<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8">
<title>JS Play, v2_1</title>
<style type="text/css">
  table { border: double; width: 60%; border-collapse: collapse }
  thead { padding: 15px; text-align: center; }
  /* adding some invisible/visible interaction rule(s) */
  .interaction { visibility: hidden; }
  /* improve table style? */
  th { background-color: lightgreen; }
  td { background-color: yellow; }
</style>
</head>
<body>
<h1>Playing with JavaScript</h1>
<!--
The following table might just as well be
generated by the Script. But, because we already
know a lot about what it contains, it's easier to
specify its main features here and delegate the actual
computation and population of its cells to the script.
Pay attention to this: it's a common "pattern."
-->
<table id="interaction">
  <!--
The "caption" element will be generated by the script because
its contents depend upon what the user typed.
-->
  <caption id="caption"></caption>
  <thead>
    <!--
    Because we already know what's in this table, it's okay
    to provide these details in the document, leaving the
    heavy lifting to the script.
    -->
    <tr>
      <th>Sum</th> <th>Absolute Difference</th> <th>Product</th> <th>Quotient</th>
    </tr>
  </thead>
  <!--
  As a matter of practice: whenever I use a "thead" I also use the "tbody"
  (and sometimes, if relevant) a "tfoot" element.
  -->
  <tbody>
    <!--
    Pay attention to the use of "ids" here.
    This allows the script to easily provide the correct
    results to the corresponding cell(s).
    -->
    <tr id="sum"></tr>
    <tr id="difference"></tr>
    <tr id="product"></tr>
    <tr id="quotient"></tr>
  </tbody>
</table>
<!--
Notice where I placed the Script in this buffer. Because the
script references objects in the DOM, I need to ensure that those
objects have "living" references when I attempt to dereference (use)
then in this script:
-->
<script type="text/javascript">
/**
*/
</script>
</body>
</html>
* Used by the various prompting routines to ensure that users do not exceed a range.

```cpp
const MAX = 100; // largest integer that user may enter.
const MIN = 0; // smallest integer that the user may enter.

/* The following functions are used to simplify the "main" logic. */

cc oo nn ss tt

MAX = 100;
// largest integer that user may enter.

cc oo nn ss tt

MIN = 0;
// smallest integer that the user may enter.

/* The following functions are used to simplify the "main" logic. */

/**
 * Preconditions: MIN and MAX have been set.
 * Postconditions: an integer >= MIN but <= the MAX is returned.
 * Note: this function will continue prompting the user until these conditions are met. This is an example of a "nag" function.
 */

function promptInt() {
    /* Ask the user once ... if all goes well, then the while statement that follows is never executed. */
    var input = window.prompt( "Enter an integer greater than or equal to " + MIN + " but not greater than " + MAX + ":");
    /* Note: we distinguish between several "kinds" of iterative statements in programming languages.
    * Bounded Iterators are constructions where the number of times that an iteration is performed is "known ahead of time."
    * Unbounded Iterators: constructions where the number of iterations are unknown at the time the computation begins. Presumably, some condition becomes true or false and that signals the end of the iteration.
    * In the usage below: we use the "while" statement which is an "unbounded" iteration construct. */
    while( input < MIN || input > MAX ) {
        input = window.prompt( "Trying again: please enter an integer greater than or equal to " + MIN + " but not greater than " + MAX + ":");
    }
    return parseInt( input );
}

/**
 * Precondition: Given two integers
 * Postconditions: return the absolute difference between these two integers, meaning that the difference between these two integers as a non-negative integer is returned.
 */

function absDifference( number1, number2 ) {
    if( number1 < number2 ) {
        return number2 - number1;
    } else {
        return number1 - number2;
    }
}

/**
 * Precondition: two integers are given, where the second is NOT zero.
 * Postconditions: the "integer" quotient is returned, which is the floor of the actual quotient.
 * [In this particular application, however, zeros should never appear.
 * The conditional is used to demonstrate how the "alert" method might be used to help debug JavaScript ....] But, consider what might happen if we shared some of these functions with other web-pages ... */

function intQuotient( number1, number2 ) {
    if( number2 === 0 ) {
        window.alert( "Attempted division by zero! ");
        return 0;
    } else {
        return Math.floor( number1 / number2 );
    }
}

/* end of private functions block */
/* Main logic: program execution starts here ... */

/* Note how the use of a function simplifies a tedious task here.

[Think about how we might enhance this interaction in the future.] */

var input1 = promptInt();
var input2 = promptInt();

/** Perform the computations. Think about the following: What are our "options"
* that we wanted ONLY non-negative integers, but the user entered negative
* integers?
* Take a look at the Chapter on Forms (Chapter ??) in your textbook and keep
* these kinds of questions in mind. Scripts are often used to "validate" forms
* input! */

var sum = input1 + input2;
/** Note again: we use a function to "hide" any complicated processing. In
* essence, we define a new "verb" and use it. */

var difference = absDifference( input1, input2 );
var product = input1 * input2;

/** The intQuotient does something unorthodox: if the
* second number is a zero, it complains and returns zero.
* BUT, this should NEVER happen ... 
* Do you see why? (Hint: look at the promptInt function, above.) */

var quotient = intQuotient( input1, input2 );
var remainder = input1 % input2;

/** Please observe the following "pattern." Get comfortable with it;
* you will use it throughout the remainder of the semester.
*
* Reflect on who "owns" these "elements." Clearly, these are objects
* that reside in the "document."
*
* For those "forward thinking" readers: what would be the result of retrieving
* a reference to an HTML element that was associated with a "class" instead of
* an "id"? (Hint: what is the difference between things that are marked with
* "id"s and those marked with "class"es?) */

document.getElementById( "sum" ).innerHTML=sum;
document.getElementById( "difference" ).innerHTML=difference;
document.getElementById( "product" ).innerHTML=product;
if( remainder === 0 ) {
  document.getElementById( "quotient" ).innerHTML=quotient;
} else { document.getElementById("quotient").innerHTML = quotient + "", with remainder: " + remainder; }
document.getElementById( "caption" ).innerHTML="Given " + input1 + " and " + input2 + ": computed the following...";

/** Turn on the visibility for objects of the class "interaction"
*/

var interactionElements = document.getElementsByClassName( "interaction" );
for( index=0; index < interactionElements.length; index++ ) {
  interactionElements[ index ].visibility=":visible";
}

</script>
</body>
</html>