EECS1012

Net-centric Introduction to Computing

Lecture

JavaScript and Forms

First - some additional stuff

Non-Boolean conditions

Console log

Non-boolean conditions

- When we looked at flow control statements, we considered Boolean expressions

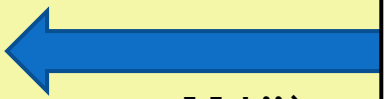
```
function myFunction()
{
  var num1 = 5;

  if (num1 < 10) {
       alert("Too small!");
  }
}
```

Is 5 less than 10? YES
This expression is true, so we execute the alert!

Non-boolean conditions

- What happens in this case?

```
function myFunction()
{
  var num1 = 5;

  if (num1) {
       alert("Too small!");
  }
}
```



Look! This means:
if (5) {
  …
}

Is this true or false?

Turns out, it is true!

Non-boolean expressions

- JavaScript considers **all expressions to be true, except the following:**

""	(empty string is considered false)
null	(null is considered false)
0	(zero is considered false, but not "0")
false	(of course-false is false)
NaN	(this happens in some math, e.g. 0/0;)*

* This means “Not a Number”.

Non-boolean conditions

- What happens in this case?

```
function myFunction()
{
  var num1 = "";

  if (num1) {
       alert("Too small!");
  }
}
```

This is false.
This will not output, because empty string is considered false.

Common condition mistake


```
function myFunction()
{
  var num1 = 3;

  if (num1=5) {
       alert("Too small!");
  }
}
```

This will evaluate to **true**! Why?

Because we used a single =, instead of a double ==

JavaScript will evaluate this as:

```
if (num1=5) {
       alert(“Too small!”);
  }
```

(1) First, perform the assignment 5 to num1.

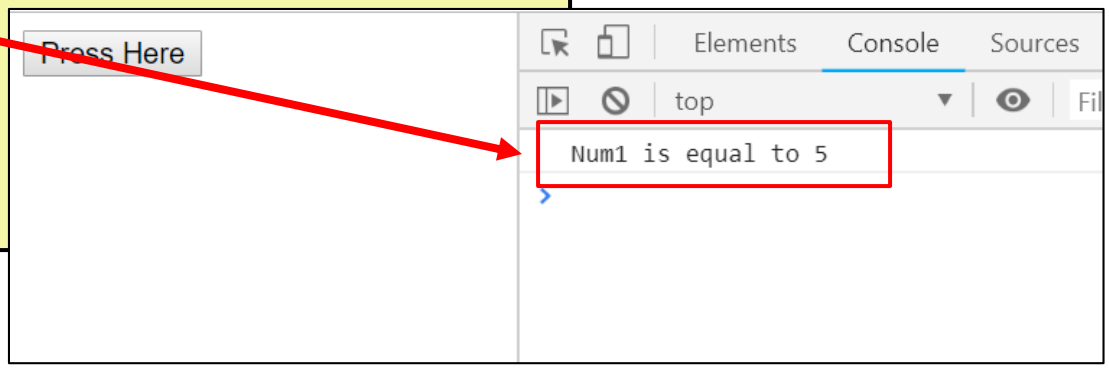
```
if (num1) {
       alert(“Too small!”);
  }
```

(2) next, evaluate if (num1) … which is true.

Console log

- We have seen how to use "alert" to output messages
- For debugging, you can also use the console object

```
function myFunction()
{
  var num1 = 5;
  if (num1==5)
  {
       console.log("Num1 is equal to" + num1);
  }
}
```

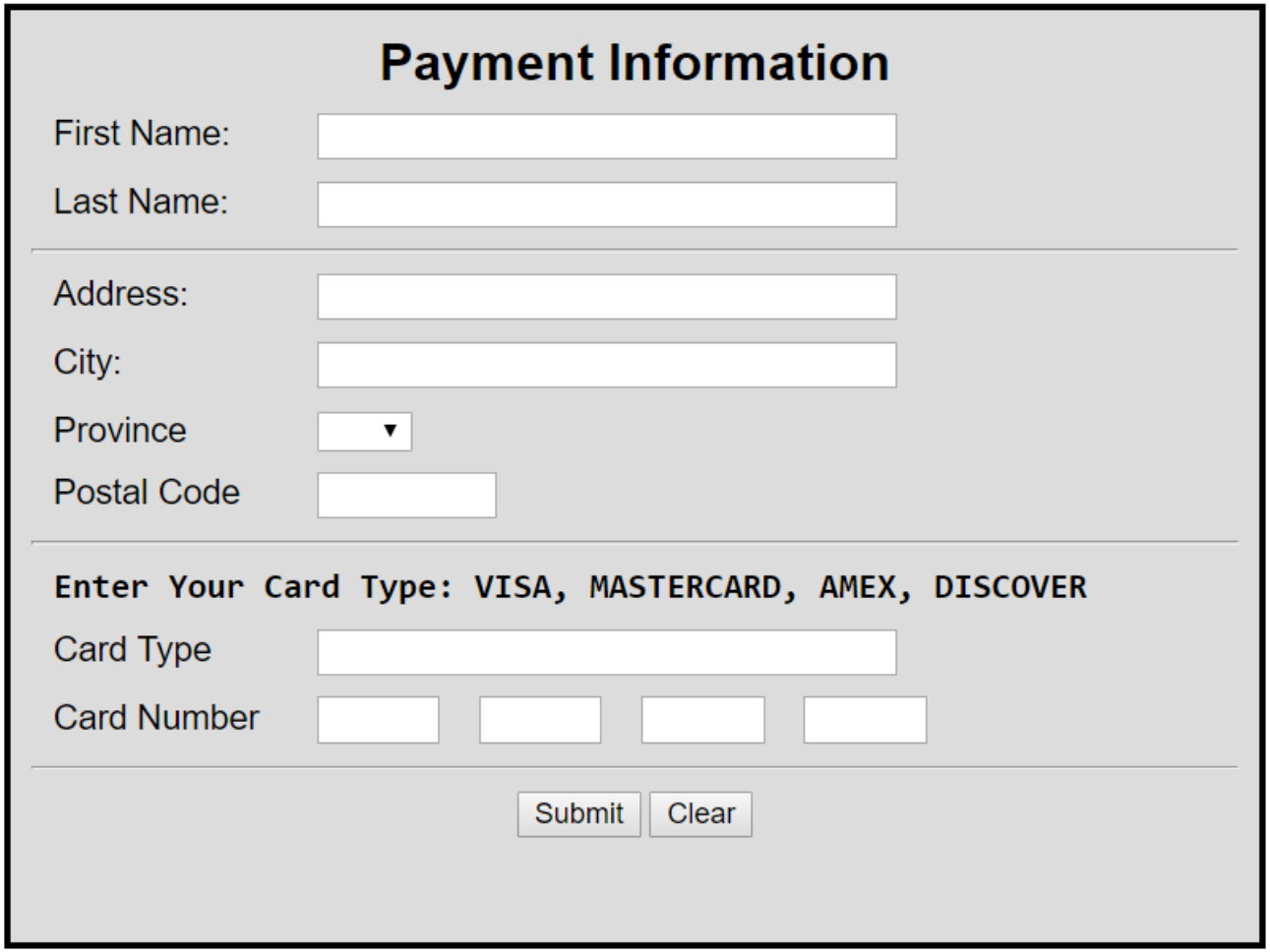


OK. . back to Form +JS

In this lecture

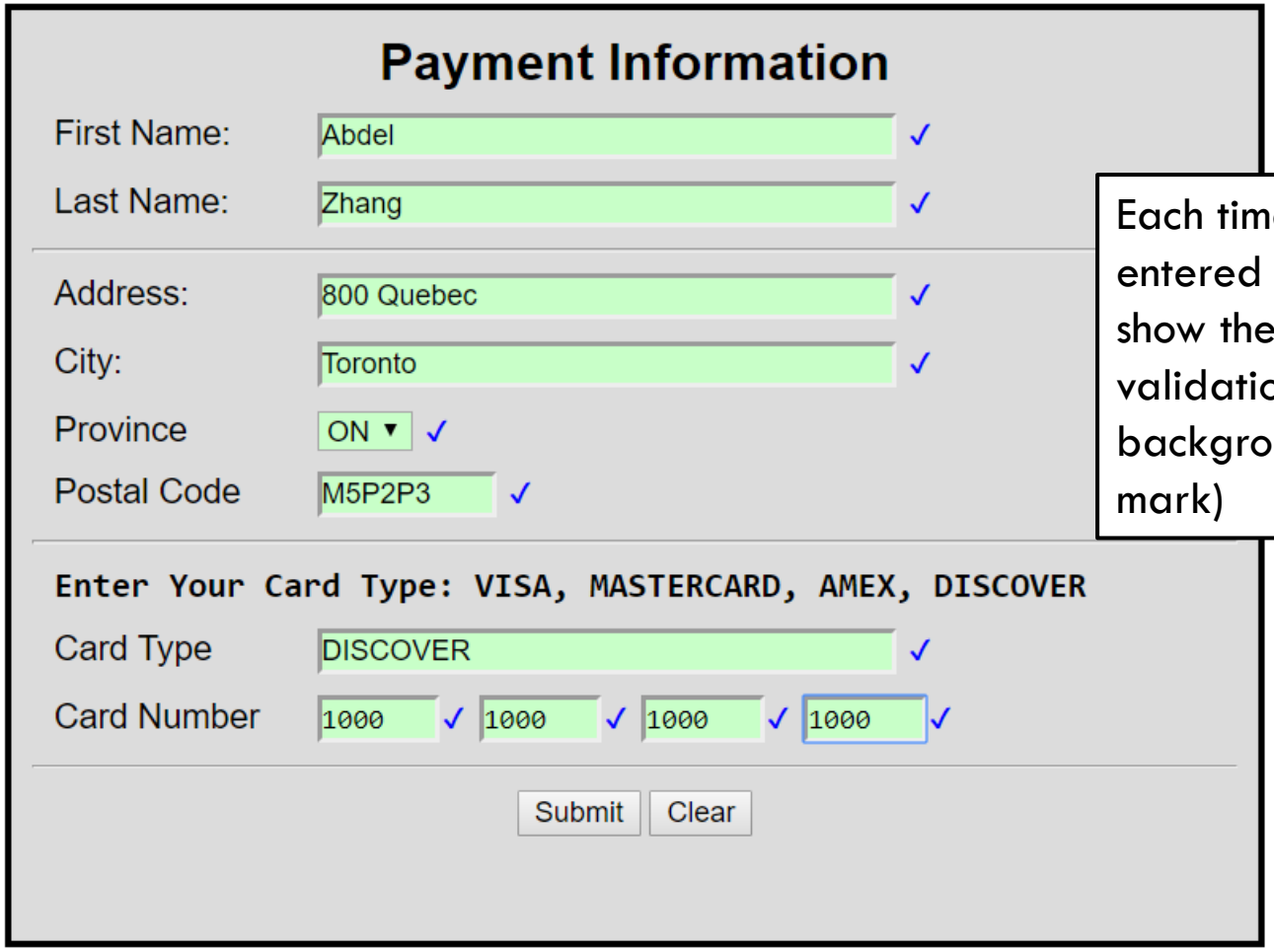
- We will use JavaScript to validate page using HTML form elements
- This will combine our knowledge of HTML Forms and JavaScript
- We will need to learn a few more concepts
  - Regular Expressions
  - Searching Arrays
  - Some additional string processing

