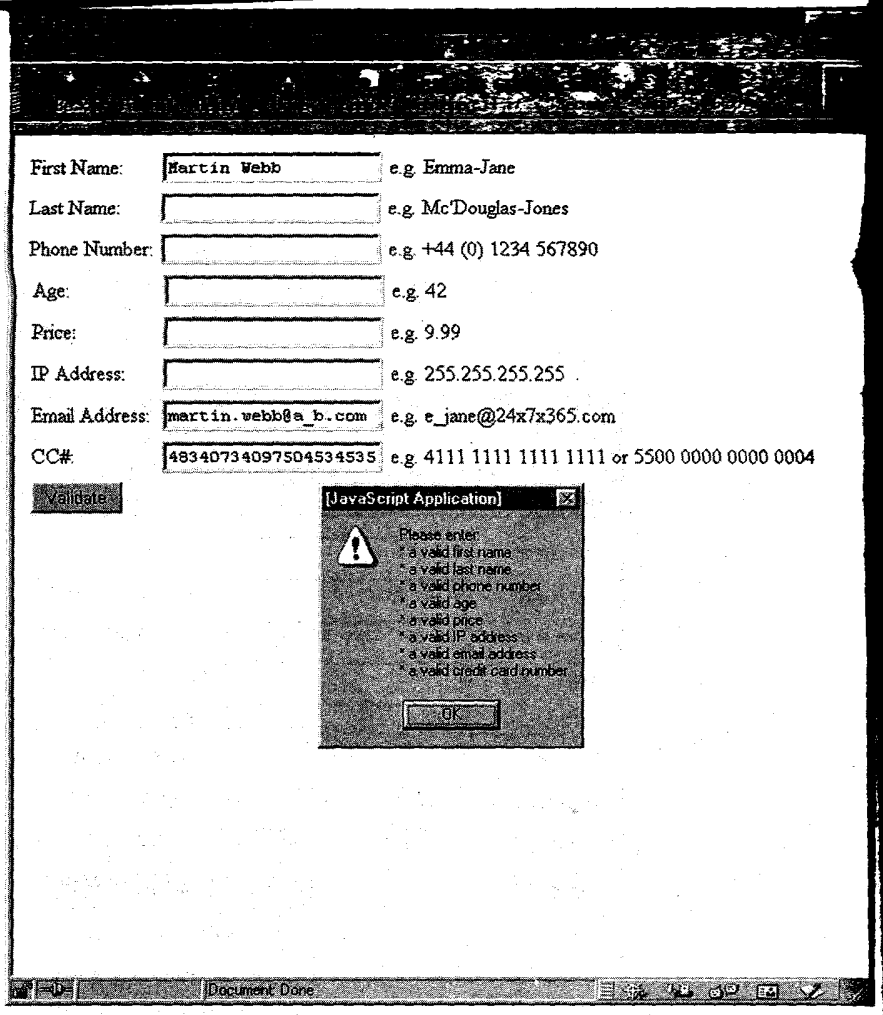
Figure 3-8
Validating Strings—Invalid form data.



Armed with these simple to complex regular expressions, you can adapt them further to match any weird and wonderful patterns of data that you need to validate.

Validating Strings Source Code

./chapter3/validate/form.htm

```
<html>

<head>
<title>Regular Expressions</title>
```

Regular expressions are only supported in JavaScript 1.2. Therefore, we set the script tag's `language` attribute to `JavaScript1.2`. Setting it to `JavaScript1.2` means only allowing browsers that support JavaScript 1.2 to process the enclosed JavaScript code:

```
<script language="JavaScript1.2"><!--
```

First Name We define several regular expressions using the forward slash characters to delimit the regular expressions. The first regular expression `isFirstName` can be used to match any alphabetical characters or the hyphen character (`-`). The metacharacter `^` indicates the start of the string, and the metacharacter `$` indicates the end of the string. The square brackets (`[` and `]`) indicate a valid range of literal characters: `A-Z` indicating all uppercase alphabetical characters from A through to Z, `a-z` indicating all lowercase alphabetical characters from a to z. To allow a hyphen character (`-`) within a range, we must escape it using a backslash character (`\`).

```
isFirstName = /^[A-Za-z\-]+$/;
```

The backslash character can be used to escape what would otherwise be treated as metacharacters to produce the literal character within regular expressions. For example, `\/`, `\\`, `\.`, `\*`, `\+`, `\?`, `\|`, `\(`, `\)`, `\[`, `\]`, `\{`, and `\}`.

