may also be written to.

Imagine you are writing an article about Khasekhemwy but have some trouble spelling his name. The following code wires up a `<textarea>` tag with an event handler that, when you press F2, inserts the string "Khasekhemwy" for you.

```
<textarea></textarea>
<script>
  let textarea = document.querySelector("textarea");
  textarea.addEventListener("keydown", event => {
    // The key code for F2 happens to be 113
    if (event.keyCode == 113) {
      replaceSelection(textarea, "Khasekhemwy");
      event.preventDefault();
    }
  });
  function replaceSelection(field, word) {
    let from = field.selectionStart, to = field.selectionEnd;
    field.value = field.value.slice(0, from) + word +
                  field.value.slice(to);
    // Put the cursor after the word
    field.selectionStart = from + word.length;
    field.selectionEnd = from + word.length;
```

```
  }
</script>
```

The `replaceSelection` function replaces the currently selected part of a text field's content with the given word and then moves the cursor after that word so that the user can continue typing.

The `"change"` event for a text field does not fire every time something is typed. Rather, it fires when the field loses focus after its content was changed. To respond immediately to changes in a text field, you should register a handler for the `"input"` event instead, which fires for every time the user types a character, deletes text, or otherwise manipulates the field's content.

The following example shows a text field and a counter displaying the current length of the text in the field:

```
<input type="text"> length: <span id="length">0</span>
<script>
  let text = document.querySelector("input");
  let output = document.querySelector("#length");
  text.addEventListener("input", () => {
    output.textContent = text.value.length;
  });
</script>
```

Checkboxes and radio buttons

A checkbox field is a binary toggle. Its value can be extracted or changed through its `checked` property, which holds a Boolean value.

```
<label>
  <input type="checkbox" id="purple"> Make this page purple
</label>
<script>
  let checkbox = document.querySelector("#purple");
  checkbox.addEventListener("change", () => {
    document.body.style.background =
      checkbox.checked ? "mediumpurple" : "";
  });
</script>
```

The `<label>` tag associates a piece of document with an input field. Clicking

anywhere on the label will activate the field, which focuses it and toggles its value when it is a checkbox or radio button.

A radio button is similar to a checkbox, but it's implicitly linked to other radio buttons with the same `name` attribute so that only one of them can be active at any time.

```
Color:
<label>
  <input type="radio" name="color" value="orange"> Orange
</label>
<label>
  <input type="radio" name="color" value="lightgreen"> Green
</label>
<label>
  <input type="radio" name="color" value="lightblue"> Blue
</label>
<script>
  let buttons = document.querySelectorAll("[name=color]");
  for (let button of Array.from(buttons)) {
    button.addEventListener("change", () => {
      document.body.style.background = button.value;
    });
  }
</script>
```

The square brackets in the CSS query given to `querySelectorAll` are used to match attributes. It selects elements whose `name` attribute is `"color"`.

Select fields

Select fields are conceptually similar to radio buttons—they also allow the user to choose from a set of options. But where a radio button puts the layout of the options under our control, the appearance of a `<select>` tag is determined by the browser.

Select fields also have a variant that is more akin to a list of checkboxes, rather than radio boxes. When given the `multiple` attribute, a `<select>` tag will allow the user to select any number of options, rather than just a single option. This will, in most browsers, show up differently than a normal select field, which is typically drawn as a *drop-down* control that shows the options only when you open it.

Each `<option>` tag has a value. This value can be defined with a `value` attribute. When that is not given, the text inside the option will count as its

value. The `value` property of a `<select>` element reflects the currently selected option. For a `multiple` field, though, this property doesn't mean much since it will give the value of only *one* of the currently selected options.

The `<option>` tags for a `<select>` field can be accessed as an array-like object through the field's `options` property. Each option has a property called `selected`, which indicates whether that option is currently selected. The property can also be written to select or deselect an option.

This example extracts the selected values from a `multiple` select field and uses them to compose a binary number from individual bits. Hold CONTROL (or COMMAND on a Mac) to select multiple options.

```
<select multiple>
  <option value="1">0001</option>
  <option value="2">0010</option>
  <option value="4">0100</option>
  <option value="8">1000</option>
</select> = <span id="output">0</span>
<script>
  let select = document.querySelector("select");
  let output = document.querySelector("#output");
  select.addEventListener("change", () => {
    let number = 0;
    for (let option of Array.from(select.options)) {
      if (option.selected) {
        number += Number(option.value);
      }
    }
    output.textContent = number;
  });
</script>
```

FILE FIELDS

File fields were originally designed as a way to upload files from the user's machine through a form. In modern browsers, they also provide a way to read such files from JavaScript programs. The field acts as a kind of gatekeeper. The script cannot simply start reading private files from the user's computer, but if the user selects a file in such a field, the browser interprets that action to mean that the script may read the file.

A file field usually looks like a button labeled with something like "choose file" or "browse", with information about the chosen file next to it.

```
<input type="file">
<script>
  let input = document.querySelector("input");
  input.addEventListener("change", () => {
    if (input.files.length > 0) {
      let file = input.files[0];
      console.log("You chose", file.name);
      if (file.type) console.log("It has type", file.type);
    }
  });
</script>
```

The `files` property of a file field element is an array-like object (again, not a real array) containing the files chosen in the field. It is initially empty. The reason there isn't simply a `file` property is that file fields also support a `multiple` attribute, which makes it possible to select multiple files at the same time.

Objects in the `files` object have properties such as `name` (the filename), `size` (the file's size in bytes, which are chunks of 8 bits), and `type` (the media type of the file, such as `text/plain` or `image/jpeg`).

What it does not have is a property that contains the content of the file. Getting at that is a little more involved. Since reading a file from disk can take time, the interface must be asynchronous to avoid freezing the document.

```
<input type="file" multiple>
<script>
  let input = document.querySelector("input");
  input.addEventListener("change", () => {
    for (let file of Array.from(input.files)) {
      let reader = new FileReader();
      reader.addEventListener("load", () => {
        console.log("File", file.name, "starts with",
                    reader.result.slice(0, 20));
      });
      reader.readAsText(file);
    }
  });
</script>
```

Reading a file is done by creating a `FileReader` object, registering a `"load"` event handler for it, and calling its `readAsText` method, giving it the file we want to read. Once loading finishes, the reader's `result` property contains the

file's content.

`FileReader`s also fire an `"error"` event when reading the file fails for any reason. The error object itself will end up in the reader's `error` property. This interface was designed before promises became part of the language. You could wrap it in a promise like this:

```
function readFileText(file) {
  return new Promise((resolve, reject) => {
    let reader = new FileReader();
    reader.addEventListener(
      "load", () => resolve(reader.result));
    reader.addEventListener(
      "error", () => reject(reader.error));
    reader.readAsText(file);
  });
}
```

Storing Data Client-Side

Simple HTML pages with a bit of JavaScript can be a great format for "mini applications"—small helper programs that automate basic tasks. By connecting a few form fields with event handlers, you can do anything from converting between centimeters and inches to computing passwords from a master password and a website name.

When such an application needs to remember something between sessions, you cannot use JavaScript bindings—those are thrown away every time the page is closed. You could set up a server, connect it to the Internet, and have your application store something there. We will see how to do that in Chapter 20. But that's a lot of extra work and complexity. Sometimes it is enough to just keep the data in the browser.

The `localStorage` object can be used to store data in a way that survives page reloads. This object allows you to file string values under names.

```
localStorage.setItem("username", "marijn");
console.log(localStorage.getItem("username"));
// → marijn
localStorage.removeItem("username");
```

A value in `localStorage` sticks around until it is overwritten, it is removed with `removeItem`, or the user clears their local data.

Sites from different domains get different storage compartments. That means data stored in `localStorage` by a given website can, in principle, be read (and overwritten) only by scripts on that same site.

Browsers do enforce a limit on the size of the data a site can store in `localStorage`. That restriction, along with the fact that filling up people's hard drives with junk is not really profitable, prevents the feature from eating up too much space.

The following code implements a crude note-taking application. It keeps a set of named notes and allows the user to edit notes and create new ones.

```
Notes: <select></select> <button>Add</button><br>
<textarea style="width: 100%"></textarea>

<script>
  let list = document.querySelector("select");
  let note = document.querySelector("textarea");

  let state;
  function setState(newState) {
    list.textContent = "";
    for (let name of Object.keys(newState.notes)) {
      let option = document.createElement("option");
      option.textContent = name;
      if (newState.selected == name) option.selected = true;
      list.appendChild(option);
    }
    note.value = newState.notes[newState.selected];

    localStorage.setItem("Notes", JSON.stringify(newState));
    state = newState;
  }
  setState(JSON.parse(localStorage.getItem("Notes")) || {
    notes: {"shopping list": "Carrots\nRaisins"},
    selected: "shopping list"
  });

  list.addEventListener("change", () => {
    setState({notes: state.notes, selected: list.value});
  });
  note.addEventListener("change", () => {
    setState({
      notes: Object.assign({}, state.notes,
                           {[state.selected]: note.value}),
      selected: state.selected
    });
```

```
  });
  document.querySelector("button")
    .addEventListener("click", () => {
      let name = prompt("Note name");
      if (name) setState({
        notes: Object.assign({}, state.notes, {[name]: ""}),
        selected: name
      });
    });
</script>
```

The script gets its starting state from the `"Notes"` value stored in `localStorage` or, if that is missing, creates an example state that has only a shopping list in it. Reading a field that does not exist from `localStorage` will yield `null`. Passing `null` to `JSON.parse` will make it parse the string `"null"` and return `null`. Thus, the `||` operator can be used to provide a default value in a situation like this.