Our starting point without JS



A webpage with HTML form elements.

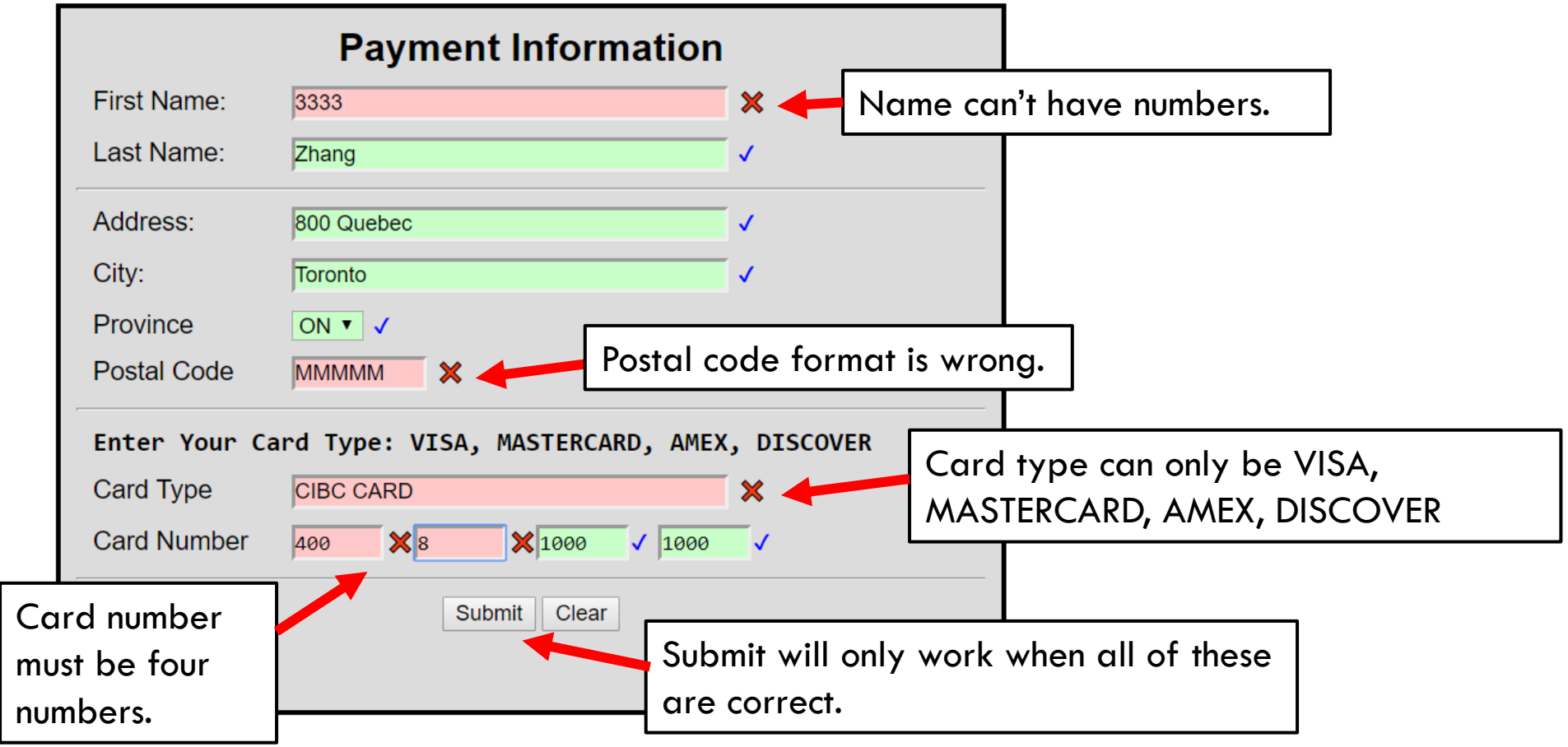
Behavior of our HTML page



Each time an item is entered correctly, we show the user instant validation (green background + check mark)

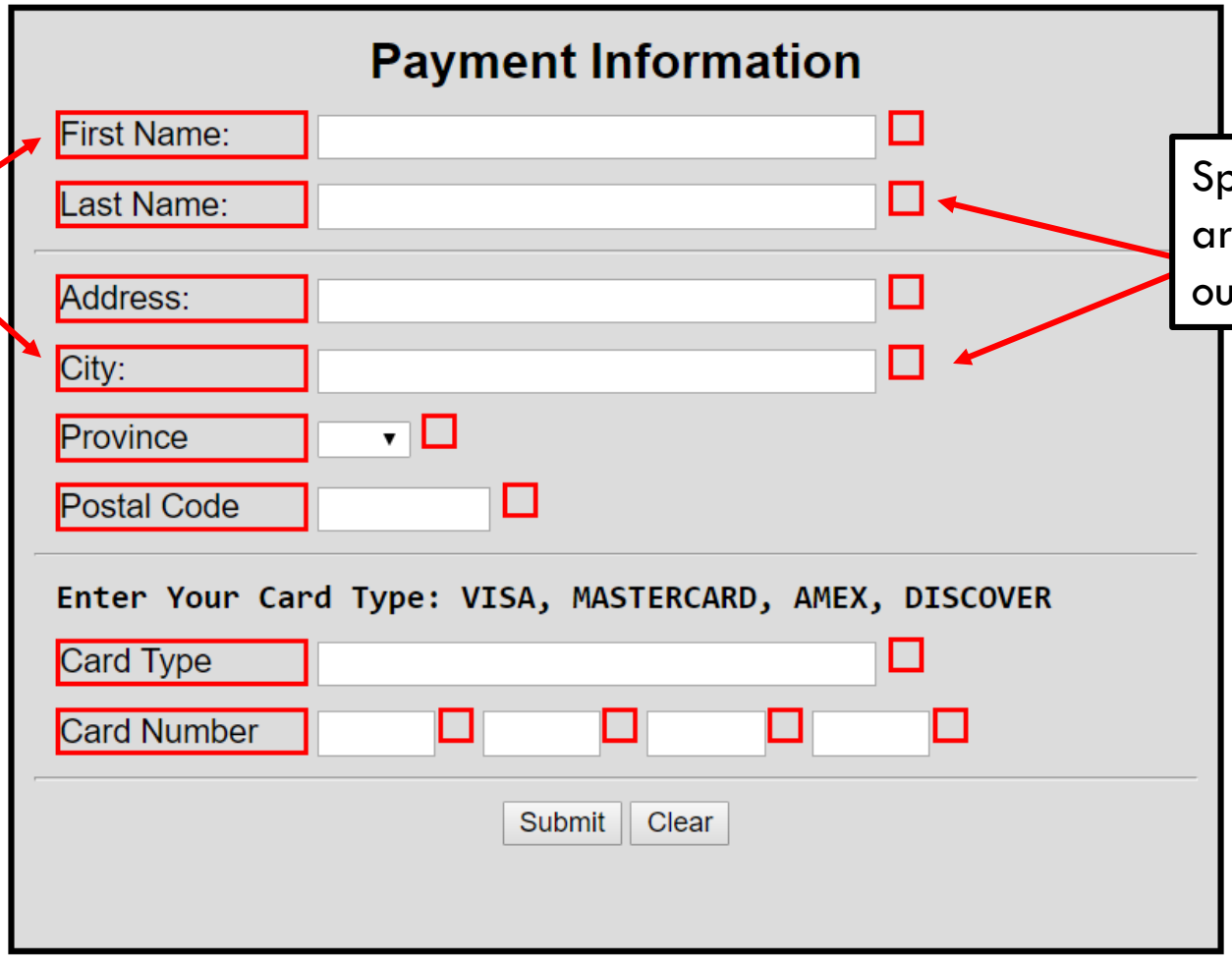
Behavior of our HTML

Errors are shown when input is incorrect.