Last Name The `isLastName` regular expression is similar to `isFirstName`, except that it also allows single quote characters (`'`) within the string:

```
isLastName  = /^[A-Za-z'\-]+$/;
```

Phone Number The `isPhone` regular expression can be used to match international phone numbers:

```
isPhone     = /^(\+\d+ )?(\(\d+\) )?[\d ]+$/;
```

The digits 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9 are represented using the character class `\d`. Repetition characters state how many times the preceding

character or group of characters can appear at this point within a match. The characters ? - match zero or one occurrence only, + - matches one or more occurrences, and * - matches zero or more occurrences.

Parentheses—that is, (and)—allow characters to be grouped together within an expression. This then allows metacharacters to act on the whole group and not just the preceding character. Therefore, in the isPhone regular expression, the portion ^(\+\d+)? states that the start of the string could have zero or one occurrence of: one plus character followed by one or more digits followed by a space caracter.

The next portion (\(\d+\))? states that the next part of the string (which could even be the start of the string) can contain zero or one occurrence of: one or more digits within a set of brackets followed by a space character.

The last portion [\d]+$ states that the end of the string must contain one or more: digits or spaces.

Age The isAge regular expression matches any string contain the numeric value 0 through to 129—any reasonable age. It uses the | character to allow matches on one or other of the two alternatives: 1[0-2]\d a 1 followed by 0, 1 or 2, followed by any digit, or \d{1,2} any digit followed by one or two other digits. The braces—that is, { and }—allow precise specification of repetition characters, in this case, a minimum of one and a maximum of two of the previous character or charactes:

```
isAge       = /^(1[0-2]\d|\d{1,2})$/;
```

Money The isMoney regular expression matches on a floating-point number with at least one digit before the decimal point and exactly two after the decimal point:

```
isMoney     = /^\d+\.\d{2}$/;
```

E-mail Address The validation of an e-mail address uses a slightly different approach. We specify two separate regular expressions, isEmail1 and isEmail2, with an isEmail() function, which shows how a string (s) can use the RegExp object's test() method to test that its regular expression matches on the passed string. Using the local AND operator (&&), we can test both at once and return true if both match or false if either the first doesn't, or if the first does but the second doesn't:

```
function isEmail(s) {
  return (isEmail1.test(s) && isEmail2.test(s));
}
```

The `isEmail1` regular expression is used to quickly match an e-mail address for groups of alphabetical characters, indicated by the character class `\w`—that is, a word character equivalent to `[a-zA-Z0-9_]` separated by hyphen or periods:

```
isEmail1    = /^\w+([\.\-]\w+)*\@\w+([\.\-]\w+)*\.\w+$/;
```

Breaking this expression down into stages:

- `^\w+` indicates that the e-mail address must start with one or more word characters.
- `([\.\-]\w+)*\@` indicates that zero or more occurrences of a hyphen or period character followed by one or more word characters must appear before the @ character.
- `\w+` indicates that one or more word characters must follow the @ character.
- `([\.\-]\w+)*` indicates that zero or more occurrences of a hyphen or period character followed by one or more word characters must appear.
- `\.\w+$` - indicates the e-mail address must finish with a period followed by one word character.

