

**Regular expressions continued...**

Regular expressions are handy when transforming plain text into HTML, and vice versa. Luckily, because these are such helpful subjects, PHP has many built-in functions to handle these tasks,

**EREg and PREG**

PHP offers two different sets of regular expression functions. The first set includes the traditional (or POSIX) functions, whose names each begin with `ereg` (for extended regular expressions; the `ereg` functions themselves are already an extension of the original feature set). The other set includes 647 [www.it-ebooks.info](http://www.it-ebooks.info) the Perl-compatible family of functions, prefaced with `preg` (for Perl-compatible regular expressions)

The `preg` functions use a library that mimics the regular expression functionality of the Perl programming language. This is a good thing because Perl allows you to do a variety of handy things with regular expressions.

There's no longer any reason to use the `ereg` functions and they are officially deprecated as of PHP 5.3.0. They offer fewer features, and they're slower than `preg` functions. The prototypes for the two sets of functions are identical, so it's easy to switch back and forth from one to another without too much confusion.

The only task of a regular expression program is to match a pattern in text. In regular expression patterns, most characters just match themselves. That is, the regular expression `rhino` matches strings that contain the five-character sequence `rhino`. The fancy business in regular expressions is due to a handful of punctuation and symbols called metacharacters. These symbols don't literally match themselves, but instead give commands to the regular expression matcher. The most frequently used metacharacters include the period (`.`), asterisk (`*`), plus sign (`+`), and question mark (`?`). (To match a literal metacharacter in a pattern, precede the character with a backslash.)

- The period means "match any character," so the pattern `.at` matches `bat`, `cat`, and even `rat`.

- The asterisk means “match 0 or more of the preceding object.” (So far, the only objects we know about are characters.)
- The plus is similar to asterisk, but means “match one or more of the preceding object.” So `.+at` matches `brat`, `sprat`, and even the `cat` inside of `catastrophe`, but not plain `at`. To match `at`, replace the `+` with an `*`.
- The question mark means “the preceding object is optional.” That is, it matches 0 or 1 of the object that precedes it. `colou?r` matches both `color` and `colour`.

To apply `*` and `+` to objects greater than one character, place the sequence of characters that make up the object inside parentheses. Parentheses allow you to group characters for more complicated matching and also capture the part of the pattern that falls inside them. A captured sequence can be referenced by `preg_replace()` to alter a string, and all captured matches can be stored in an array that’s passed as a third parameter to `preg_match()` and `preg_match_all()`. The `preg_match_all()` function is similar to `preg_match()`, but it finds all possible matches inside a string, instead of stopping at the first match.

Example shows a few examples of `preg_match()`, `preg_match_all()`, and `preg_replace()` at work.

*Example 23-1. Using preg functions*

```
if (preg_match('{<title>.+</title>}', $html)) {
    print "The page has a title!\n";
}

if (preg_match_all('/<li>/', $html, $matches)) {
    print 'Page has ' . count($matches[0]) . " list items\n";
}

// turn bold into italic
$italics = preg_replace('/(<\/?)b(>)/', '$1i$2', $bold);
```

- Normally, the pattern delimiter character, which starts and ends the pattern string, is `/`. Because the pattern delimiter character needs to be backslash-escaped if it appears as a literal inside the pattern.
- The preceding code uses open and close curly braces as delimiters in the first pattern string to avoid this problem. Any nonalphanumeric, non whitespace character (except backslash) can be a pattern delimiter character. If you use an open-bracket character as the opening delimiter, you can use a corresponding close bracket as the closing delimiter.

- The preceding code uses open and close curly braces as delimiters in the first pattern string to avoid this problem. Any nonalphanumeric, nonwhitespace character (except backslash) can be a pattern delimiter character. If you use an open-bracket character as the opening delimiter, you can use a corresponding close bracket as the closing delimiter.
- Anchoring your pattern enables matching against strings that only contain characters that the pattern describes. The caret (^) and the dollar sign (\$) anchor the pattern at the beginning and the end of the string, respectively. Without them, a match can occur anywhere in the string. So whereas [a-z0-9]+ means “one or more of a digit or lowercase English letter,” ^[a-z0-9]+ means “begins with one or more of a digit or lowercase English letter,” [a-z0-9]+\$ means “ends with one or more of a digit or lowercase English letter,” and ^[a-z0-9]+\$ means “contains only one or more of a digit or lowercase English letter.”