Our starting point

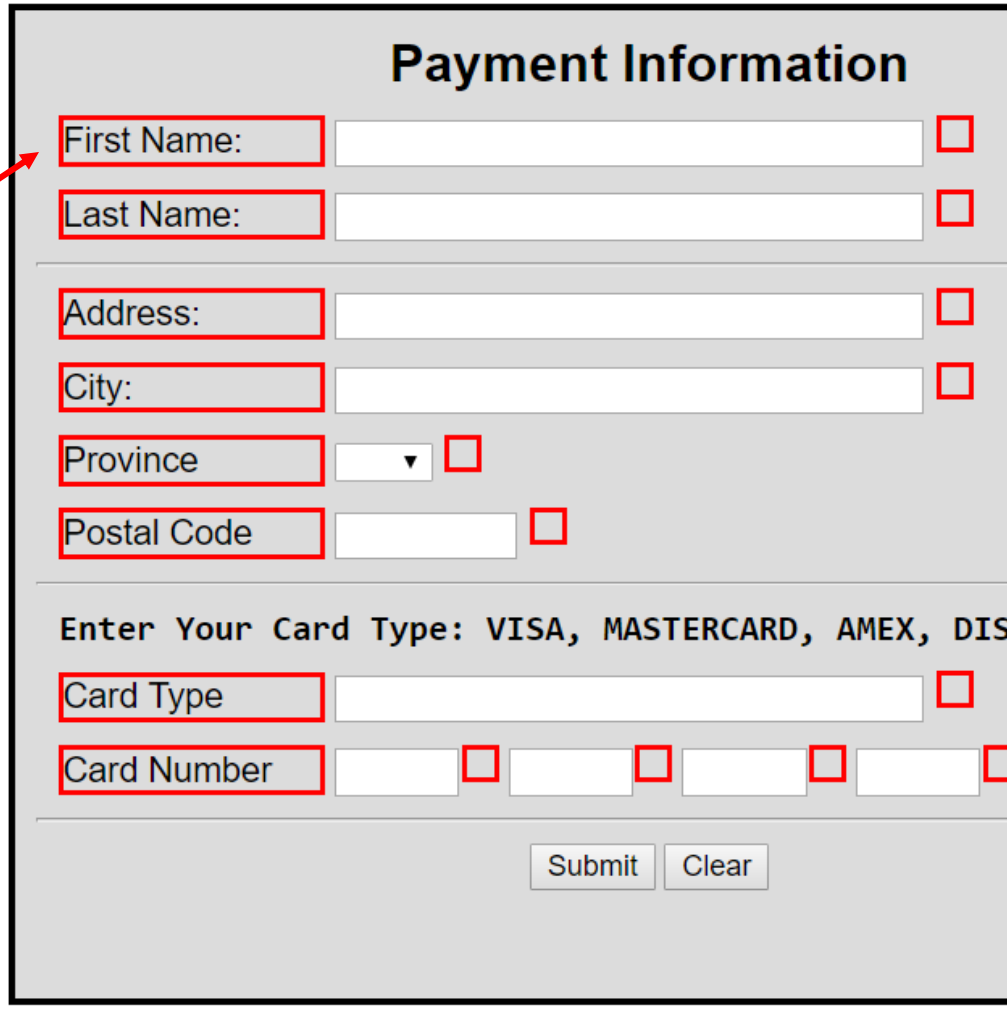
Span element is used to provide consistent lengths.



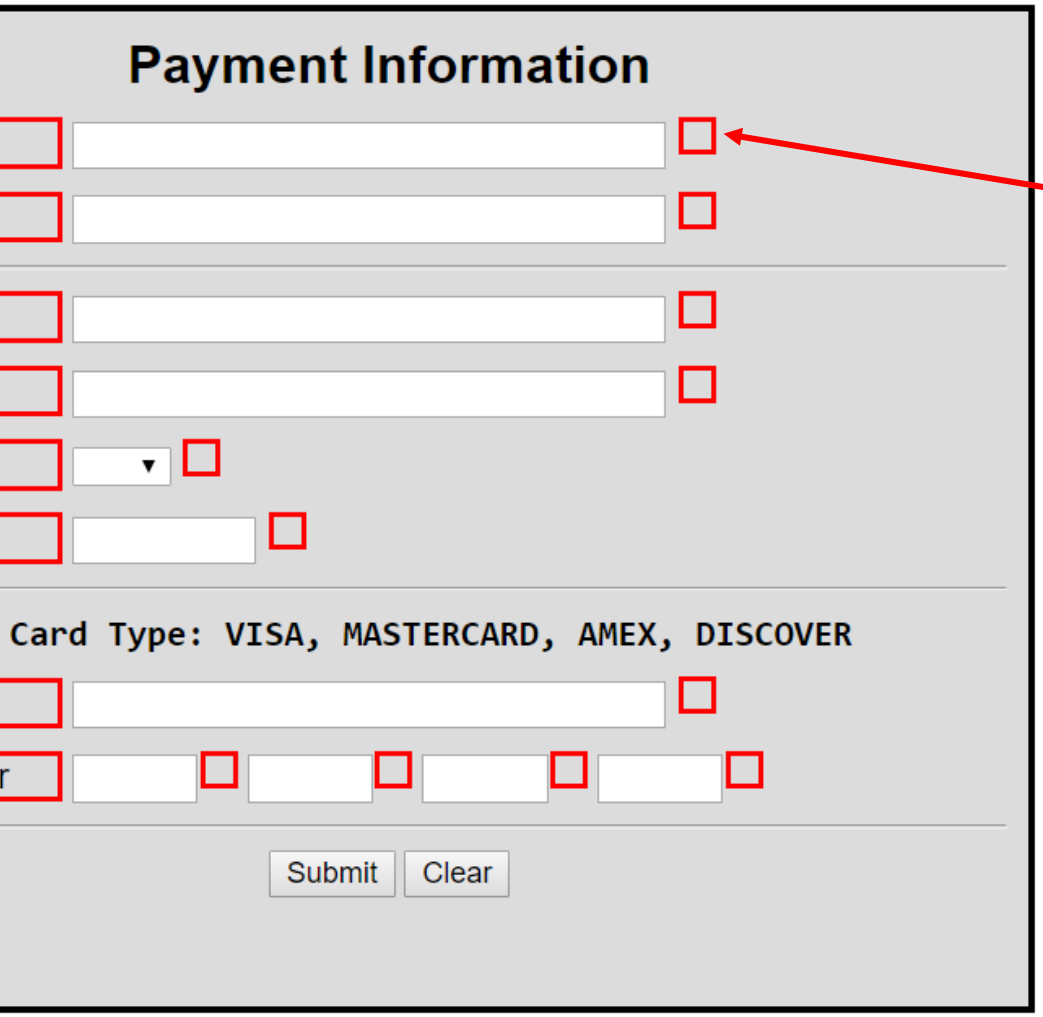
Span elements are used for our ✓ or ✗.

Our starting point


```
.fieldName {
   display: inline-block;
   float: left;
   width: 20%;
   padding-right: 10px;
   font-family: sans-serif;
}
```



Our starting point




```
.message {
  font-family: monospace;
  color: blue;
  margin-left: 1px;
  width: 1em;
  height: 1em;
  display: inline-block;
}
```

CSS inline-block

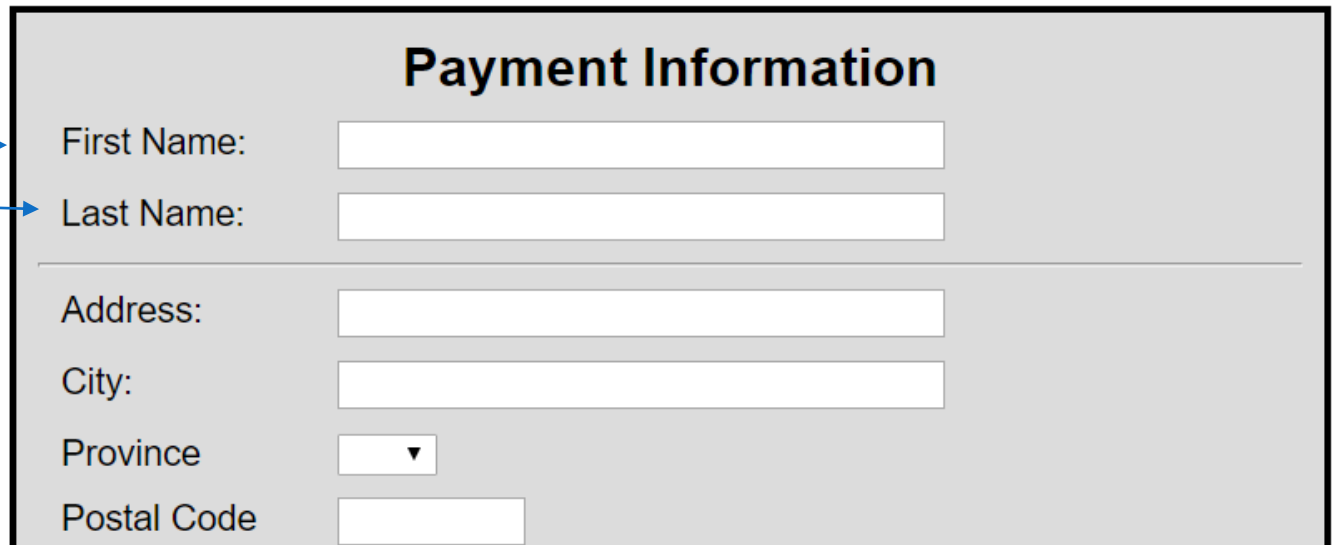
- <spans> are inline elements
- We have changed them to inline-block

```
display: inline-block;
```

- This allows us to specific a fixed width and height, but doesn't cause a line break like a block element would
- See: https://www.w3schools.com/css/css_inline-block.asp

Adding/Removing classes


```
<p> <span class="fieldName">First Name: </span>
<input type="text" id="FirstName" name="FirstName" class="input">
<span class="message"></span></p>

<!-- Last Name input -->
<p> <span class="fieldName"> Last Name: </span>
<input id="LastName" type="text" name="LastName" class="input">
<span class="message"></span></p>
```