Unfortunately, the word character class `\w` includes a underscore character (_), which is not valid in a domain name. As `isEmail1` stands, it will incorrectly allow an e-mail address through of the format `first.second@domain_name.com`, where `domain_name.com` is invalid. The `isEmail2` regular expression overcomes this by not allowing the underscore character to appear after the @ character:

```
isEmail2    = /^.*@[^_]*$/;
```

We first match any and all characters up to the @ character using `^.*@`. The period metacharacter matches any character, the * repetition character matches zero of more occurrences of the previous character (i.e., the period metacharacter). Then we used negated character range using the caret character (^) to negate all the characters in the range—in this case, just the underscore character (_)—so that we match any character except the underscore character from the @ character to the end of the string.

If you think this will catch all valid and invalid e-mail addresses, think again. It will only vet the simplest forms of e-mail addresses, but this should be enough for our purposes.

IP Address The `isIPAddress` regular expression uses yet anothe approach to create a regular expression. An IP address has the basic for `255.255.255.255`, where the four numbers can be anything from through to 255. There appears to be a lot of repetition, so rather than creat a long regular expression by hand, we can use the `RegExp` object construc tor to build up a large regular expression using small string components.

First, we define an `ip` string that holds a regular expression to repre sent a number 0 through to 255. Note that as it is a string, we mus escape any backslash characters with a preceding backslash character:

```
ip = '(25[0-5]|2[0-4]\\d|1\\d\\d|\\d\\d|\\d)';
```

Next, we define an `ipdot` string to represent the regular expressio for the IP number followed by a period character. Again, the backslas character must be escaped:

```
ipdot = ip + '\\
```

We next create the Regular Expression using the `RegExp` object con structor combining three `ipdot` strings and one `ip` string:

```
isIPaddress = new RegExp('^'+ipdot+ipdot+ipdot+ip+'$');
```

Credit Card Numbers We can even create regular expressions to val idate the patterns of credit card numbers.

A Visa card number starts with 4 followed by 14 or 15 other digits:

```
isCCvisa       = /^4\d{14,15}$/;
```

A MasterCard number starts with `51`, `52`, `53`, `54`, or `55` followed by 1 other digits:

```
isCCmastercard = /^5[1-5]\d{14}$/;
```

An American Express number starts with `34` or `37` followed by 1 other digits:

```
isCCamex       = /^3[47]\d{13}$/;
```

A Diners Club card number starts with `300`, `301`, `302`, `304`, `305`, `360`, or `380` followed by 11 other digits:

```
isCCdinerscard = /^(30[0-5]|360|380)\d{11}$/;
```

A Discover card number starts with `6011` followed by 12 other digits:

```
isCCdiscover    = /^6011\d{12}$/;
```

An enRoute card number starts with `2014` or `2149` followed by 11 other digits:

```
isCCenroute     = /^(2014|2149)\d{11}$/;
```

A JCB card number starts with `2132` or `1800` followed by 11 other digits or starts with 3 and is followed by 15 other digits:

```
isCCjcb         = /^$((2131|1800)\d{11})|(3\d{15})/;
```

Rather than accept one credit card type and then vet it using the appropriate regular expression, the `isCC()` function accepts a string argument `s` for the credit card number and then uses each and every regular expression to attempt to find a matching credit card company.

```
function isCC(s) {
```

Before using any of the regular expressions to find a match, the `RegExp` object's `replace()` method is used to replace any whitespace characters, using the `\s` character class, with an empty string:

```
  s = s.replace(/\s/g,'');
```

Each credit card regular expression is tested using the `RegExp` object's `test()` method. If a match is found, the `test()` method returns `true`. In this case we return the results of the call to the `isLUHN()` method, passing the `s` string as input for all credit card companies except for enRoute:

```
  if (isCCvisa.test(s))       return isLUHN(s);
  if (isCCmastercard.test(s)) return isLUHN(s);
  if (isCCamex.test(s))       return isLUHN(s);
  if (isCCdinerscard.test(s)) return isLUHN(s);
  if (isCCdiscover.test(s))   return isLUHN(s);
  if (isCCenroute.test(s))    return true;
  if (isCCjcb.test(s))        return isLUHN(s);
```

Finally, if no credit card company has been matched, we simply return the results of the call to the `isLUHN()` function, again passing the `s` string: