Breakout

The Game

In the classic game of breakout, rows of bricks line the top of the screen, and a ball bounces around breaking the bricks. If the ball reaches the bottom of the screen, the player loses. To prevent that, the user must move a small paddle back and forth near the bottom of the screen. Here is an example of the game in play:

Breakout!

Initially, ten rows of ten bricks each are placed on the screen. The paddle is on the screen, but the ball is not. As soon as the user clicks the mouse, a ball is served and the game begins. The ball starts just below the bottom row of bricks, and is aimed at an angle toward the bottom of the screen.

As the user moves the mouse back and forth, the paddle tracks the mouse back and forth. The ball will bounce off the paddle, the sides and top of the screen, and the bricks. Whenever it hits a brick, the ball bounces as normal, but it also breaks the brick that it hit. The object of the game is to break all of the bricks.

Getting started

I provide the HTML for the board. You must not make any changes to it. All changes must be via JavaScript (including event hooks) in an external file:

- breakout.html

Download breakout.html and save it somewhere. Create a file called breakout.js and put all of your work in that file. Load the HTML file in your browser, and hit the reload button each time you make changes and want to test them.
Enabling strict mode tells the browser to catch many common errors that would otherwise be ignored. Put this line at the very top of your JavaScript file to enable it:

```
"use strict";
```

**The HTML**

The screen starts out as nothing more than a large box with a title at the top. You must use JavaScript to create the bricks, the paddle, and the ball. The file does include an embedded stylesheet that will help with this.

The main box (which marks the boundaries of the playing field) is a `<div>` with the ID `main`. It is styled with `position: relative;`, and the `div`s within it using `position: absolute;`, which makes it act like the boundaries of the screen as far as positioning elements is concerned. For example, when you place a `brick` `div` within `main`, you can specify its position (using `top`, `left`, `right`, `bottom`) and its coordinates will be interpreted relative to the main box.

**How to proceed**

Start by loading the HTML into Chrome and use the developer console to try things out.

**Bricks**

The first thing to try is putting elements on the screen. You can do this by adding `div` elements inside the `main` element. Look up the following DOM functions and apply them:

- `document.getElementById`
- `document.querySelector`
- `document.querySelectorAll`
- `document.createElement`
- `elt.classList.add`
- `elt.classList.remove`
- `elt.appendChild`

Where `elt` is any DOM element.

Try adding a `div` to `main` with the three classes `brick`, `row3`, and `col7`. The stylesheet will cause this to be displayed as a brick in a specific position on the board.

Next, try creating all 100 bricks on the screen by inserting a series of `div` tags. Use the class names “row0” through “row9” and “col0” through “col9” to position and color the bricks. This is one of the first things your game must do to setup the board.

When one of the bricks is broken, it needs to be hidden. You could remove it, but simply hiding it works just as well. Select a brick and add the class `broken` to it. When you do, the brick (or bricks) you have selected should disappear from the display. Try selecting a whole row or a whole column of bricks using a single selector, and then write a loop to break them all. Try doing the same for all of the bricks on the screen.

To restore a brick, remove the `broken` class.

**The paddle**

To place the paddle on the screen, add a `div` with ID `paddle` to `main`. You can set an element’s ID by assigning a string to the ‘id’ property of the object. Try this in the console.

The stylesheet will give the paddle a shape and a vertical position, but you must set its horizontal position. Do this by setting the “left” property of the paddle’s style to give the position of the left side of the paddle. To center it, use this, assuming ‘paddle’ is a reference to the paddle `div`:

```
paddle.style.left = (800 - 140) / 2;
```

The `main` box is 800 pixels wide, and the paddle is 140 pixels wide, so this expression centers it. Try moving it to different positions by changing its left (or right) position and verify that it moves.

To make the paddle track the mouse, you need to capture events associated with mouse movements. To do this, set an event handler for the `mousemove` event of the `body` (accessible as `document.body`). The handler must
accept a single argument, which is an object containing information about the event. The event object will have several fields indicating the current position of the mouse on the screen. The most reliable attribute to look at for our purposes is called \texttt{clientX}, and it tells the position of the mouse relative to the client viewing area. So if the event handler function receives an object \(e\) as its argument, \(e.clientX\) will be the horizontal position of the mouse when the event was triggered.