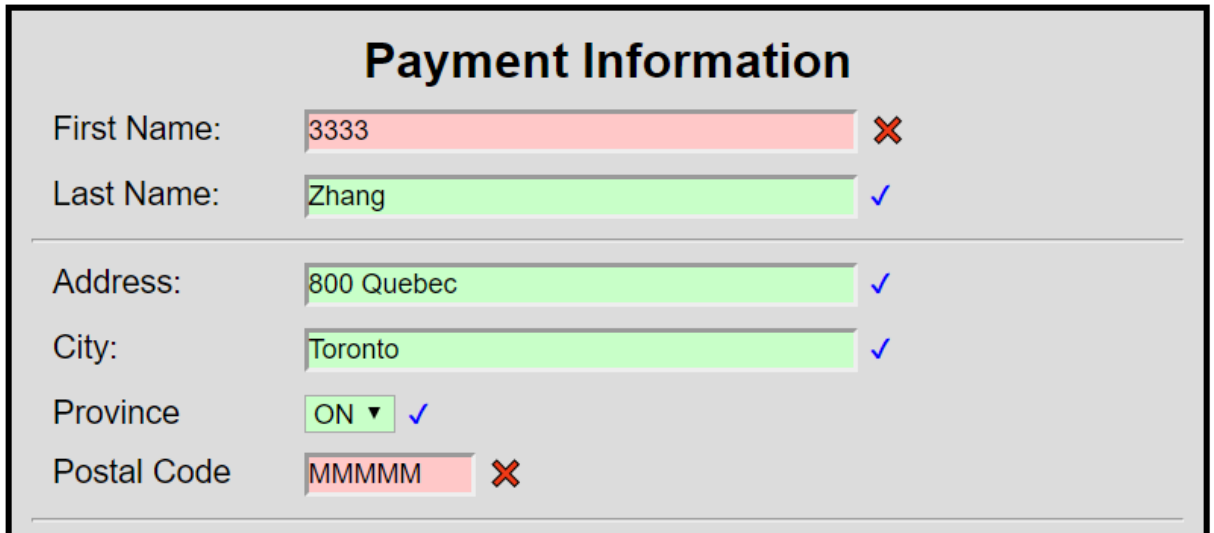
Adding/Removing classes


```
.successful {    background-color: rgb(200, 255, 200);  }
.unsuccessful {    background-color: rgb(255, 200, 200); }
```

```
$("FirstName").addClassName("successful");
$("LastName").addClassName("unsuccessful");
```

We can dynamically add and remove a class for an element.

```
$("LastName").addRemoveName("unsuccessful");
```



How about the ✓ and ✗?

- These are special Unicode characters

  https://www.w3schools.com/charsets/ref_utf_dingbats.asp

- To insert them in HTML

✔ ✓

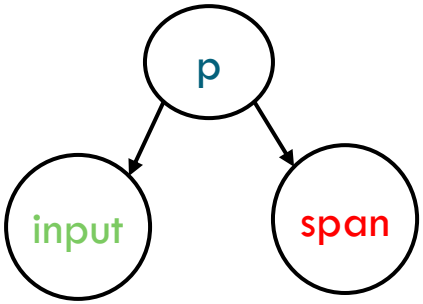
✖ ✗

If we want to add a ✓ or ✗


```
<p> <span class="fieldName">First Name: </span>
 <input type="text" id="FirstName" name="FirstName" class="input">
 <span class="message"></span> </p>

 <!-- Last Name input -->
 <p> <span class="fieldName"> Last Name: </span>
 <input id="LastName" type="text" name="LastName" class="input">
 <span class="message"></span> </p>
```

There is no ID for our span! Only IDs for our inputs.

But look at the DOM tree.



The span is what?

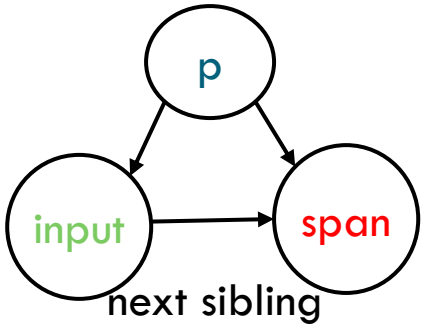
It is the "next sibling" of the input!

So, we can always do the following


```
<p> <span class="fieldName">First Name: </span>
 <input type="text" id="FirstName" name="FirstName" class="input">
 <span class="message"></span> </p>
```


```
$("FirstName").addClassName("unsuccessful");
$("FirstName").nextElementSibling.innerHTML = "&#10060;";
```

This code will make the input box have the background color we want.
Then the code will add the X in the span next to our input element.



The span is what?
It is the "**next sibling**" of the input!

Recap

- So, we have seen what our HTML page looks like
- We know our CSS
- We have seen how to modify the HTML
- Now we just need to determine how to validate our input fields

Regular expressions

Sometimes you will see this referred to as "regex"

Introducing regular expressions

- A **regular expression** is a text string that defines a character pattern
- One use of regular expressions is **pattern-matching,** in which a text string is tested to see whether it matches the pattern defined by a regular expression

Creating a regular expression

- **Creating a regular expression**
  - You create a regular expression in JavaScript using the command

```
var re = /pattern/;
```

  - This syntax for creating regular expressions is sometimes referred to as a **regular expression literal**
  - **We can use the regular expression variable**

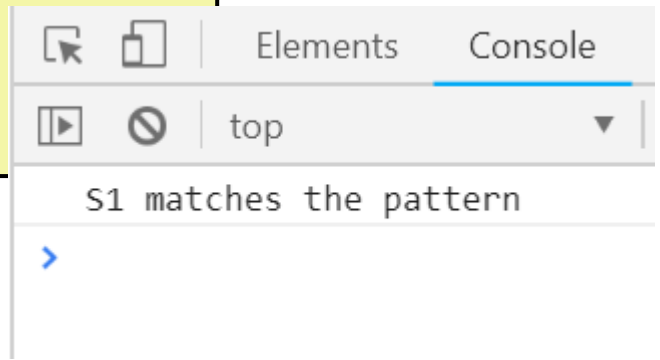
```
re.test("some string")
```

The method test(..) return true if the string ***contains*** the pattern

Example


```
function myFunction()
{
  var re = /ABC/;  /* Pattern ABC */
  var s1 = "ABC";  /* a string */
  var s2 = "DBC";  /* a string */

  if (re.test(s1)) {
        console.log("S1 matches the pattern");
  }
  if (re.test(s2)) {
        console.log("S2 matches the pattern");
  }

}
```

This is true.



String "ABC" matches pattern /ABC/

String "DBC" does not match pattern /ABC/

Power of regex

- A string comparison has to be exact
- A regex finds the pattern

```
var re = /ABC/;  /* Pattern ABC */
var s1 = "ZBBABC01?";  /* a string
```

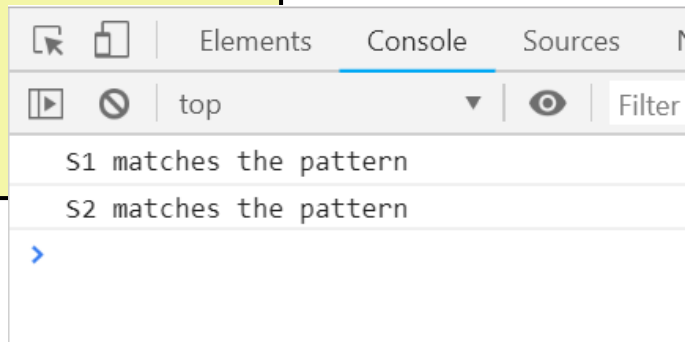
```
"ZBBABC01?"== "ABC"                 FALSE

"ZBBABC01?" contains "ABC"          TRUE
```

Regex finds a pattern

- Test to see if a string contains a pattern

```
function myFunction()
{
  var re = /ABC/;  /* Pattern ABC */
  var s1 = "12ABCZE";  /* a string */
  var s2 = "ABC";  /* a string */

  if (re.test(s1)) {
        console.log("S1 matches the pattern");
  }
  if (re.test(s2)) {
        console.log("S2 matches the pattern");
  }

}
```

This is also true.



String "12**ABC**ZE" matches pattern /ABC/
String "ABC" matches pattern /ABC/

The real power of regex!

- Patterns can include characters (or character ranges) specified between brackets []
- For example:

[ADC]	– matches the letter A or D or C
[9bC]	– matches a 9, b, or C
[a-z]	– matches any lowercase letter
[A-Z]	– matches any uppercase letter
[0-9]	– matches any digit (a number)
[a-zA-Z]	– matches any upper or lower case letter
[a-zA-Z0-9]	– matches any letter or digit
[a-zA-Z0-9\ ]	-- matches any letter, digit or space

More interesting example

- Example

```
var re = /[A-Z][0-9][A-Z]/;
```

This will match a three character pattern that (1) starts with a letter, (2) is follows by number, and (3) is followed by another letter.

Matches to the re pattern:

```
A0Z   Z0Z
C8B   Y1Y
B1E   N3A
F8F   M6P
```

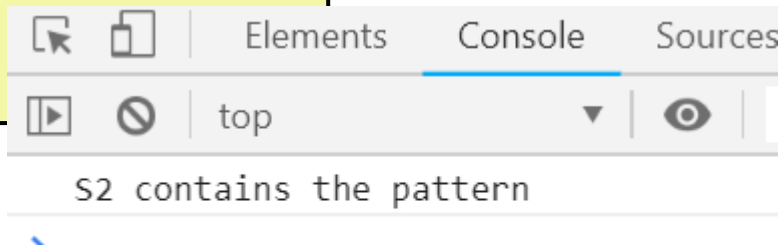
Patterns that don't match the re pattern:

```
a0Z   ZAZ
C83   Y11
BBE   33A
f8f   M_P
```

Another example


```
function myFunction()
{
  var re = /[A-Z][0-9][A-Z]/;  /* Regex */
  var s1 = "ABC";  /* a string */
  var s2 = "D9C";  /* a string */

  if (re.test(s1)) {
        console.log("S1 contains the pattern");
  }
  if (re.test(s2)) {
        console.log("S2 contains the pattern");
  }

}
```

This is true.



regex is letter-digit-letter.
s1="ABC" does not match this pattern.
s2="D9C" **does** match this pattern.