The `setState` method makes sure the DOM is showing a given state and stores the new state to `localStorage`. Event handlers call this function to move to a new state.

The use of `Object.assign` in the example is intended to create a new object that is a clone of the old `state.notes`, but with one property added or overwritten. `Object.assign` takes its first argument and adds all properties from any further arguments to it. Thus, giving it an empty object will cause it to fill a fresh object. The square brackets notation in the third argument is used to create a property whose name is based on some dynamic value.

There is another object, similar to `localStorage`, called `sessionStorage`. The difference between the two is that the content of `sessionStorage` is forgotten at the end of each *session*, which for most browsers means whenever the browser is closed.

Summary

In this chapter, we discussed how the HTTP protocol works. A *client* sends a request, which contains a method (usually `GET`) and a path that identifies a resource. The *server* then decides what to do with the request and responds with a status code and a response body. Both requests and responses may contain headers that provide additional information.

The interface through which browser JavaScript can make HTTP requests is called `fetch`. Making a request looks like this:

```
fetch("/18_http.html").then(r => r.text()).then(text => {
  console.log(`The page starts with ${text.slice(0, 15)}`);
});
```

Browsers make `GET` requests to fetch the resources needed to display a web page. A page may also contain forms, which allow information entered by the user to be sent as a request for a new page when the form is submitted.

HTML can represent various types of form fields, such as text fields, checkboxes, multiple-choice fields, and file pickers.

Such fields can be inspected and manipulated with JavaScript. They fire the `"change"` event when changed, fire the `"input"` event when text is typed, and receive keyboard events when they have keyboard focus. Properties like `value` (for text and select fields) or `checked` (for checkboxes and radio buttons) are used to read or set the field's content.

When a form is submitted, a `"submit"` event is fired on it. A JavaScript handler can call `preventDefault` on that event to disable the browser's default behavior. Form field elements may also occur outside of a form tag.

When the user has selected a file from their local file system in a file picker field, the `FileReader` interface can be used to access the content of this file from a JavaScript program.

The `localStorage` and `sessionStorage` objects can be used to save information in a way that survives page reloads. The first object saves the data forever (or until the user decides to clear it), and the second saves it until the browser is closed.

Exercises

Content negotiation

One of the things HTTP can do is called *content negotiation*. The `Accept` request header is used to tell the server what type of document the client would like to get. Many servers ignore this header, but when a server knows of various ways to encode a resource, it can look at this header and send the one that the client prefers.

The URL *https://eloquentjavascript.net/author* is configured to respond with either plaintext, HTML, or JSON, depending on what the client asks for. These formats are identified by the standardized *media types* `text/plain`, `text/html`, and `application/json`.

Send requests to fetch all three formats of this resource. Use the `headers`

property in the options object passed to `fetch` to set the header named `Accept` to the desired media type.

Finally, try asking for the media type `application/rainbows+unicorns` and see which status code that produces.

A JavaScript workbench

Build an interface that allows people to type and run pieces of JavaScript code.

Put a button next to a `<textarea>` field that, when pressed, uses the `Function` constructor we saw in Chapter 10 to wrap the text in a function and call it. Convert the return value of the function, or any error it raises, to a string and display it below the text field.

Conway's Game of Life

Conway's Game of Life is a simple simulation that creates artificial "life" on a grid, each cell of which is either alive or not. Each generation (turn), the following rules are applied:

- Any live cell with fewer than two or more than three live neighbors dies.
- Any live cell with two or three live neighbors lives on to the next generation.
- Any dead cell with exactly three live neighbors becomes a live cell.

A *neighbor* is defined as any adjacent cell, including diagonally adjacent ones.

Note that these rules are applied to the whole grid at once, not one square at a time. That means the counting of neighbors is based on the situation at the start of the generation, and changes happening to neighbor cells during this generation should not influence the new state of a given cell.

Implement this game using whichever data structure you find appropriate. Use `Math.random` to populate the grid with a random pattern initially. Display it as a grid of checkbox fields, with a button next to it to advance to the next generation. When the user checks or unchecks the checkboxes, their changes should be included when computing the next generation.

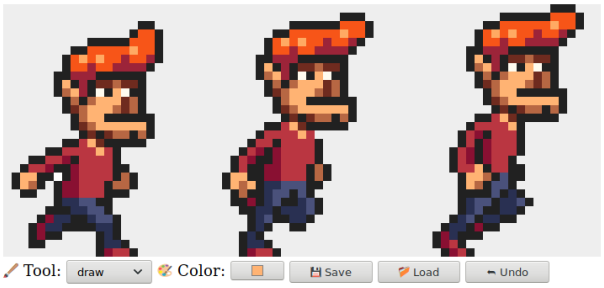
> *"I look at the many colors before me. I look at my blank canvas. Then, I try to apply colors like words that shape poems, like notes that shape music."*
>
> —Joan Miro

CHAPTER 19

PROJECT: A PIXEL ART EDITOR

The material from the previous chapters gives you all the elements you need to build a basic web application. In this chapter, we will do just that.

Our application will be a pixel drawing program, where you can modify a picture pixel by pixel by manipulating a zoomed-in view of it, shown as a grid of colored squares. You can use the program to open image files, scribble on them with your mouse or other pointer device, and save them. This is what it will look like:



Painting on a computer is great. You don't need to worry about materials, skill, or talent. You just start smearing.

COMPONENTS

The interface for the application shows a big `<canvas>` element on top, with a number of form fields below it. The user draws on the picture by selecting a tool from a `<select>` field and then clicking, touching, or dragging across the canvas. There are tools for drawing single pixels or rectangles, for filling an area, and for picking a color from the picture.

We will structure the editor interface as a number of *components*, objects that are responsible for a piece of the DOM and that may contain other components inside them.

The state of the application consists of the current picture, the selected tool, and the selected color. We'll set things up so that the state lives in a single

value, and the interface components always base the way they look on the current state.

To see why this is important, let's consider the alternative—distributing pieces of state throughout the interface. Up to a certain point, this is easier to program. We can just put in a color field and read its value when we need to know the current color.

But then we add the color picker—a tool that lets you click the picture to select the color of a given pixel. To keep the color field showing the correct color, that tool would have to know that it exists and update it whenever it picks a new color. If you ever add another place that makes the color visible (maybe the mouse cursor could show it), you have to update your color-changing code to keep that synchronized.

In effect, this creates a problem where each part of the interface needs to know about all other parts, which is not very modular. For small applications like the one in this chapter, that may not be a problem. For bigger projects, it can turn into a real nightmare.

To avoid this nightmare on principle, we're going to be strict about *data flow*. There is a state, and the interface is drawn based on that state. An interface component may respond to user actions by updating the state, at which point the components get a chance to synchronize themselves with this new state.

In practice, each component is set up so that when it is given a new state, it also notifies its child components, insofar as those need to be updated. Setting this up is a bit of a hassle. Making this more convenient is the main selling point of many browser programming libraries. But for a small application like this, we can do it without such infrastructure.

Updates to the state are represented as objects, which we'll call *actions*. Components may create such actions and *dispatch* them—give them to a central state management function. That function computes the next state, after which the interface components update themselves to this new state.

We're taking the messy task of running a user interface and applying some structure to it. Though the DOM-related pieces are still full of side effects, they are held up by a conceptually simple backbone: the state update cycle. The state determines what the DOM looks like, and the only way DOM events can change the state is by dispatching actions to the state.

There are *many* variants of this approach, each with its own benefits and problems, but their central idea is the same: state changes should go through a single well-defined channel, not happen all over the place.

Our components will be classes conforming to an interface. Their constructor is given a state—which may be the whole application state or some smaller value if it doesn't need access to everything—and uses that to build up a `dom`

property. This is the DOM element that represents the component. Most constructors will also take some other values that won't change over time, such as the function they can use to dispatch an action.

Each component has a `syncState` method that is used to synchronize it to a new state value. The method takes one argument, the state, which is of the same type as the first argument to its constructor.