You may also like to use the \texttt{offsetLeft} attribute of the \texttt{main div} element, which will tell you where the left edge of the box is relative to the viewing area. Those two values should be enough to tell you how far the mouse is from the left edge of the \texttt{main} box. Set the \texttt{style.left} property for the \texttt{paddle} element (as above) and the paddle will move to follow the mouse (at least horizontally).

Of course, you will not want the paddle to leave the screen, so you should check if its new position is too far to either side, and adjust it accordingly. You may also wish to consider the width of the paddle, and set it so that the middle of the paddle tracks the mouse cursor.

**The ball**

Placing the ball on the screen is similar to the paddle: create a \texttt{div} with ID \texttt{ball} inside the \texttt{main div}. Set its \texttt{style.left} and \texttt{style.top} attributes to control its position. The styling for the ball uses rounded corners to make it look like a round ball. If you want to use an image instead, set it as the background of the ball \texttt{div}.

The ball should only appear on the page when the user clicks, and from that point on it should move without user interaction. Make sure the function that receives the click event has a name, say \texttt{createBall}, then inside the handler disable the handler so that the user cannot create multiple balls by clicking multiple times. Disabling an event handler is similar to enabling it, but uses the \texttt{removeEventListener} method instead of \texttt{addEventListener}.

To achieve motion, you must hook into an event based on a timer. The \texttt{window} object has a method called \texttt{setInterval} that takes two arguments: a JavaScript function to be called, and the number of milliseconds to wait between calls to it:

\begin{verbatim}
var intervalID = window.setInterval(moveBall, 20);
\end{verbatim}

20 milliseconds is a good number to use. That will update the ball 50 times per second, which is the right frequency to fool the eye into thinking the motion is continuous. Unfortunately, JavaScript only guarantees it will wait at least as long as you specify, but it may wait longer.

Note that \texttt{setInterval} returns an ID. Save this ID and you can use it later to cancel the scheduled event (when the game ends). To do so, use:

\begin{verbatim}
window.clearInterval(intervalID);
\end{verbatim}

Start by making the ball bounce around without considering collisions with other objects on the screen. This code will give a reasonable initial velocity:

\begin{verbatim}
// ball velocity
var msPerFrame = 20;
var secondsPerFrame = msPerFrame / 1000;

// This sets horizontal rate to 200-600 pixels/second
var vx = secondsPerFrame * (Math.floor(Math.random() * 400) + 200);
if (Math.random() < 0.5) vx = -vx;

// This sets vertical rate to 500 pixels/second
var vy = secondsPerFrame * 500;
\end{verbatim}

\(vx\) and \(vy\) give the number of pixels per frame in the horizontal and vertical directions. Add it to the old position to get the new horizontal position. Likewise for the new vertical position. Start the ball at \((385, 300)\) which is (almost) the center of the screen right below the bricks.

Your \texttt{moveBall()} function should update the coordinates of the ball, and then check if it has hit a wall. For now, treat all walls the same way. If the ball has fallen off a side of the screen, move it back onto the screen and reverse its velocity. For example, if it is off the left edge, change its \(x\) position back to zero and negate the \(x\) velocity. Be sure to consider the size of the ball (30 pixels in diameter) in all of your computations.

Once you have computed a new location, update the element's style properties to place the ball in its new
position.

Once you have done this, you should be able to click on the screen and watch the ball bounce around endlessly inside the window.

**Collisions**

Now that you can move the paddle and watch the ball bounce around, you need to make objects seem solid. To simplify things, we will assume that the ball is a square, and detect collisions only at its four corners. You already track the position of the top left corner, so you can easily compute the positions of the other four corners. The top right is $(x+30,y)$, the bottom right is $(x+30,y+30)$, and the bottom left is $(x,y+30)$.