Regular expression *flags*

- **Setting regular expression flags**
  - To make a regular expression not sensitive to case, use the regular expression literal **`/pattern/i`**

```
var re = /[A-Z][0-9][A-Z]/i;
```

Matches to this pattern:

a0Z	Z0Z
C8b	y1Y
b1E	N3A
f8f	n6P

Patterns that don't match:

10Z	ZAZ
C83	Y11
Bbe	33A
38F	M_P

Because of the i flag at the end, case is ignored.

Regex quantifiers: *, +, ?

`*` means 0 or more occurrences

- `/abc*/` matches "ab", "abc", "abcc", "abccc", ...
- `/a*b*c*/` matches "a", "aabc", "bbcc", "aaabbbccc", ...

`+` means 1 or more occurrences

- `/ab+c+/` matches "abc", "abbbcc", "abccc", ...
- `/Goo+gle/` matches "Google", "Gooogle", "Goooogle", ...

`?` means 0 or 1 occurrences

- `/Martina?/` matches "Martin" or "Martina"
- `/A?B?C?/` matches "AC", "BC", "ABC", "A", "B", "C", ...

We can combine with ranges


```
var re = /[A-Z]+[0-9]+[A-Z]+/i;
```

Case doesn't matter

One or more letter

Followed by one or more number

Followed by one or more letter

Matches to this pattern:

```
aa00Zee
ccd1234567a
AbA000000BAB
b123456CC
```

Patterns that don't match:

```
10ZZZZZ0
0000
aa111
1a1
```

Regex anchors ^, $ (part 1)

- The ^ and $ specify something that matches at the beginning (^) or the end of a string ($)

```
var re = /[A-Z][0-9][A-Z]/i;
```

This can match any string that has a Letter-Number-Letter pattern, e.g.;

"0000micC9C" (true) "A11e9Zeee" (true) "b1Z000" (true)

```
var re = /^[A-Z][0-9][A-Z]/i;
```

This can match any string that has a Letter-Number-Letter pattern at the beginning of the string only;

"0000micC9C" **(false)** "Ae9Z" **(false)** "b1Z000" (true)

patterns match, but not at the beginning of the string

pattern matches and is at the beginning

Regex anchors ^, $ (part 2)

- The ^ and $ specify something that matches at the beginning (^) or the end of a string ($)

```
var re = /[A-Z][0-9][A-Z]$/i;
```

This can match any string that has a Letter-Number-Letter pattern at the **end** of the string only;

"0000micC9C" (true) "A11e9Zeee" **(false)** "b1Z000" **(false)**

```
var re = /^[A-Z][0-9][A-Z]$/i;
```

This can match any string that has a Letter-Number-Letter pattern at the beginning and end! So, this can only exactly match a three char string with this pattern.

"0000micC9C" **(false)** "A11e9Zeee" **(false)** "b1Z000" **(false)** "a8C" (true)

It might seem confusing, but this does start and end with the same pattern.

Some more examples

(1) What is a regex for a string that is at least one or more letters?

```
var re = /[A-Z]+/i;
```

(2) What is a regex for a string that is at least one or more numbers?

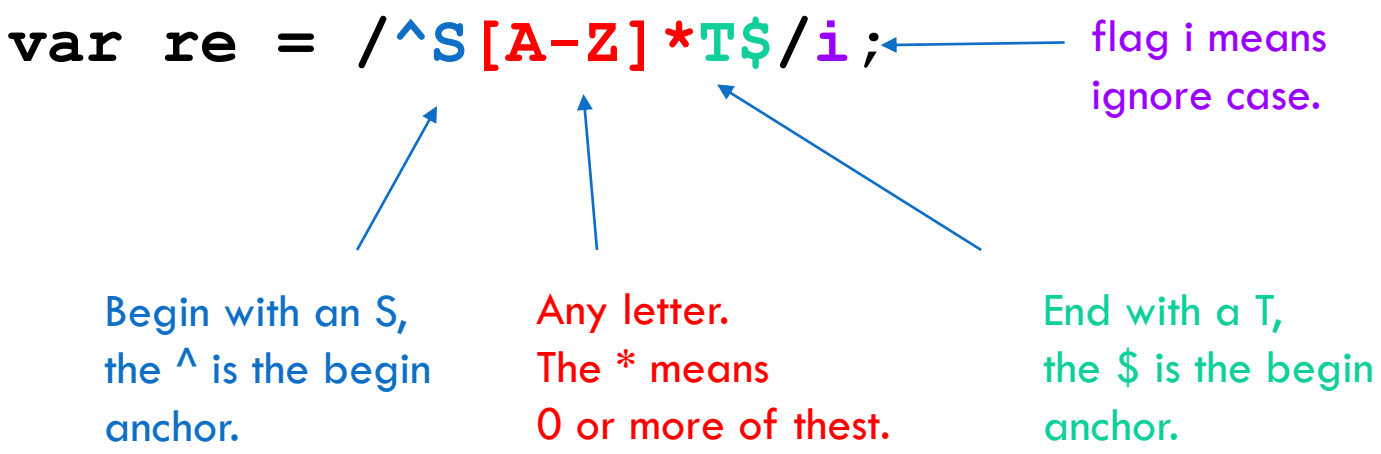
```
var re = /[0-9]+/;
```

(3) What is a regex for a string that starts with one number and then is follows by one or more letters?

```
var re = /^[0-9][A-Z]+/i;
```

What about this example?

- A string that begins with an S and ends with a T, it can have 0 or more letters between the S and T



Canadian postal code example

- Canadian postal code is in the following format
  - Letter-Number-Letter-Number-Letter-Number
  - Example, York's postal code: M3J1P3

- What is the regular expression for this?

```
var re = /^[A-Z][0-9][A-Z][0-9][A-Z][0-9]$/;
```

Letter-Number-Letter-Number-Letter-Number

The ^ means the string has to start with this pattern. The $ means it has to end with this pattern. This combination restricts the string to being exactly 6 characters. Will this expression allow lowercase? If not, how can you make it allow lowercase?

Recap

- There is more to regex, but it is outside the scope of this class and for our purposes
- **Why did we have to learn about regex?**
- Think about how we can restrict out input for our form. . .
- For example, we want to make sure the postal code is correct?
  - Using regular expression is a powerful tool for validating input

Simple String and Array Search

Strings

- Strings are one of the most common data types in JS
- Our form input are always treated as strings
- It is useful to know a few additional string methods

String methods

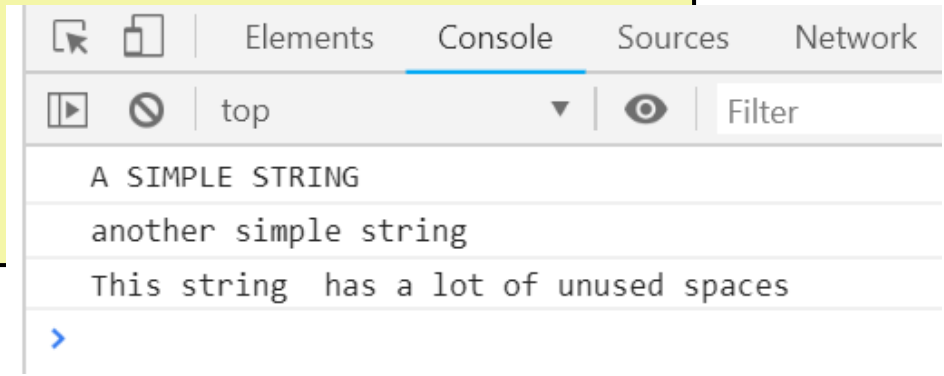
toLowerCase()	Converts a string to lowercase letters
toString()	Returns the value of a String object
toUpperCase()	Converts a string to uppercase letters
trim()	Removes whitespace from both ends of a string

All string methods return a new value. They do not change the original variable. So, if we want to change the original, we have to do the following:

```
var s1 = "SOMEString";
s1 = s1.toLowerCase();   /* this returns to all lowercase and    */
                         /* sets it back to the original variable */
```

Examples


```
function myFunction()
{
  var s1 = "A simple string";
  s1=s1.toUpperCase();                    /* convert all to upper case */

  var s2 = "Another simple STRING";
  s2=s2.toLowerCase();                    /* convert all to lower case */

  var s3 = "    This string  has a lot of unused spaces     ";
  s3=s3.trim();                           /* remove spaces before and after */

 console.log(s1);  /* output the strings */
 console.log(s2);
 console.log(s3);


}
```



Why is this useful


```
<input id="cardtype" type="text">
<button onclick="checkCard();">Click</button>
```

```
function checkCard() {
   var cardType = $("cardType").value;
   cardType = cardType.toUpperCase();
   if ( cardType == "VISA" )
   {
            console.log("VISA was entered");
   }
}
```

We can check the string and not worry about the case. This will always ensure the string is upper case.

Global String method


```
var checkmark = String.fromCharCode(10004);
```

This global String object method fromCharCode(xxx) generates a string using any Unicode value.

This is the same as checkmark="✓";
(It is just hard to type that char, I cut and pasted it)

https://www.w3schools.com/jsref/jsref_fromcharcode.asp

Array includes() method

- Consider an array of strings:

```
var myArray = ["MB", "SK", "QC", "ON"];
```

We can see if the array includes an element as follows:

```
myArray.includes("NB");        // return FALSE
myArray.includes("ON");        // return TRUE
```

Why is this useful


```
<input id="cardtype" type="text">
<button onclick="checkCard();">Click</button>
```

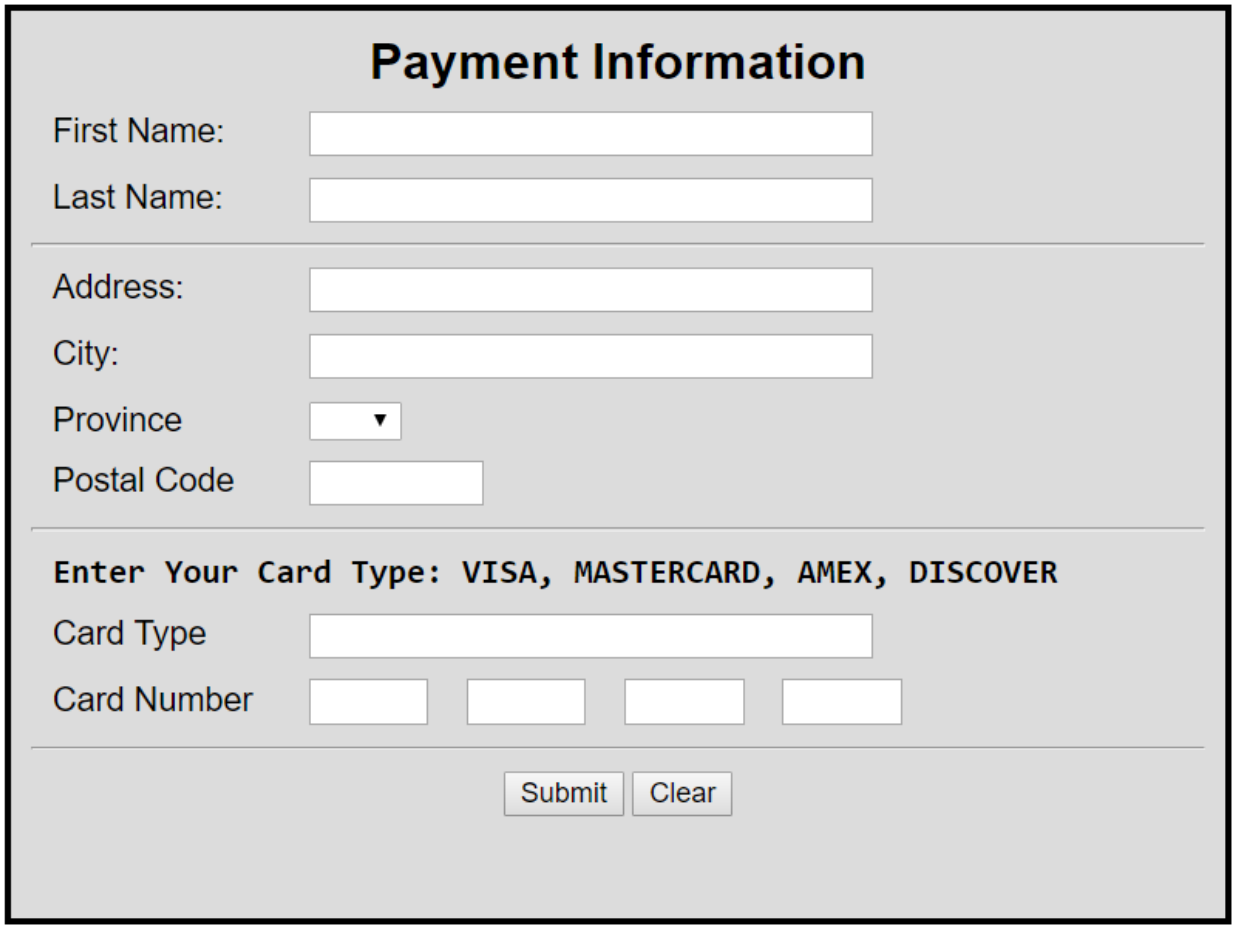
```
function checkCard() {
   var cardType = $("cardType").value;
   var cards = ["VISA", "MASTERCARD", "DISCOVER"];
   cardType = cardType.toUpperCase();
   if ( cards.include(cardType) )
   {
            console.log("A valid card was entered");
   }
}
```

We can keep an array of values. We can check to see if our input is one of these values!

OK – now we are ready

We have enough info with regex, strings, and arrays to verify our data.

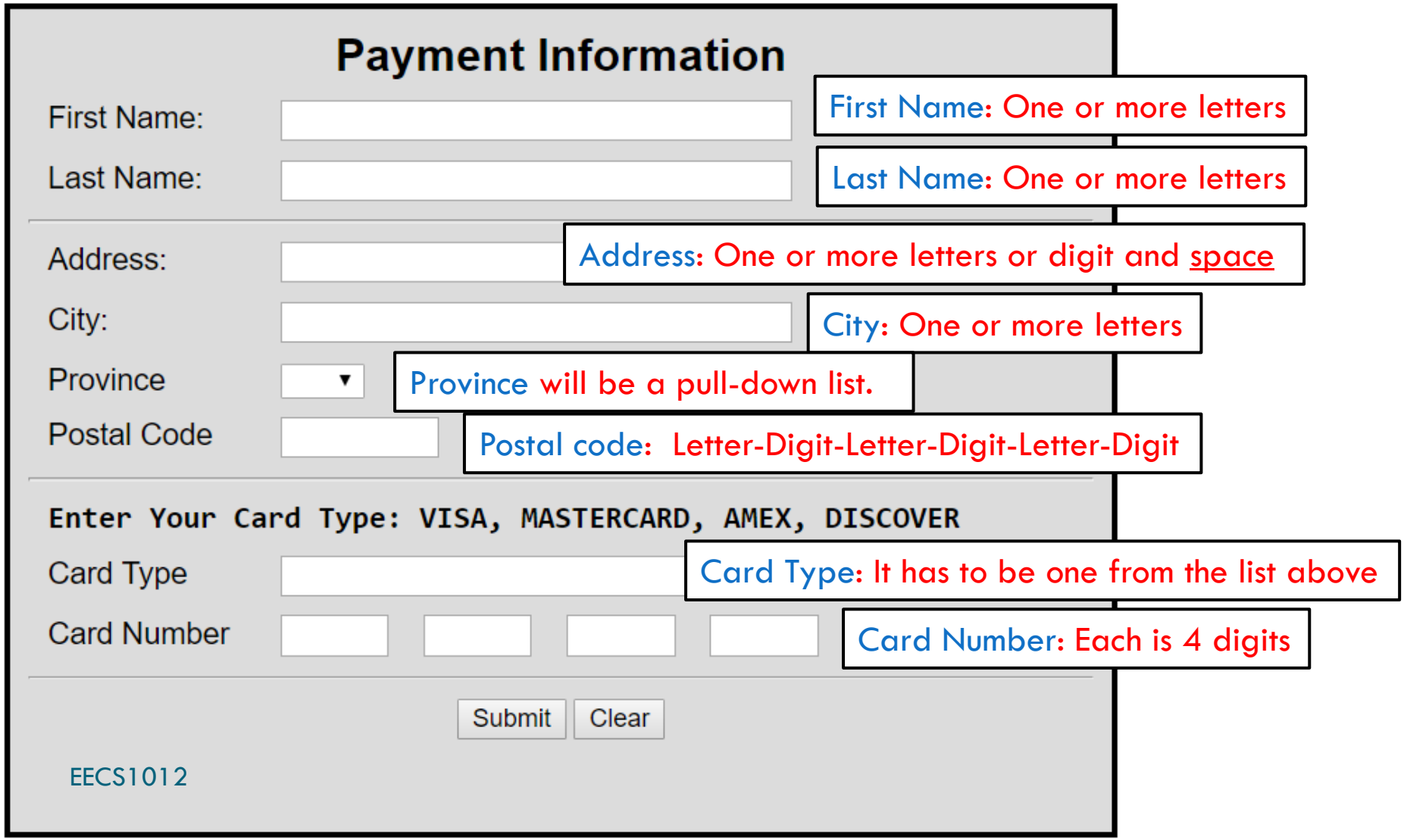
Let's consider each input



We need to first determine what we will considered to be a correct input for our fields.

Once we do this, we can decide the best way to validate.