The state

The application state will be an object with `picture`, `tool`, and `color` properties. The picture is itself an object that stores the width, height, and pixel content of the picture. The pixels are stored in an array, in the same way as the matrix class from Chapter 6—row by row, from top to bottom.

```
class Picture {
  constructor(width, height, pixels) {
    this.width = width;
    this.height = height;
    this.pixels = pixels;
  }
  static empty(width, height, color) {
    let pixels = new Array(width * height).fill(color);
    return new Picture(width, height, pixels);
  }
  pixel(x, y) {
    return this.pixels[x + y * this.width];
  }
  draw(pixels) {
    let copy = this.pixels.slice();
    for (let {x, y, color} of pixels) {
      copy[x + y * this.width] = color;
    }
    return new Picture(this.width, this.height, copy);
  }
}
```

We want to be able to treat a picture as an immutable value, for reasons that we'll get back to later in the chapter. But we also sometimes need to update a whole bunch of pixels at a time. To be able to do that, the class has a `draw` method that expects an array of updated pixels—objects with `x`, `y`, and `color` properties—and creates a new picture with those pixels overwritten. This method uses `slice` without arguments to copy the entire pixel array—the

start of the slice defaults to 0, and the end defaults to the array's length.

The `empty` method uses two pieces of array functionality that we haven't seen before. The `Array` constructor can be called with a number to create an empty array of the given length. The `fill` method can then be used to fill this array with a given value. These are used to create an array in which all pixels have the same color.

Colors are stored as strings containing traditional CSS color codes made up of a hash sign (`#`) followed by six hexadecimal (base-16) digits—two for the red component, two for the green component, and two for the blue component. This is a somewhat cryptic and inconvenient way to write colors, but it is the format the HTML color input field uses, and it can be used in the `fillStyle` property of a canvas drawing context, so for the ways we'll use colors in this program, it is practical enough.

Black, where all components are zero, is written `"#000000"`, and bright pink looks like `"#ff00ff"`, where the red and blue components have the maximum value of 255, written `ff` in hexadecimal digits (which use a to f to represent digits 10 to 15).

We'll allow the interface to dispatch actions as objects whose properties overwrite the properties of the previous state. The color field, when the user changes it, could dispatch an object like `{color: field.value}`, from which this update function can compute a new state.

```
function updateState(state, action) {
  return Object.assign({}, state, action);
}
```

This rather cumbersome pattern, in which `Object.assign` is used to first add the properties of `state` to an empty object and then overwrite some of those with the properties from `action`, is common in JavaScript code that uses immutable objects. A more convenient notation for this, in which the triple-dot operator is used to include all properties from another object in an object expression, is in the final stages of being standardized. With that addition, you could write `{...state, ...action}` instead. At the time of writing, this doesn't yet work in all browsers.

DOM BUILDING

One of the main things that interface components do is creating DOM structure. We again don't want to directly use the verbose DOM methods for that, so

here's a slightly expanded version of the `elt` function:

```
function elt(type, props, ...children) {
  let dom = document.createElement(type);
  if (props) Object.assign(dom, props);
  for (let child of children) {
    if (typeof child != "string") dom.appendChild(child);
    else dom.appendChild(document.createTextNode(child));
  }
  return dom;
}
```

The main difference between this version and the one we used in Chapter 16 is that it assigns *properties* to DOM nodes, not *attributes*. This means we can't use it to set arbitrary attributes, but we *can* use it to set properties whose value isn't a string, such as `onclick`, which can be set to a function to register a click event handler.

This allows the following style of registering event handlers:

```
<body>
  <script>
    document.body.appendChild(elt("button", {
      onclick: () => console.log("click")
    }, "The button"));
  </script>
</body>
```

The canvas

The first component we'll define is the part of the interface that displays the picture as a grid of colored boxes. This component is responsible for two things: showing a picture and communicating pointer events on that picture to the rest of the application.

As such, we can define it as a component that knows about only the current picture, not the whole application state. Because it doesn't know how the application as a whole works, it cannot directly dispatch actions. Rather, when responding to pointer events, it calls a callback function provided by the code that created it, which will handle the application-specific parts.

```
const scale = 10;
```

```
class PictureCanvas {
  constructor(picture, pointerDown) {
    this.dom = elt("canvas", {
      onmousedown: event => this.mouse(event, pointerDown),
      ontouchstart: event => this.touch(event, pointerDown)
    });
    this.syncState(picture);
  }
  syncState(picture) {
    if (this.picture == picture) return;
    this.picture = picture;
    drawPicture(this.picture, this.dom, scale);
  }
}
```

We draw each pixel as a 10-by-10 square, as determined by the `scale` constant. To avoid unnecessary work, the component keeps track of its current picture and does a redraw only when `syncState` is given a new picture.

The actual drawing function sets the size of the canvas based on the scale and picture size and fills it with a series of squares, one for each pixel.

```
function drawPicture(picture, canvas, scale) {
  canvas.width = picture.width * scale;
  canvas.height = picture.height * scale;
  let cx = canvas.getContext("2d");

  for (let y = 0; y < picture.height; y++) {
    for (let x = 0; x < picture.width; x++) {
      cx.fillStyle = picture.pixel(x, y);
      cx.fillRect(x * scale, y * scale, scale, scale);
    }
  }
}
```

When the left mouse button is pressed while the mouse is over the picture canvas, the component calls the `pointerDown` callback, giving it the position of the pixel that was clicked—in picture coordinates. This will be used to implement mouse interaction with the picture. The callback may return another callback function to be notified when the pointer is moved to a different pixel while the button is held down.

```
PictureCanvas.prototype.mouse = function(downEvent, onDown) {
  if (downEvent.button != 0) return;
```

```
  let pos = pointerPosition(downEvent, this.dom);
  let onMove = onDown(pos);
  if (!onMove) return;
  let move = moveEvent => {
    if (moveEvent.buttons == 0) {
      this.dom.removeEventListener("mousemove", move);
    } else {
      let newPos = pointerPosition(moveEvent, this.dom);
      if (newPos.x == pos.x && newPos.y == pos.y) return;
      pos = newPos;
      onMove(newPos);
    }
  };
  this.dom.addEventListener("mousemove", move);
};

function pointerPosition(pos, domNode) {
  let rect = domNode.getBoundingClientRect();
  return {x: Math.floor((pos.clientX - rect.left) / scale),
          y: Math.floor((pos.clientY - rect.top) / scale)};
}
```

Since we know the size of the pixels and we can use `getBoundingClientRect` to find the position of the canvas on the screen, it is possible to go from mouse event coordinates (`clientX` and `clientY`) to picture coordinates. These are always rounded down so that they refer to a specific pixel.

With touch events, we have to do something similar, but using different events and making sure we call `preventDefault` on the `"touchstart"` event to prevent panning.

```
PictureCanvas.prototype.touch = function(startEvent,
                                         onDown) {
  let pos = pointerPosition(startEvent.touches[0], this.dom);
  let onMove = onDown(pos);
  startEvent.preventDefault();
  if (!onMove) return;
  let move = moveEvent => {
    let newPos = pointerPosition(moveEvent.touches[0],
                                 this.dom);
    if (newPos.x == pos.x && newPos.y == pos.y) return;
    pos = newPos;
    onMove(newPos);
  };
  let end = () => {
```

```
      this.dom.removeEventListener("touchmove", move);
      this.dom.removeEventListener("touchend", end);
    };
    this.dom.addEventListener("touchmove", move);
    this.dom.addEventListener("touchend", end);
  };
```

For touch events, `clientX` and `clientY` aren't available directly on the event object, but we can use the coordinates of the first touch object in the `touches` property.

The application

To make it possible to build the application piece by piece, we'll implement the main component as a shell around a picture canvas and a dynamic set of tools and controls that we pass to its constructor.

The *controls* are the interface elements that appear below the picture. They'll be provided as an array of component constructors.

The *tools* do things like drawing pixels or filling in an area. The application shows the set of available tools as a `<select>` field. The currently selected tool determines what happens when the user interacts with the picture with a pointer device. The set of available tools is provided as an object that maps the names that appear in the drop-down field to functions that implement the tools. Such functions get a picture position, a current application state, and a `dispatch` function as arguments. They may return a move handler function that gets called with a new position and a current state when the pointer moves to a different pixel.

```
class PixelEditor {
  constructor(state, config) {
    let {tools, controls, dispatch} = config;
    this.state = state;

    this.canvas = new PictureCanvas(state.picture, pos => {
      let tool = tools[this.state.tool];
      let onMove = tool(pos, this.state, dispatch);
      if (onMove) return pos => onMove(pos, this.state);
    });
    this.controls = controls.map(
      Control => new Control(state, config));
    this.dom = elt("div", {}, this.canvas.dom, elt("br"),
                   ...this.controls.reduce(
```

```
                    (a, c) => a.concat(" ", c.dom), []));
  }
  syncState(state) {
    this.state = state;
    this.canvas.syncState(state.picture);
    for (let ctrl of this.controls) ctrl.syncState(state);
  }
}
```

The pointer handler given to `PictureCanvas` calls the currently selected tool with the appropriate arguments and, if that returns a move handler, adapts it to also receive the state.

All controls are constructed and stored in `this.controls` so that they can be updated when the application state changes. The call to `reduce` introduces spaces between the controls' DOM elements. That way they don't look so pressed together.

The first control is the tool selection menu. It creates a `<select>` element with an option for each tool and sets up a `"change"` event handler that updates the application state when the user selects a different tool.

```
class ToolSelect {
  constructor(state, {tools, dispatch}) {
    this.select = elt("select", {
      onchange: () => dispatch({tool: this.select.value})
    }, ...Object.keys(tools).map(name => elt("option", {
      selected: name == state.tool
    }, name)));
    this.dom = elt("label", null, "🖌 Tool: ", this.select);
  }
  syncState(state) { this.select.value = state.tool; }
}
```

By wrapping the label text and the field in a `<label>` element, we tell the browser that the label belongs to that field so that you can, for example, click the label to focus the field.

We also need to be able to change the color, so let's add a control for that. An HTML `<input>` element with a `type` attribute of `color` gives us a form field that is specialized for selecting colors. Such a field's value is always a CSS color code in `"#RRGGBB"` format (red, green, and blue components, two digits per color). The browser will show a color picker interface when the user interacts with it.

Depending on the browser, the color picker might look like this:

This control creates such a field and wires it up to stay synchronized with the application state's `color` property.

```
class ColorSelect {
  constructor(state, {dispatch}) {
    this.input = elt("input", {
      type: "color",
      value: state.color,
      onchange: () => dispatch({color: this.input.value})
    });
    this.dom = elt("label", null, "🎨 Color: ", this.input);
  }
  syncState(state) { this.input.value = state.color; }
}
```

Drawing tools

Before we can draw anything, we need to implement the tools that will control the functionality of mouse or touch events on the canvas.

The most basic tool is the draw tool, which changes any pixel you click or tap to the currently selected color. It dispatches an action that updates the picture to a version in which the pointed-at pixel is given the currently selected color.

```
function draw(pos, state, dispatch) {
  function drawPixel({x, y}, state) {
```

```
    let drawn = {x, y, color: state.color};
    dispatch({picture: state.picture.draw([drawn])});
  }
  drawPixel(pos, state);
  return drawPixel;
}
```

The function immediately calls the `drawPixel` function but then also returns it so that it is called again for newly touched pixels when the user drags or swipes over the picture.

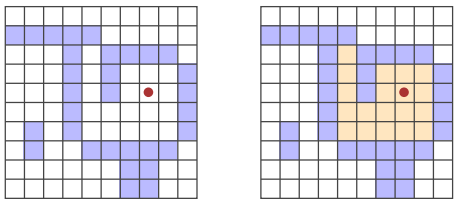
To draw larger shapes, it can be useful to quickly create rectangles. The `rectangle` tool draws a rectangle between the point where you start dragging and the point that you drag to.

```
function rectangle(start, state, dispatch) {
  function drawRectangle(pos) {
    let xStart = Math.min(start.x, pos.x);
    let yStart = Math.min(start.y, pos.y);
    let xEnd = Math.max(start.x, pos.x);
    let yEnd = Math.max(start.y, pos.y);
    let drawn = [];
    for (let y = yStart; y <= yEnd; y++) {
      for (let x = xStart; x <= xEnd; x++) {
        drawn.push({x, y, color: state.color});
      }
    }
    dispatch({picture: state.picture.draw(drawn)});
  }
  drawRectangle(start);
  return drawRectangle;
}
```

An important detail in this implementation is that when dragging, the rectangle is redrawn on the picture from the *original* state. That way, you can make the rectangle larger and smaller again while creating it, without the intermediate rectangles sticking around in the final picture. This is one of the reasons why immutable picture objects are useful—we'll see another reason later.

Implementing flood fill is somewhat more involved. This is a tool that fills the pixel under the pointer and all adjacent pixels that have the same color. "Adjacent" means directly horizontally or vertically adjacent, not diagonally. This picture illustrates the set of pixels colored when the flood fill tool is used

at the marked pixel:



Interestingly, the way we'll do this looks a bit like the pathfinding code from Chapter 7. Whereas that code searched through a graph to find a route, this code searches through a grid to find all "connected" pixels. The problem of keeping track of a branching set of possible routes is similar.

```
const around = [{dx: -1, dy: 0}, {dx: 1, dy: 0},
                {dx: 0, dy: -1}, {dx: 0, dy: 1}];

function fill({x, y}, state, dispatch) {
  let targetColor = state.picture.pixel(x, y);
  let drawn = [{x, y, color: state.color}];
  for (let done = 0; done < drawn.length; done++) {
    for (let {dx, dy} of around) {
      let x = drawn[done].x + dx, y = drawn[done].y + dy;
      if (x >= 0 && x < state.picture.width &&
          y >= 0 && y < state.picture.height &&
          state.picture.pixel(x, y) == targetColor &&
          !drawn.some(p => p.x == x && p.y == y)) {
        drawn.push({x, y, color: state.color});
      }
    }
  }
  dispatch({picture: state.picture.draw(drawn)});
}
```

The array of drawn pixels doubles as the function's work list. For each pixel reached, we have to see whether any adjacent pixels have the same color and haven't already been painted over. The loop counter lags behind the length of the `drawn` array as new pixels are added. Any pixels ahead of it still need to be explored. When it catches up with the length, no unexplored pixels remain, and the function is done.

The final tool is a color picker, which allows you to point at a color in the picture to use it as the current drawing color.

```
function pick(pos, state, dispatch) {
```

```
    dispatch({color: state.picture.pixel(pos.x, pos.y)});
  }
```

Saving and loading

When we've drawn our masterpiece, we'll want to save it for later. We should add a button for downloading the current picture as an image file. This control provides that button:

```
class SaveButton {
  constructor(state) {
    this.picture = state.picture;
    this.dom = elt("button", {
      onclick: () => this.save()
    }, "💾 Save");
  }
  save() {
    let canvas = elt("canvas");
    drawPicture(this.picture, canvas, 1);
    let link = elt("a", {
      href: canvas.toDataURL(),
      download: "pixelart.png"
    });
    document.body.appendChild(link);
    link.click();
    link.remove();
  }
  syncState(state) { this.picture = state.picture; }
}
```

The component keeps track of the current picture so that it can access it when saving. To create the image file, it uses a `<canvas>` element that it draws the picture on (at a scale of one pixel per pixel).

The `toDataURL` method on a canvas element creates a URL that starts with `data:`. Unlike `http:` and `https:` URLs, data URLs contain the whole resource in the URL. They are usually very long, but they allow us to create working links to arbitrary pictures, right here in the browser.

To actually get the browser to download the picture, we then create a link element that points at this URL and has a `download` attribute. Such links, when clicked, make the browser show a file save dialog. We add that link to

the document, simulate a click on it, and remove it again.

You can do a lot with browser technology, but sometimes the way to do it is rather odd.

And it gets worse. We'll also want to be able to load existing image files into our application. To do that, we again define a button component.

```
class LoadButton {
  constructor(_, {dispatch}) {
    this.dom = elt("button", {
      onclick: () => startLoad(dispatch)
    }, "📁 Load");
  }
  syncState() {}
}

function startLoad(dispatch) {
  let input = elt("input", {
    type: "file",
    onchange: () => finishLoad(input.files[0], dispatch)
  });
  document.body.appendChild(input);
  input.click();
  input.remove();
}
```

To get access to a file on the user's computer, we need the user to select the file through a file input field. But I don't want the load button to look like a file input field, so we create the file input when the button is clicked and then pretend that this file input itself was clicked.

When the user has selected a file, we can use `FileReader` to get access to its contents, again as a data URL. That URL can be used to create an `<img>` element, but because we can't get direct access to the pixels in such an image, we can't create a `Picture` object from that.

```
function finishLoad(file, dispatch) {
  if (file == null) return;
  let reader = new FileReader();
  reader.addEventListener("load", () => {
    let image = elt("img", {
      onload: () => dispatch({
        picture: pictureFromImage(image)
      }),
      src: reader.result
```

```
    });
  });
  reader.readAsDataURL(file);
}
```

To get access to the pixels, we must first draw the picture to a `<canvas>` element. The canvas context has a `getImageData` method that allows a script to read its pixels. So, once the picture is on the canvas, we can access it and construct a `Picture` object.

```
function pictureFromImage(image) {
  let width = Math.min(100, image.width);
  let height = Math.min(100, image.height);
  let canvas = elt("canvas", {width, height});
  let cx = canvas.getContext("2d");
  cx.drawImage(image, 0, 0);
  let pixels = [];
  let {data} = cx.getImageData(0, 0, width, height);

  function hex(n) {
    return n.toString(16).padStart(2, "0");
  }
  for (let i = 0; i < data.length; i += 4) {
    let [r, g, b] = data.slice(i, i + 3);
    pixels.push("#" + hex(r) + hex(g) + hex(b));
  }
  return new Picture(width, height, pixels);
}
```

We'll limit the size of images to 100 by 100 pixels since anything bigger will look *huge* on our display and might slow down the interface.