To check if a point is inside a brick, start with the following:

```javascript
var brickx = 80; // dimensions of a brick
var bricky = 20;
var row = Math.floor((y - 100) / bricky);
var col = Math.floor(x / brickx);
if (row < 0 || row >= 10 || col < 0 || col >= 10)
  // not in the right area
if (((x+2) % brickx < 4 || (y+2) % bricky < 4)
  // not quite in the brick--it's in the white border around a brick
  // otherwise, row and column give the brick number
```

This code will help you determine if a given $x$ and $y$ are inside a brick, and you must also check if the brick at the given position has been broken or not. You can do this by checking if it has the `[broken]` class set, or you can use another object to track which bricks have been broken. For the latter approach, create an object, and set a property on the object every time you break a brick, for example:

```javascript
var key = String(col) + ',' + String(row);
if (key in brokenBricks) {
  // brick is already broken, pretend it's not there
} else {
  // brick has not been broken yet
  brokenBricks[key] = true; // now it has
  // make brick invisible
  // reverse y direction of ball
}
```

Every time you break a brick, you should flip the vertical direction of the ball. Write this code as a function, and call it for all four corners surrounding the ball. Make sure that you only flip the direction once, even if the ball breaks multiple bricks at once.

Once you do that, you should be able to watch the ball bounce around, breaking bricks as it goes, until eventually the entire board is cleared.

Next, you need to make the ball bounce off the paddle. Checking the corners off the ball doesn’t work as well here, because the user can move the paddle quickly (while bricks just sit there). Instead, do the following: if the left edge of the ball is to the right of the right edge of the paddle, there is no collision. If the right edge of the ball is left of the left edge of the paddle, there is no collision. If the bottom of the ball is above the top of the paddle, no collision. If the top of the ball is below the bottom of the paddle, no collision. Otherwise, there is a collision. In this case, set the direction of the ball to up (use $vy = -Math.abs(vy);$). Note that simply reversing the direction of the ball may not work, because the paddle and the ball could overlap after a rapid paddle movement, causing the ball to bounce multiple times before it clears the paddle.

With paddle bouncing implemented, the only thing left is to terminate the game. Instead of having the ball bounce off the bottom of the board, have it go away if it hits the bottom, and declare the game a loss (an alert will do the trick). If the last brick has been broken, declare the game a win (the easiest way is to count the number of bricks broken). In either case, reset the board and start over (when the user clicks to serve another ball).

**Extras**

There is a lot to do, but you can see progress quite regularly, so you will always know if you are on the right track. Start early and take it in steps. Don’t do anything extra until you have all the basic functionality working.

You are only required to implement the basic version of the game. To make it nicer to play, I recommend a few
additions:

- **Improve the user control over bounces:** The program gets rather boring if the only thing the player has to do is hit the ball. It is far more interesting if the player can control the ball by hitting it at different parts of the paddle. The way the old arcade game worked was that the ball would bounce in both the $x$ and $y$ directions if you hit it on the edge of the paddle from which the ball was coming.

- **Add in the “kicker.”** The arcade version of Breakout lured you in by starting off slowly. But, as soon as you thought you were getting the hang of things, the program sped up, making life just a bit more exciting. Implement this feature by doubling the horizontal velocity of the ball on the seventh time it hits the paddle, figuring that that’s around the time that the player is growing complacent.

- **Keep score:** You could easily keep score, generating points for each brick. In the arcade game, bricks were more valuable higher up in the array, so that you got more points for red bricks than cyan bricks. You could display the score underneath the main board, since it won’t get in the way there.

- **Improve collision detection:** Sometimes the ball “hits” a brick without actually making contact. You could improve the collision detection by developing a more sophisticated test to check if the ball has hit a brick.

- **Add levels:** Instead of simply starting over after a win, make the ball move faster, make the paddle shorter, and/or make the points worth more as the user gets through more levels.

**What to turn in**

Include comments throughout your code documenting what you have done. Set your solution running on a server somewhere. This can be the public_html directory of your CIT account. Submit a link to a playable version of your game to Canvas.