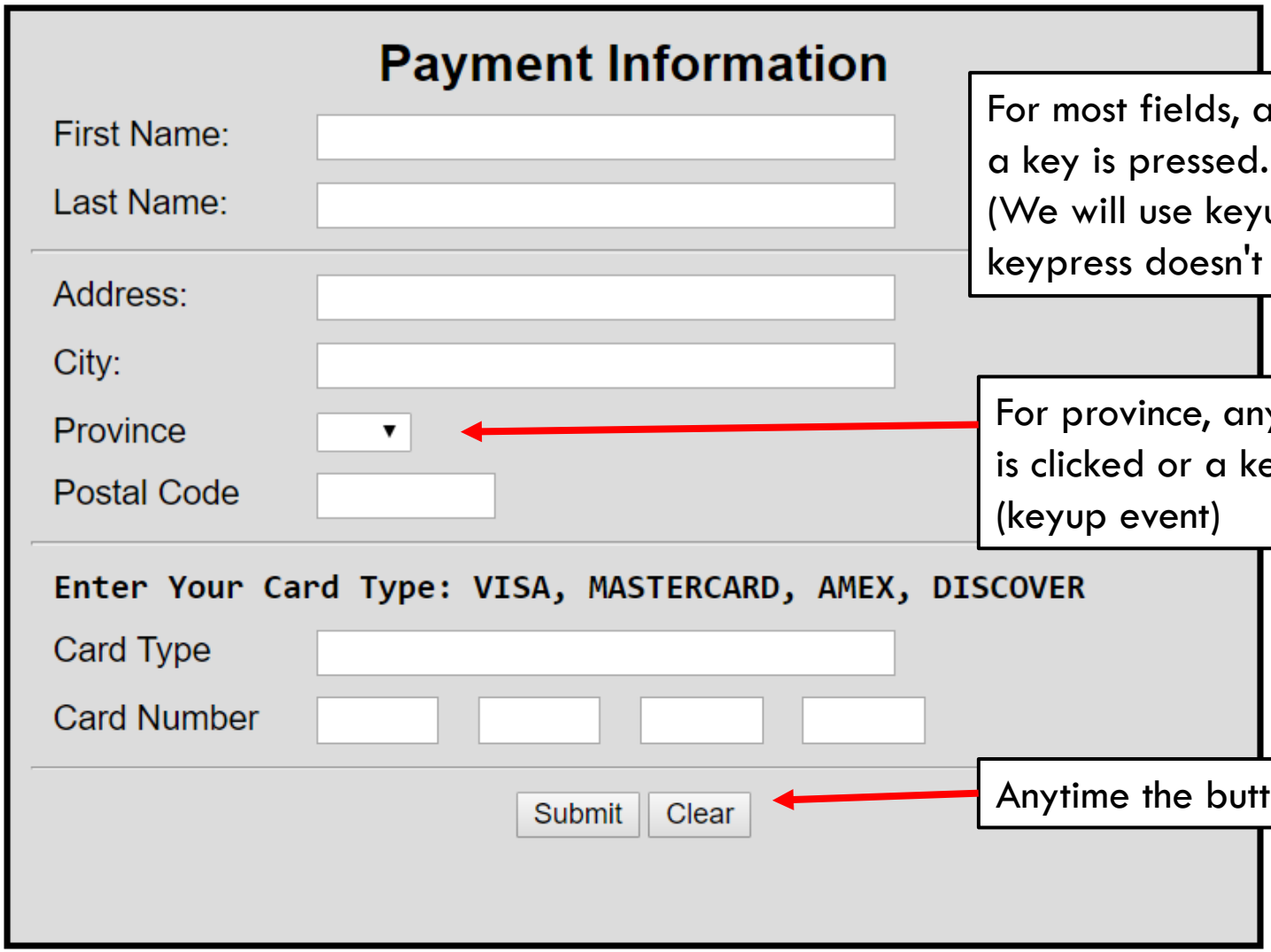
Each field


Payment Information
First Name:
Last Name:
Address:
City:
Province
Postal Code
Enter Your Card Type: VISA, MASTERCARD, AMEX, DISCOVER
Card Type
Card Number
Submit
Clear
EECS1012
First Name: One or more letters
Last Name: One or more letters
Address: One or more letters or digit and space
City: One or more letters
Province will be a pull-down list.
Postal code: Letter-Digit-Letter-Digit-Letter-Digit
Card Type: It has to be one from the list above
Card Number: Each is 4 digits

Validations for our example

First Name: One or more letters	`/[a-zA-Z]+/i`
Last Name: One or more letters	`/[a-zA-Z]+/i`
Address: One or more letters, digit and space	`/[0-9a-zA-Z\ ]+/i`
City: One or more letters	`/[0-9a-zA-Z]+/i`
Province will be a pull-down list.	no regex – but make sure something was selected
Postal code: Letter-Digit-Letter-Digit-Letter-Digit	`/^[A-Z][0-9][A-Z][0-9][A-Z][0-9]$/i`
Card Type: It has to be one from the list above	use array includes()
Card Number: Each is 4 digits	`/^[0-9][0-9][0-9][0-9]$/`

What events do we observe?



For most fields, anytime a key is pressed.
(We will use keyup – sometimes keypress doesn't get backspace)

For province, anytime the mouse is clicked or a key is pressed.
(keyup event)

Anytime the buttons are pressed.

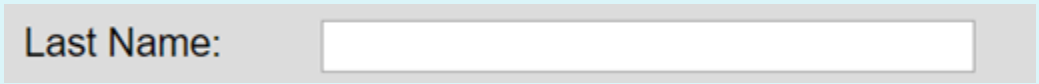
Our HTML file source (part 1)

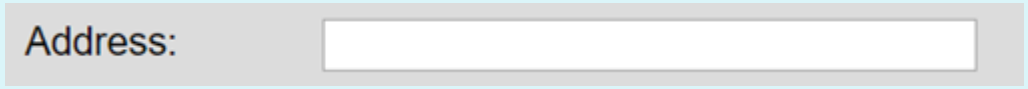

```
<html lang="en">
<head>
<title> Form Example </title>
<meta charset="utf-8">
<link href="FormsInput.css" rel="stylesheet">
<script src="prototype.js"></script>
<script src="FormsInput.js"></script>
</head>
<body>
<form>
<div class="box">
  <!-- First Name input -->
  <h2> Payment Information </h2>
  <p> <span class="fieldName">First Name: </span>
  <input type="text" id="FirstName" name="FirstName" class="input">
  <span class="message"></span></p>
```

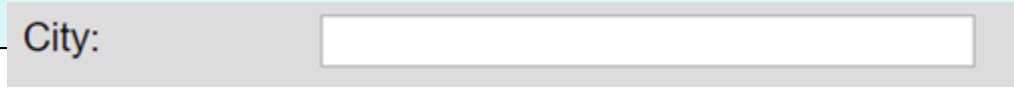
First Name:

Our HTML file source (part 2)


```
<!-- Last Name input -->
 <p> <span class="fieldName"> Last Name: </span>
 <input id="LastName" type="text" name="LastName" class="input">
 <span class="message"></span></p>

 <hr>
 <!-- Address -->
 <p> <span class="fieldName"> Address: </span>
 <input id="Address" type="text" name="LastName" class="input">
 <span class="message"></span></p>

 <!-- City -->
 <p> <span class="fieldName"> City: </span>
 <input id="City" type="text" name="LastName" class="input">
 <span class="message"></span></p>
```

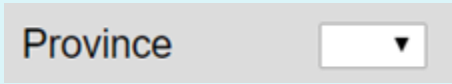




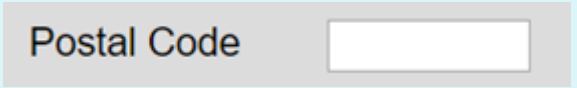


Our HTML file source (part 3)


```
  <!-- Provincene -->
  <p> <span class="fieldName"> Province </span>
  <select id="provinceList">
  </select>
  <span class="message"></span></p>

  <!-- Postal Code -->
  <p> <span class="fieldName"> Postal Code </span>
  <input id="PostalCode" type="text" name="LastName" class="input"
style="width:6em;">
  <span class="message"></span></p>

<hr>
```



Note – list is empty!



Our HTML file source (part 4)

```
<!-- Credit Card Input -->
 <p class="types"> Enter Your Card Type: VISA, MASTERCARD,
 AMEX, DISCOVER </p>
 <p> <span class="fieldName"> Card Type </span>
 <input type="text" id="Code" name="Code" class="input">
 <span class="message"></span></p>
```

Enter Your Card Type: VISA, MASTERCARD, AMEX, DISCOVER

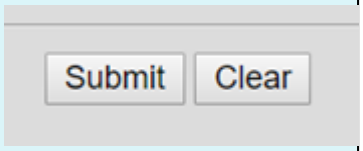
Card Type

```
<!-- Credit Card numbers Input -->
 <p> <span class="fieldName">Card Number </span>
  <input type="text" id="cc1" class="CCinput" maxlength="4">
   <span c lass="message"></span>
 <input type="text" id="cc2" class="CCinput" maxlength="4">
   <span class="message"></span>
 <input type="text" id="cc3" class="CCinput" maxlength="4">
   <span class="message"></span>
 <input type="text" id="cc4" class="CCinput" maxlength="4">
   <span class="message"></span>
 </p>
```

Card Number

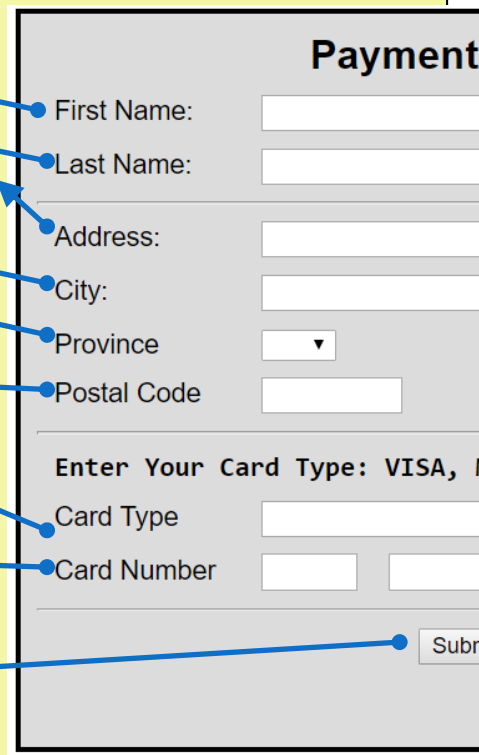
Our HTML file source (part 4)


```
<hr>
  <p style="text-align:center;">
    <button  id="submit" type="button"> Submit </button>
    <button> Clear </button> </p>
  <p style="text-align:center;" id="formError">   <p>
 </div>
 </form>
</body>
</html
```



Our JS file (part 1) - onload

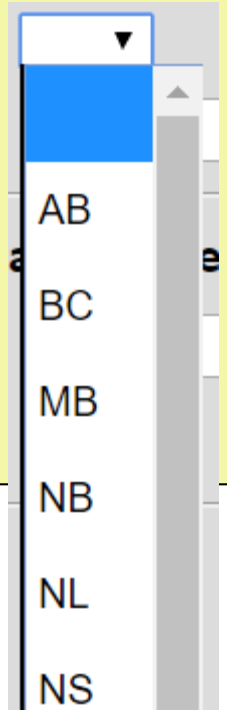

```
window.onload = function() {
   /* you could use keypress too */
   $("FirstName").observe("keyup", enforceLettersOnly);
   $("LastName").observe("keyup", enforceLettersOnly);
   $("Address").observe("keyup", enforceNumberLettersSpace);
   $("City").observe("keyup", enforceLettersOnly);
   $("provinceList").observe("click", enforceNotEmpty);
   $("provinceList").observe("keyup", enforceNotEmpty);
   $("PostalCode").observe("keyup", enforcePostalCode);
   $("cardType").observe("keyup", enforceCardType);
   $("cc1").observe("keyup", enforceCCNumber);
   $("cc2").observe("keyup", enforceCCNumber);
   $("cc3").observe("keyup", enforceCCNumber);
   $("cc4").observe("keyup", enforceCCNumber);
   $("submit").observe("click", submitForm);
```



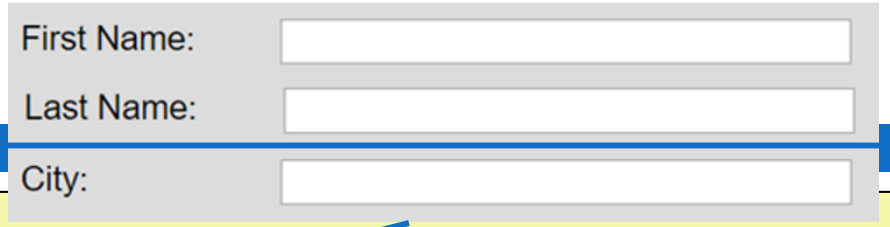
Our JS file (part 2) - onload


```
 /* this code is still part of onload */

/* Add provice codes */
   var provinceCodes = ["", "AB", "BC", "MB", "NB", "NL", "NS", "NT",
                        "NU", "ON", "PE", "QC", "SK", "YT"];

   /* add the options here */
   var option=null;
   for(var i=0; i < provinceCodes.length; i++)
   {
     option = new Option(provinceCodes[i]);
     $("provinceList").appendChild(option);
   }
}
```

Notice that the first option is an empty string.



JS file (part 3)




```
function enforceLettersOnly()
{
  var reg = new RegExp(/^[a-zA-Z]+$/);

  if (reg.test($(this).value) == true) {
      $(this).removeClassName("unsuccessful");
      $(this).addClassName("successful");
      $(this).nextElementSibling.innerHTML = "&#10004;";
    } else {
      $(this).addClassName("unsuccessful");
      $(this).removeClassName("successful");
      $(this).nextElementSibling.innerHTML = "&#10060;";
     }
}
```

Regex for LettersOnly

If true, remove class unsuccessful and add successful. Then change sibling span to ✓.

else, remove successful and then add unsuccessful. Then change sibling span to ✗

Called by our First and Last Name and City element

```
$("FirstName").observe("keyup", enforceLettersOnly);
$("LastName").observe("keyup", enforceLettersOnly);
$("City").observe("keyup", enforceLettersOnly);
```

JS file (part 4)

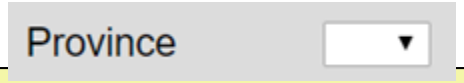
Address:

```
function enforceNumberLettersSpace()
{
  var reg = new RegExp(/^[a-zA-Z0-9\ ]+$/);

  if (reg.test($(this).value) == true) {
      $(this).removeClassName("unsuccessful");
      $(this).addClassName("successful");
      $(this).nextElementSibling.innerHTML = String.fromCharCode(10004);
    } else {
      $(this).addClassName("unsuccessful");
      $(this).removeClassName("successful");
      $(this).nextElementSibling.innerHTML = "&#10060;";
     }
}
```

Called by our Address element

```
$("Address").observe("keyup", enforceNumberLettersSpace);
```

JS file (part 5)




```
function enforceNotEmpty()
{
  if (this.value != "") {
      $(this).removeClassName("unsuccessful");
      $(this).addClassName("successful");
      $(this).nextElementSibling.innerHTML = "&#10004;";
    } else {
      $(this).addClassName("unsuccessful");
      $(this).removeClassName("successful");
      $(this).nextElementSibling.innerHTML = "&#10060;";
     }
}
```

For Province check to make sure selection isn't empty string

Called by our provinceList element (when clicked or modified by keyboard)

```
$("provinceList").observe("click", enforceNotEmpty);
$("provinceList").observe("keyup", enforceNotEmpty);
```

JS file (part 6)

Postal Code

```
function enforcePostalCode()
 {
     var reg = new RegExp(/^[A-Z][0-9][A-Z][0-9][A-Z][0-9]$/i);
     if (reg.test($(this).value) == true) {
         $(this).removeClassName("unsuccessful");
         $(this).addClassName("successful");
         $(this).nextElementSibling.innerHTML = "&#10004;";
       } else {
         $(this).addClassName("unsuccessful");
         $(this).removeClassName("successful");
         $(this).nextElementSibling.innerHTML = "&#10060;";
        }
 }
```

Regex for Postal Code

Called by our postal code element

```
$("PostalCode").observe("keyup", enforcePostalCode);
```

JS file (part 7)

Enter Your Card Type: VISA, MASTERCARD, AMEX, DISCOVER

Card Type

```
function enforceCardType()
{
  var codes=["VISA", "MASTERCARD", "AMEX", "DISCOVER"];
  var cardType = $("cardType").value;
  cardType = cardType.toUpperCase();
  if (codes.includes(cardType))
  {
    $(this).removeClassName("unsuccessful");
    $(this).addClassName("successful");
    $(this).nextElementSibling.innerHTML = "&#10004;";
  }
  else {
    $(this).addClassName("unsuccessful");
    $(this).removeClassName("successful");
    $(this).nextElementSibling.innerHTML = "&#10060;";
  }
}
```

(1) Create an array with card names.
(2) Get cardType input, change to upper case.
(3) Then search to see if that is in the array using includes() method.

Called by our cardType element

```
$("cardType").observe("keyup", enforceCardType);
```

JS file (part 8)

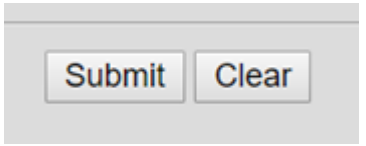
Regex for Digit-Digit-Digit-Digit

```
function enforeCCNumber()
 {
   var reg = new RegExp(/^[0-9][0-9][0-9][0-9]$/);
   if (reg.test($(this).value) == true) {
       $(this).removeClassName("unsuccessful");
       $(this).addClassName("successful");
       $(this).nextElementSibling.innerHTML = "&#10004;";
     } else {
       $(this).addClassName("unsuccessful");
       $(this).removeClassName("successful");
       $(this).nextElementSibling.innerHTML = "&#10060;";
      }
 }
```

Called by our CCNumber elements

```
$("cc1").observe("keyup", enforceCCNumber);
$("cc2").observe("keyup", enforceCCNumber);
$("cc3").observe("keyup", enforceCCNumber);
$("cc4").observe("keyup", enforceCCNumber);
```

JS file (part 9)




```
function submitForm() {
  var spans = document.getElementsByClassName("message");
  var valid = true;

  for(var i=0; i < spans.length; i++)
  {
    if (spans[i].innerHTML != String.fromCharCode(10004))
    {
      valid = false;
    }
  }

  if (valid)    {
    $("submit").style.border = "5px blue solid";
  }
  else {
    $("formError").innerHTML = "Make sure all fields are completed correctly.";
    setTimeout(clearErrorMsg, 1500);
  }
}
```

Get all the span elements.

Set valid to true.

(1) Loop through all span elements.
(2) Check to see if each's innerHTML is equal to the check mark unicode.
(3) If not, then set valid to false!

if valid is still true. Allow submit.

otherwise, show error message. set a timer to remove the message.

Called by our submit button

```
$("submit").observe("click", submitForm);
```

JS file (part 10)


```
function clearErrorMsg()
{
  $("formError").innerHTML = " ";
}
```

This function is used to clear our error message after 1.5 seconds.

We could be a bit more efficient


```
  if ( ( SOME CONDITION ) == true) {
       $(this).removeClassName("unsuccessful");
       $(this).addClassName("successful");
       $(this).nextElementSibling.innerHTML = "&#10004;";
     } else {
       $(this).addClassName("unsuccessful");
       $(this).removeClassName("successful");
       $(this).nextElementSibling.innerHTML = "&#10060;";
  }
```

This part in our code was quite redundant.

We could change this to two function()

```
function markSuccessful(element) {
    $(element).removeClassName("unsuccessful");
    $(element).addClassName("successful");
    $(element).nextElementSibling.innerHTML = "&#10004;";
}
```

```
function markUnsuccessful(element) {
       $(element).addClassName("unsuccessful");
       $(element).removeClassName("successful");
       $(element).nextElementSibling.innerHTML = "&#10060;";
}
```

Updated version


```
  if ( SOME CONDITION ) == true) {
       markSuccessful(this); /* passes this element to our func */
     } else {
       markUnsuccessful(this);  /* passes this element to our func */
  }
```

```
function markSuccessful(element) {
    $(element).removeClassName("unsuccessful");
    $(element).addClassName("successful");
    $(element).nextElementSibling.innerHTML = "&#10004;";
}
```

```
function markUnsuccessful(element) {
       $(element).addClassName("unsuccessful");
       $(element).removeClassName("successful");
       $(element).nextElementSibling.innerHTML = "&#10060;";
}
```

And that is how we do it

- You have seen an excellent test case of using your knowledge of:

1. HTML
2. CSS
3. FORMS
4. JavaScript
5. Most important - our own logic to make it work.

Summary

- non-Boolean conditions
- Some extra string and array functions
- Regular Expressions
- Commonly found case study putting together all our knowledge