Example:

*Example 23-2. Matching with character classes and anchors*

```
$thisFileContents = file_get_contents(__FILE__);
// http://php.net/language.variables gives a regular expression for
// valid variable names in php. Beginning the pattern with |$ matches
// a literal $
$matchCount = preg_match_all('/\${[a-zA-Z_\x7f-\xff][a-zA-Z0-9_\x7f-\xff]*/',
                             $thisFileContents, $matches);
print "Matches: $matchCount\n";
foreach ($matches[0] as $variableName) {
    print "$variableName\n";
}
```

Example 23-2 prints each variable name it uses:

```
Matches: 8
$thisFileContents
$matchCount
$thisFileContents
$matches
$matchCount
$matches
$variableName
$variableName
```

To make a character class match the complement of what's inside it, begin the class with a caret. A caret outside a character class anchors a pattern at the beginning of a string; a caret inside a character class means "match everything except what's listed in the square brackets." For example, the character class `[^aeiou]` matches everything but lowercase English vowels. Note that the opposite of `[aeiou]` isn't `[bcdfghjklmnpqrstvwxyz]`. The character class `[^aeiou]` also matches uppercase vowels such as AEIOU, numbers such as 123, URLs such as `http://www.cnpq.br/`, and even emoticons such as `:)`. The vertical bar (`|`), also known as the pipe, specifies alternatives. Example:

*Example 23-3. Matching with |*

```
$text = "The files are cuddly.gif, report.pdf, and cute.jpg.";
if (preg_match_all('/[a-zA-Z0-9]+\.(gif|jpe?g)/', $text, $matches)) {
    print "The image files are: " . implode(', ', $matches[0]);
}
```

Example 23-3 prints:

```
The image files are: cuddly.gif,cute.jpg
```

Switching from ereg to preg

## Problem

You want to convert from using `ereg` functions to `preg` functions.

## Solution

First, you have to add delimiters to your patterns:

```
preg_match('/pattern/', 'string');
```

For case-insensitive matching, use the `/i` modifier with `preg_match()` instead:

```
preg_match('/pattern/i', 'string');
```

When using integers instead of strings as patterns or replacement values, convert the

number to hexadecimal and specify it using an escape sequence:

```
$hex = dechex($number);
```

```
preg_match("/\x$hex/", 'string');
```

## Discussion

There are a few major differences between `ereg` and `preg`. First, when you use `preg`

functions, the pattern isn't just the string `pattern`; it also needs delimiters, as in Perl, so

it's `/pattern/` instead.<sup>1</sup> So:

```
ereg('pattern', 'string');
```

becomes:

```
preg_match('/pattern/', 'string');
```

When choosing your pattern delimiters, don't put your delimiter character inside the

regular expression pattern, or you'll close the pattern early. If you can't find a way to

avoid this problem, you need to escape any instances of your delimiters using the backslash.

Instead of doing this by hand, call `addslashes()`.

For example, if you use / as your delimiter:

```
$ereg_pattern = '<b>.+</b>';
```

```
$preg_pattern = addslashes($ereg_pattern, '/');
```

the value of `$preg_pattern` is now `<b>.+<\/b>`.

The `preg` functions don't have a parallel series of case-insensitive functions. They have

a case-insensitive modifier instead. To convert, change:

```
eregi('pattern', 'string');
```

to:

```
preg_match('/pattern/i', 'string');
```

Adding the `i` after the closing delimiter makes the change.

Finally, there is one last obscure difference. If you use a number (not a string) as a pattern

or replacement value in `ereg_replace()`, it's assumed you are referring to the ASCII

value of a character. Therefore, because 9 is the ASCII representation of tab (i.e., `/t`),

this code inserts tabs at the beginning of each line:

```
$tab = 9;
```

```
$replaced = ereg_replace('^', $tab, $string);
```

Here's how to convert linefeed endings:

```
$converted = ereg_replace(10, 12, $text);
```

To avoid this feature in `ereg` functions, use this instead:

```
$tab = '9';
```

On the other hand, `preg_replace()` treats the number 9 as the one-character string

'9', not as a tab substitute. To convert these character codes for use in `preg_re`

place()), convert them to hexadecimal and prefix them with `\x`. For example, 9 becomes `\x9` or `\x09`, and 12 becomes `\x0c`. Alternatively, you can use `\t`, `\r`, and `\n` for tabs, carriage returns, and linefeeds, respectively.

NOTE: For any doubt contact on Whatsapp no.: 9873961590