The `data` property of the object returned by `getImageData` is an array of color components. For each pixel in the rectangle specified by the arguments, it contains four values, which represent the red, green, blue, and *alpha* components of the pixel's color, as numbers between 0 and 255. The alpha part represents opacity—when it is zero, the pixel is fully transparent, and when it is 255, it is fully opaque. For our purpose, we can ignore it.

The two hexadecimal digits per component, as used in our color notation, correspond precisely to the 0 to 255 range—two base-16 digits can express $16^2 = 256$ different numbers. The `toString` method of numbers can be given a base as argument, so `n.toString(16)` will produce a string representation in base 16. We have to make sure that each number takes up two digits, so the

`hex` helper function calls `padStart` to add a leading zero when necessary.

We can load and save now! That leaves one more feature before we're done.

Undo history

Half of the process of editing is making little mistakes and correcting them. So an important feature in a drawing program is an undo history.

To be able to undo changes, we need to store previous versions of the picture. Since it's an immutable value, that is easy. But it does require an additional field in the application state.

We'll add a `done` array to keep previous versions of the picture. Maintaining this property requires a more complicated state update function that adds pictures to the array.

But we don't want to store *every* change, only changes a certain amount of time apart. To be able to do that, we'll need a second property, `doneAt`, tracking the time at which we last stored a picture in the history.

```
function historyUpdateState(state, action) {
  if (action.undo == true) {
    if (state.done.length == 0) return state;
    return Object.assign({}, state, {
      picture: state.done[0],
      done: state.done.slice(1),
      doneAt: 0
    });
  } else if (action.picture &&
             state.doneAt < Date.now() - 1000) {
    return Object.assign({}, state, action, {
      done: [state.picture, ...state.done],
      doneAt: Date.now()
    });
  } else {
    return Object.assign({}, state, action);
  }
}
```

When the action is an undo action, the function takes the most recent picture from the history and makes that the current picture. It sets `doneAt` to zero so that the next change is guaranteed to store the picture back in the history, allowing you to revert to it another time if you want.

Otherwise, if the action contains a new picture and the last time we stored something is more than a second (1000 milliseconds) ago, the `done` and `doneAt`

properties are updated to store the previous picture.

The undo button component doesn't do much. It dispatches undo actions when clicked and disables itself when there is nothing to undo.

```
class UndoButton {
  constructor(state, {dispatch}) {
    this.dom = elt("button", {
      onclick: () => dispatch({undo: true}),
      disabled: state.done.length == 0
    }, "⮌ Undo");
  }
  syncState(state) {
    this.dom.disabled = state.done.length == 0;
  }
}
```

Let's draw

To set up the application, we need to create a state, a set of tools, a set of controls, and a dispatch function. We can pass them to the `PixelEditor` constructor to create the main component. Since we'll need to create several editors in the exercises, we first define some bindings.

```
const startState = {
  tool: "draw",
  color: "#000000",
  picture: Picture.empty(60, 30, "#f0f0f0"),
  done: [],
  doneAt: 0
};

const baseTools = {draw, fill, rectangle, pick};

const baseControls = [
  ToolSelect, ColorSelect, SaveButton, LoadButton, UndoButton
];

function startPixelEditor({state = startState,
                           tools = baseTools,
                           controls = baseControls}) {
  let app = new PixelEditor(state, {
    tools,
    controls,
```

```
    dispatch(action) {
      state = historyUpdateState(state, action);
      app.syncState(state);
    }
  });
  return app.dom;
}
```

When destructuring an object or array, you can use `=` after a binding name to give the binding a default value, which is used when the property is missing or holds `undefined`. The `startPixelEditor` function makes use of this to accept an object with a number of optional properties as an argument. If you don't provide a `tools` property, for example, `tools` will be bound to `baseTools`.

This is how we get an actual editor on the screen:

```
<div></div>
<script>
  document.querySelector("div")
    .appendChild(startPixelEditor({}));
</script>
```

Why is this so hard?

Browser technology is amazing. It provides a powerful set of interface building blocks, ways to style and manipulate them, and tools to inspect and debug your applications. The software you write for the browser can be run on almost every computer and phone on the planet.

At the same time, browser technology is ridiculous. You have to learn a large number of silly tricks and obscure facts to master it, and the default programming model it provides is so problematic that most programmers prefer to cover it in several layers of abstraction rather than deal with it directly.

And though the situation is definitely improving, it mostly does so in the form of more elements being added to address shortcomings—creating even more complexity. A feature used by a million websites can't really be replaced. Even if it could, it would be hard to decide what it should be replaced with.

Technology never exists in a vacuum—we're constrained by our tools and the social, economic, and historical factors that produced them. This can be annoying, but it is generally more productive to try to build a good understanding of how the *existing* technical reality works—and why it is the way it

is—than to rage against it or hold out for another reality.

New abstractions *can* be helpful. The component model and data flow convention I used in this chapter is a crude form of that. As mentioned, there are libraries that try to make user interface programming more pleasant. At the time of writing, React and Angular are popular choices, but there's a whole cottage industry of such frameworks. If you're interested in programming web applications, I recommend investigating a few of them to understand how they work and what benefits they provide.

Exercises

There is still room for improvement in our program. Let's add a few more features as exercises.

Keyboard bindings

Add keyboard shortcuts to the application. The first letter of a tool's name selects the tool, and CONTROL-Z or COMMAND-Z activates undo.

Do this by modifying the `PixelEditor` component. Add a `tabIndex` property of 0 to the wrapping `<div>` element so that it can receive keyboard focus. Note that the *property* corresponding to the `tabindex` *attribute* is called `tabIndex`, with a capital I, and our `elt` function expects property names. Register the key event handlers directly on that element. This means you have to click, touch, or tab to the application before you can interact with it with the keyboard.

Remember that keyboard events have `ctrlKey` and `metaKey` (for the COMMAND key on Mac) properties that you can use to see whether those keys are held down.

Efficient drawing

During drawing, the majority of work that our application does happens in `drawPicture`. Creating a new state and updating the rest of the DOM isn't very expensive, but repainting all the pixels on the canvas is quite a bit of work.

Find a way to make the `syncState` method of `PictureCanvas` faster by redrawing only the pixels that actually changed.

Remember that `drawPicture` is also used by the save button, so if you change it, either make sure the changes don't break the old use or create a new version with a different name.

Also note that changing the size of a `<canvas>` element, by setting its `width` or `height` properties, clears it, making it entirely transparent again.

Circles

Define a tool called `circle` that draws a filled circle when you drag. The center of the circle lies at the point where the drag or touch gesture starts, and its radius is determined by the distance dragged.

Proper lines

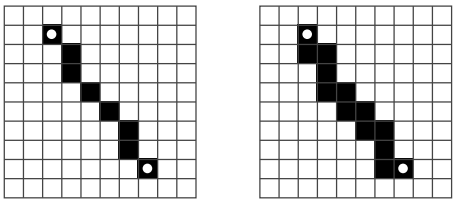
This is a more advanced exercise than the preceding two, and it will require you to design a solution to a nontrivial problem. Make sure you have plenty of time and patience before starting to work on this exercise, and do not get discouraged by initial failures.

On most browsers, when you select the `draw` tool and quickly drag across the picture, you don't get a closed line. Rather, you get dots with gaps between them because the `"mousemove"` or `"touchmove"` events did not fire quickly enough to hit every pixel.

Improve the `draw` tool to make it draw a full line. This means you have to make the motion handler function remember the previous position and connect that to the current one.

To do this, since the pixels can be an arbitrary distance apart, you'll have to write a general line drawing function.

A line between two pixels is a connected chain of pixels, as straight as possible, going from the start to the end. Diagonally adjacent pixels count as a connected. So a slanted line should look like the picture on the left, not the picture on the right.



Finally, if we have code that draws a line between two arbitrary points, we might as well use it to also define a `line` tool, which draws a straight line between the start and end of a drag.

> *"A student asked, 'The programmers of old used only simple machines and no programming languages, yet they made beautiful programs. Why do we use complicated machines and programming languages?'. Fu-Tzu replied, 'The builders of old used only sticks and clay, yet they made beautiful huts.'"*
>
> —Master Yuan-Ma, The Book of Programming

CHAPTER 20

Node.js

So far, we have used the JavaScript language in a single environment: the browser. This chapter and the next one will briefly introduce Node.js, a program that allows you to apply your JavaScript skills outside of the browser. With it, you can build anything from small command line tools to HTTP servers that power dynamic websites.

These chapters aim to teach you the main concepts that Node.js uses and to give you enough information to write useful programs for it. They do not try to be a complete, or even a thorough, treatment of the platform.

If you want to follow along and run the code in this chapter, you'll need to install Node.js version 10.1 or higher. To do so, go to *https://nodejs.org* and follow the installation instructions for your operating system. You can also find further documentation for Node.js there.

Background

One of the more difficult problems with writing systems that communicate over the network is managing input and output—that is, the reading and writing of data to and from the network and hard drive. Moving data around takes time, and scheduling it cleverly can make a big difference in how quickly a system responds to the user or to network requests.

In such programs, asynchronous programming is often helpful. It allows the program to send and receive data from and to multiple devices at the same time without complicated thread management and synchronization.

Node was initially conceived for the purpose of making asynchronous programming easy and convenient. JavaScript lends itself well to a system like Node. It is one of the few programming languages that does not have a built-in way to do in- and output. Thus, JavaScript could be fit onto Node's rather eccentric approach to in- and output without ending up with two inconsistent interfaces. In 2009, when Node was being designed, people were already doing callback-based programming in the browser, so the community around the

language was used to an asynchronous programming style.

The node command

When Node.js is installed on a system, it provides a program called `node`, which is used to run JavaScript files. Say you have a file `hello.js`, containing this code:

```
let message = "Hello world";
console.log(message);
```

You can then run `node` from the command line like this to execute the program:

```
$ node hello.js
Hello world
```

The `console.log` method in Node does something similar to what it does in the browser. It prints out a piece of text. But in Node, the text will go to the process's standard output stream, rather than to a browser's JavaScript console. When running `node` from the command line, that means you see the logged values in your terminal.

If you run `node` without giving it a file, it provides you with a prompt at which you can type JavaScript code and immediately see the result.

```
$ node
> 1 + 1
2
> [-1, -2, -3].map(Math.abs)
[1, 2, 3]
> process.exit(0)
$
```

The `process` binding, just like the `console` binding, is available globally in Node. It provides various ways to inspect and manipulate the current program. The `exit` method ends the process and can be given an exit status code, which tells the program that started `node` (in this case, the command line shell) whether the program completed successfully (code zero) or encountered an error (any other code).

To find the command line arguments given to your script, you can read

`process.argv`, which is an array of strings. Note that it also includes the name of the `node` command and your script name, so the actual arguments start at index 2. If `showargv.js` contains the statement `console.log(process.argv)`, you could run it like this:

```
$ node showargv.js one --and two
["node", "/tmp/showargv.js", "one", "--and", "two"]
```

All the standard JavaScript global bindings, such as `Array`, `Math`, and `JSON`, are also present in Node's environment. Browser-related functionality, such as `document` or `prompt`, is not.

Modules

Beyond the bindings I mentioned, such as `console` and `process`, Node puts few additional bindings in the global scope. If you want to access built-in functionality, you have to ask the module system for it.

The CommonJS module system, based on the `require` function, was described in Chapter 10. This system is built into Node and is used to load anything from built-in modules to downloaded packages to files that are part of your own program.

When `require` is called, Node has to resolve the given string to an actual file that it can load. Pathnames that start with `/`, `./`, or `../` are resolved relative to the current module's path, where `.` stands for the current directory, `../` for one directory up, and `/` for the root of the file system. So if you ask for `"./graph"` from the file `/tmp/robot/robot.js`, Node will try to load the file `/tmp/robot/graph.js`.

The `.js` extension may be omitted, and Node will add it if such a file exists. If the required path refers to a directory, Node will try to load the file named `index.js` in that directory.

When a string that does not look like a relative or absolute path is given to `require`, it is assumed to refer to either a built-in module or a module installed in a `node_modules` directory. For example, `require("fs")` will give you Node's built-in file system module. And `require("robot")` might try to load the library found in `node_modules/robot/`. A common way to install such libraries is by using NPM, which we'll come back to in a moment.

Let's set up a small project consisting of two files. The first one, called `main.js`, defines a script that can be called from the command line to reverse a string.

```
const {reverse} = require("./reverse");

// Index 2 holds the first actual command line argument
let argument = process.argv[2];

console.log(reverse(argument));
```

The file `reverse.js` defines a library for reversing strings, which can be used both by this command line tool and by other scripts that need direct access to a string-reversing function.

```
exports.reverse = function(string) {
  return Array.from(string).reverse().join("");
};
```

Remember that adding properties to `exports` adds them to the interface of the module. Since Node.js treats files as CommonJS modules, `main.js` can take the exported `reverse` function from `reverse.js`.

We can now call our tool like this:

```
$ node main.js JavaScript
tpircSavaJ
```

Installing with NPM

NPM, which was introduced in Chapter 10, is an online repository of JavaScript modules, many of which are specifically written for Node. When you install Node on your computer, you also get the `npm` command, which you can use to interact with this repository.

NPM's main use is downloading packages. We saw the `ini` package in Chapter 10. We can use NPM to fetch and install that package on our computer.

```
$ npm install ini
npm WARN enoent ENOENT: no such file or directory,
         open '/tmp/package.json'
+ ini@1.3.5
added 1 package in 0.552s

$ node
> const {parse} = require("ini");
```

```
> parse("x = 1\ny = 2");
{ x: '1', y: '2' }
```

After running `npm install`, NPM will have created a directory called `node_modules`. Inside that directory will be an `ini` directory that contains the library. You can open it and look at the code. When we call `require("ini")`, this library is loaded, and we can call its `parse` property to parse a configuration file.

By default NPM installs packages under the current directory, rather than in a central place. If you are used to other package managers, this may seem unusual, but it has advantages—it puts each application in full control of the packages it installs and makes it easier to manage versions and clean up when removing an application.

Package files

In the `npm install` example, you could see a warning about the fact that the `package.json` file did not exist. It is recommended to create such a file for each project, either manually or by running `npm init`. It contains some information about the project, such as its name and version, and lists its dependencies.

The robot simulation from Chapter 7, as modularized in the exercise in Chapter 10, might have a `package.json` file like this:

```
{
  "author": "Marijn Haverbeke",
  "name": "eloquent-javascript-robot",
  "description": "Simulation of a package-delivery robot",
  "version": "1.0.0",
  "main": "run.js",
  "dependencies": {
    "dijkstrajs": "^1.0.1",
    "random-item": "^1.0.0"
  },
  "license": "ISC"
}
```

When you run `npm install` without naming a package to install, NPM will install the dependencies listed in `package.json`. When you install a specific package that is not already listed as a dependency, NPM will add it to `package.json`.

Versions

A `package.json` file lists both the program's own version and versions for its dependencies. Versions are a way to deal with the fact that packages evolve separately, and code written to work with a package as it existed at one point may not work with a later, modified version of the package.

NPM demands that its packages follow a schema called *semantic versioning*, which encodes some information about which versions are *compatible* (don't break the old interface) in the version number. A semantic version consists of three numbers, separated by periods, such as `2.3.0`. Every time new functionality is added, the middle number has to be incremented. Every time compatibility is broken, so that existing code that uses the package might not work with the new version, the first number has to be incremented.

A caret character (`^`) in front of the version number for a dependency in `package.json` indicates that any version compatible with the given number may be installed. So, for example, `"^2.3.0"` would mean that any version greater than or equal to 2.3.0 and less than 3.0.0 is allowed.

The `npm` command is also used to publish new packages or new versions of packages. If you run `npm publish` in a directory that has a `package.json` file, it will publish a package with the name and version listed in the JSON file to the registry. Anyone can publish packages to NPM—though only under a package name that isn't in use yet since it would be somewhat scary if random people could update existing packages.

Since the `npm` program is a piece of software that talks to an open system—the package registry—there is nothing unique about what it does. Another program, `yarn`, which can be installed from the NPM registry, fills the same role as `npm` using a somewhat different interface and installation strategy.

This book won't delve further into the details of NPM usage. Refer to *https://npmjs.org* for further documentation and a way to search for packages.

The file system module

One of the most commonly used built-in modules in Node is the `fs` module, which stands for *file system*. It exports functions for working with files and directories.

For example, the function called `readFile` reads a file and then calls a callback with the file's contents.

```
let {readFile} = require("fs");
readFile("file.txt", "utf8", (error, text) => {
```

```
  if (error) throw error;
  console.log("The file contains:", text);
});
```

The second argument to `readFile` indicates the *character encoding* used to decode the file into a string. There are several ways in which text can be encoded to binary data, but most modern systems use UTF-8. So unless you have reasons to believe another encoding is used, pass `"utf8"` when reading a text file. If you do not pass an encoding, Node will assume you are interested in the binary data and will give you a `Buffer` object instead of a string. This is an array-like object that contains numbers representing the bytes (8-bit chunks of data) in the files.

```
const {readFile} = require("fs");
readFile("file.txt", (error, buffer) => {
  if (error) throw error;
  console.log("The file contained", buffer.length, "bytes.",
              "The first byte is:", buffer[0]);
});
```

A similar function, `writeFile`, is used to write a file to disk.

```
const {writeFile} = require("fs");
writeFile("graffiti.txt", "Node was here", err => {
  if (err) console.log(`Failed to write file: ${err}`);
  else console.log("File written.");
});
```

Here it was not necessary to specify the encoding—`writeFile` will assume that when it is given a string to write, rather than a `Buffer` object, it should write it out as text using its default character encoding, which is UTF-8.

The `fs` module contains many other useful functions: `readdir` will return the files in a directory as an array of strings, `stat` will retrieve information about a file, `rename` will rename a file, `unlink` will remove one, and so on. See the documentation at *https://nodejs.org* for specifics.

Most of these take a callback function as the last parameter, which they call either with an error (the first argument) or with a successful result (the second). As we saw in Chapter 11, there are downsides to this style of programming—the biggest one being that error handling becomes verbose and error-prone.

Though promises have been part of JavaScript for a while, at the time of writ-

ing their integration into Node.js is still a work in progress. There is an object `promises` exported from the `fs` package since version 10.1 that contains most of the same functions as `fs` but uses promises rather than callback functions.

```
const {readFile} = require("fs").promises;
readFile("file.txt", "utf8")
  .then(text => console.log("The file contains:", text));
```

Sometimes you don't need asynchronicity, and it just gets in the way. Many of the functions in `fs` also have a synchronous variant, which has the same name with `Sync` added to the end. For example, the synchronous version of `readFile` is called `readFileSync`.

```
const {readFileSync} = require("fs");
console.log("The file contains:",
            readFileSync("file.txt", "utf8"));
```

Do note that while such a synchronous operation is being performed, your program is stopped entirely. If it should be responding to the user or to other machines on the network, being stuck on a synchronous action might produce annoying delays.

The HTTP module

Another central module is called `http`. It provides functionality for running HTTP servers and making HTTP requests.

This is all it takes to start an HTTP server:

```
const {createServer} = require("http");
let server = createServer((request, response) => {
  response.writeHead(200, {"Content-Type": "text/html"});
  response.write(`
    <h1>Hello!</h1>
    <p>You asked for <code>${request.url}</code></p>`);
  response.end();
});
server.listen(8000);
console.log("Listening! (port 8000)");
```

If you run this script on your own machine, you can point your web browser at *http://localhost:8000/hello* to make a request to your server. It will respond

with a small HTML page.

The function passed as argument to `createServer` is called every time a client connects to the server. The `request` and `response` bindings are objects representing the incoming and outgoing data. The first contains information about the request, such as its `url` property, which tells us to what URL the request was made.

So, when you open that page in your browser, it sends a request to your own computer. This causes the server function to run and send back a response, which you can then see in the browser.

To send something back, you call methods on the `response` object. The first, `writeHead`, will write out the response headers (see Chapter 18). You give it the status code (200 for "OK" in this case) and an object that contains header values. The example sets the `Content-Type` header to inform the client that we'll be sending back an HTML document.

Next, the actual response body (the document itself) is sent with `response.write`. You are allowed to call this method multiple times if you want to send the response piece by piece, for example to stream data to the client as it becomes available. Finally, `response.end` signals the end of the response.

The call to `server.listen` causes the server to start waiting for connections on port 8000. This is why you have to connect to *localhost:8000* to speak to this server, rather than just *localhost*, which would use the default port 80.

When you run this script, the process just sits there and waits. When a script is listening for events—in this case, network connections—`node` will not automatically exit when it reaches the end of the script. To close it, press CONTROL-C.

A real web server usually does more than the one in the example—it looks at the request's method (the `method` property) to see what action the client is trying to perform and looks at the request's URL to find out which resource this action is being performed on. We'll see a more advanced server later in this chapter.

To act as an HTTP *client*, we can use the `request` function in the `http` module.

```
const {request} = require("http");
let requestStream = request({
  hostname: "eloquentjavascript.net",
  path: "/20_node.html",
  method: "GET",
  headers: {Accept: "text/html"}
}, response => {
  console.log("Server responded with status code",
```

```
                response.statusCode);
});
requestStream.end();
```

The first argument to `request` configures the request, telling Node what server to talk to, what path to request from that server, which method to use, and so on. The second argument is the function that should be called when a response comes in. It is given an object that allows us to inspect the response, for example to find out its status code.

Just like the `response` object we saw in the server, the object returned by `request` allows us to stream data into the request with the `write` method and finish the request with the `end` method. The example does not use `write` because `GET` requests should not contain data in their request body.

There's a similar `request` function in the `https` module that can be used to make requests to `https:` URLs.

Making requests with Node's raw functionality is rather verbose. There are much more convenient wrapper packages available on NPM. For example, `node-fetch` provides the promise-based `fetch` interface that we know from the browser.

Streams

We have seen two instances of writable streams in the HTTP examples—namely, the response object that the server could write to and the request object that was returned from `request`.

Writable streams are a widely used concept in Node. Such objects have a `write` method that can be passed a string or a `Buffer` object to write something to the stream. Their `end` method closes the stream and optionally takes a value to write to the stream before closing. Both of these methods can also be given a callback as an additional argument, which they will call when the writing or closing has finished.

It is possible to create a writable stream that points at a file with the `createWriteStream` function from the `fs` module. Then you can use the `write` method on the resulting object to write the file one piece at a time, rather than in one shot as with `writeFile`.

Readable streams are a little more involved. Both the `request` binding that was passed to the HTTP server's callback and the `response` binding passed to the HTTP client's callback are readable streams—a server reads requests and then writes responses, whereas a client first writes a request and then reads

a response. Reading from a stream is done using event handlers, rather than methods.

Objects that emit events in Node have a method called `on` that is similar to the `addEventListener` method in the browser. You give it an event name and then a function, and it will register that function to be called whenever the given event occurs.

Readable streams have `"data"` and `"end"` events. The first is fired every time data comes in, and the second is called whenever the stream is at its end. This model is most suited for *streaming* data that can be immediately processed, even when the whole document isn't available yet. A file can be read as a readable stream by using the `createReadStream` function from `fs`.

This code creates a server that reads request bodies and streams them back to the client as all-uppercase text:

```
const {createServer} = require("http");
createServer((request, response) => {
  response.writeHead(200, {"Content-Type": "text/plain"});
  request.on("data", chunk =>
    response.write(chunk.toString().toUpperCase()));
  request.on("end", () => response.end());
}).listen(8000);
```

The `chunk` value passed to the data handler will be a binary `Buffer`. We can convert this to a string by decoding it as UTF-8 encoded characters with its `toString` method.

The following piece of code, when run with the uppercasing server active, will send a request to that server and write out the response it gets:

```
const {request} = require("http");
request({
  hostname: "localhost",
  port: 8000,
  method: "POST"
}, response => {
  response.on("data", chunk =>
    process.stdout.write(chunk.toString()));
}).end("Hello server");
// → HELLO SERVER
```

The example writes to `process.stdout` (the process's standard output, which is a writable stream) instead of using `console.log`. We can't use `console.log`

because it adds an extra newline character after each piece of text that it writes, which isn't appropriate here since the response may come in as multiple chunks.

A FILE SERVER

Let's combine our newfound knowledge about HTTP servers and working with the file system to create a bridge between the two: an HTTP server that allows remote access to a file system. Such a server has all kinds of uses—it allows web applications to store and share data, or it can give a group of people shared access to a bunch of files.

When we treat files as HTTP resources, the HTTP methods `GET`, `PUT`, and `DELETE` can be used to read, write, and delete the files, respectively. We will interpret the path in the request as the path of the file that the request refers to.

We probably don't want to share our whole file system, so we'll interpret these paths as starting in the server's working directory, which is the directory in which it was started. If I ran the server from `/tmp/public/` (or `C:\tmp\public\` on Windows), then a request for `/file.txt` should refer to `/tmp/public/file.txt` (or `C:\tmp\public\file.txt`).

We'll build the program piece by piece, using an object called `methods` to store the functions that handle the various HTTP methods. Method handlers are `async` functions that get the request object as argument and return a promise that resolves to an object that describes the response.

```
const {createServer} = require("http");

const methods = Object.create(null);

createServer((request, response) => {
  let handler = methods[request.method] || notAllowed;
  handler(request)
    .catch(error => {
      if (error.status != null) return error;
      return {body: String(error), status: 500};
    })
    .then(({body, status = 200, type = "text/plain"}) => {
       response.writeHead(status, {"Content-Type": type});
       if (body && body.pipe) body.pipe(response);
       else response.end(body);
    });
}).listen(8000);
```

```
async function notAllowed(request) {
  return {
    status: 405,
    body: `Method ${request.method} not allowed.`
  };
}
```

This starts a server that just returns 405 error responses, which is the code used to indicate that the server refuses to handle a given method.

When a request handler's promise is rejected, the `catch` call translates the error into a response object, if it isn't one already, so that the server can send back an error response to inform the client that it failed to handle the request.

The `status` field of the response description may be omitted, in which case it defaults to 200 (OK). The content type, in the `type` property, can also be left off, in which case the response is assumed to be plain text.

When the value of `body` is a readable stream, it will have a `pipe` method that is used to forward all content from a readable stream to a writable stream. If not, it is assumed to be either `null` (no body), a string, or a buffer, and it is passed directly to the response's `end` method.

To figure out which file path corresponds to a request URL, the `urlPath` function uses Node's built-in `url` module to parse the URL. It takes its pathname, which will be something like `"/file.txt"`, decodes that to get rid of the `%20`-style escape codes, and resolves it relative to the program's working directory.

```
const {parse} = require("url");
const {resolve, sep} = require("path");

const baseDirectory = process.cwd();

function urlPath(url) {
  let {pathname} = parse(url);
  let path = resolve(decodeURIComponent(pathname).slice(1));
  if (path != baseDirectory &&
      !path.startsWith(baseDirectory + sep)) {
    throw {status: 403, body: "Forbidden"};
  }
  return path;
}
```

As soon as you set up a program to accept network requests, you have to

start worrying about security. In this case, if we aren't careful, it is likely that we'll accidentally expose our whole file system to the network.

File paths are strings in Node. To map such a string to an actual file, there is a nontrivial amount of interpretation going on. Paths may, for example, include `../` to refer to a parent directory. So one obvious source of problems would be requests for paths like `/../secret_file`.

To avoid such problems, `urlPath` uses the `resolve` function from the `path` module, which resolves relative paths. It then verifies that the result is *below* the working directory. The `process.cwd` function (where `cwd` stands for "current working directory") can be used to find this working directory. The `sep` binding from the `path` package is the system's path separator—a backslash on Windows and a forward slash on most other systems. When the path doesn't start with the base directory, the function throws an error response object, using the HTTP status code indicating that access to the resource is forbidden.

We'll set up the `GET` method to return a list of files when reading a directory and to return the file's content when reading a regular file.

One tricky question is what kind of `Content-Type` header we should set when returning a file's content. Since these files could be anything, our server can't simply return the same content type for all of them. NPM can help us again here. The `mime` package (content type indicators like `text/plain` are also called *MIME types*) knows the correct type for a large number of file extensions.

The following `npm` command, in the directory where the server script lives, installs a specific version of `mime`:

```
$ npm install mime@2.2.0
```

When a requested file does not exist, the correct HTTP status code to return is 404. We'll use the `stat` function, which looks up information about a file, to find out both whether the file exists and whether it is a directory.

```
const {createReadStream} = require("fs");
const {stat, readdir} = require("fs").promises;
const mime = require("mime");

methods.GET = async function(request) {
  let path = urlPath(request.url);
  let stats;
  try {
    stats = await stat(path);
  } catch (error) {
    if (error.code != "ENOENT") throw error;
```

```
    else return {status: 404, body: "File not found"};
  }
  if (stats.isDirectory()) {
    return {body: (await readdir(path)).join("\n")};
  } else {
    return {body: createReadStream(path),
            type: mime.getType(path)};
  }
};
```

Because it has to touch the disk and thus might take a while, `stat` is asynchronous. Since we're using promises rather than callback style, it has to be imported from `promises` instead of directly from `fs`.

When the file does not exist, `stat` will throw an error object with a `code` property of `"ENOENT"`. These somewhat obscure, Unix-inspired codes are how you recognize error types in Node.

The `stats` object returned by `stat` tells us a number of things about a file, such as its size (`size` property) and its modification date (`mtime` property). Here we are interested in the question of whether it is a directory or a regular file, which the `isDirectory` method tells us.

We use `readdir` to read the array of files in a directory and return it to the client. For normal files, we create a readable stream with `createReadStream` and return that as the body, along with the content type that the `mime` package gives us for the file's name.

The code to handle `DELETE` requests is slightly simpler.

```
const {rmdir, unlink} = require("fs").promises;

methods.DELETE = async function(request) {
  let path = urlPath(request.url);
  let stats;
  try {
    stats = await stat(path);
  } catch (error) {
    if (error.code != "ENOENT") throw error;
    else return {status: 204};
  }
  if (stats.isDirectory()) await rmdir(path);
  else await unlink(path);
  return {status: 204};
};
```

When an HTTP response does not contain any data, the status code 204 ("no content") can be used to indicate this. Since the response to deletion doesn't need to transmit any information beyond whether the operation succeeded, that is a sensible thing to return here.

You may be wondering why trying to delete a nonexistent file returns a success status code, rather than an error. When the file that is being deleted is not there, you could say that the request's objective is already fulfilled. The HTTP standard encourages us to make requests *idempotent*, which means that making the same request multiple times produces the same result as making it once. In a way, if you try to delete something that's already gone, the effect you were trying to do has been achieved—the thing is no longer there.

This is the handler for `PUT` requests:

```
const {createWriteStream} = require("fs");

function pipeStream(from, to) {
  return new Promise((resolve, reject) => {
    from.on("error", reject);
    to.on("error", reject);
    to.on("finish", resolve);
    from.pipe(to);
  });
}

methods.PUT = async function(request) {
  let path = urlPath(request.url);
  await pipeStream(request, createWriteStream(path));
  return {status: 204};
};
```

We don't need to check whether the file exists this time—if it does, we'll just overwrite it. We again use `pipe` to move data from a readable stream to a writable one, in this case from the request to the file. But since `pipe` isn't written to return a promise, we have to write a wrapper, `pipeStream`, that creates a promise around the outcome of calling `pipe`.

When something goes wrong when opening the file, `createWriteStream` will still return a stream, but that stream will fire an `"error"` event. The output stream to the request may also fail, for example if the network goes down. So we wire up both streams' `"error"` events to reject the promise. When `pipe` is done, it will close the output stream, which causes it to fire a `"finish"` event. That's the point where we can successfully resolve the promise (returning nothing).

The full script for the server is available at *https://eloquentjavascript.net/code/file_server.js*. You can download that and, after installing its dependencies, run it with Node to start your own file server. And, of course, you can modify and extend it to solve this chapter's exercises or to experiment.

The command line tool `curl`, widely available on Unix-like systems (such as macOS and Linux), can be used to make HTTP requests. The following session briefly tests our server. The `-X` option is used to set the request's method, and `-d` is used to include a request body.

```
$ curl http://localhost:8000/file.txt
File not found
$ curl -X PUT -d hello http://localhost:8000/file.txt
$ curl http://localhost:8000/file.txt
hello
$ curl -X DELETE http://localhost:8000/file.txt
$ curl http://localhost:8000/file.txt
File not found
```

The first request for `file.txt` fails since the file does not exist yet. The `PUT` request creates the file, and behold, the next request successfully retrieves it. After deleting it with a `DELETE` request, the file is again missing.

Summary

Node is a nice, small system that lets us run JavaScript in a nonbrowser context. It was originally designed for network tasks to play the role of a *node* in a network. But it lends itself to all kinds of scripting tasks, and if writing JavaScript is something you enjoy, automating tasks with Node works well.

NPM provides packages for everything you can think of (and quite a few things you'd probably never think of), and it allows you to fetch and install those packages with the `npm` program. Node comes with a number of built-in modules, including the `fs` module for working with the file system and the `http` module for running HTTP servers and making HTTP requests.

All input and output in Node is done asynchronously, unless you explicitly use a synchronous variant of a function, such as `readFileSync`. When calling such asynchronous functions, you provide callback functions, and Node will call them with an error value and (if available) a result when it is ready.

Exercises

Search tool

On Unix systems, there is a command line tool called `grep` that can be used to quickly search files for a regular expression.

Write a Node script that can be run from the command line and acts somewhat like `grep`. It treats its first command line argument as a regular expression and treats any further arguments as files to search. It should output the names of any file whose content matches the regular expression.

When that works, extend it so that when one of the arguments is a directory, it searches through all files in that directory and its subdirectories.

Use asynchronous or synchronous file system functions as you see fit. Setting things up so that multiple asynchronous actions are requested at the same time might speed things up a little, but not a huge amount, since most file systems can read only one thing at a time.

Directory creation

Though the `DELETE` method in our file server is able to delete directories (using `rmdir`), the server currently does not provide any way to *create* a directory.

Add support for the `MKCOL` method ("make collection"), which should create a directory by calling `mkdir` from the `fs` module. `MKCOL` is not a widely used HTTP method, but it does exist for this same purpose in the *WebDAV* standard, which specifies a set of conventions on top of HTTP that make it suitable for creating documents.

A public space on the web

Since the file server serves up any kind of file and even includes the right `Content-Type` header, you can use it to serve a website. Since it allows everybody to delete and replace files, it would be an interesting kind of website: one that can be modified, improved, and vandalized by everybody who takes the time to create the right HTTP request.

Write a basic HTML page that includes a simple JavaScript file. Put the files in a directory served by the file server and open them in your browser.

Next, as an advanced exercise or even a weekend project, combine all the knowledge you gained from this book to build a more user-friendly interface for modifying the website—from *inside* the website.

Use an HTML form to edit the content of the files that make up the website, allowing the user to update them on the server by using HTTP requests, as

described in Chapter 18.

Start by making only a single file editable. Then make it so that the user can select which file to edit. Use the fact that our file server returns lists of files when reading a directory.

Don't work directly in the code exposed by the file server since if you make a mistake, you are likely to damage the files there. Instead, keep your work outside of the publicly accessible directory and copy it there when testing.