

maXbox



# maXbox Starter 20

## Start with Regular Expressions

### 1.1 From TRex to RegEx

Regular expressions are the main way many tools matches' patterns within strings. For example, finding pieces of text within a larger document, or finding a restriction site within a larger sequence. This tutorial illustrates what a RegEx is and what you can do to find, match, compare or replace text of documents or code.

As you will see Regular expressions are composed of characters, character classes, metacharacters, quantifiers, and assertions. Hope you did already read Starters 1 till 19 at:

<http://sourceforge.net/apps/mediawiki/maxbox/>

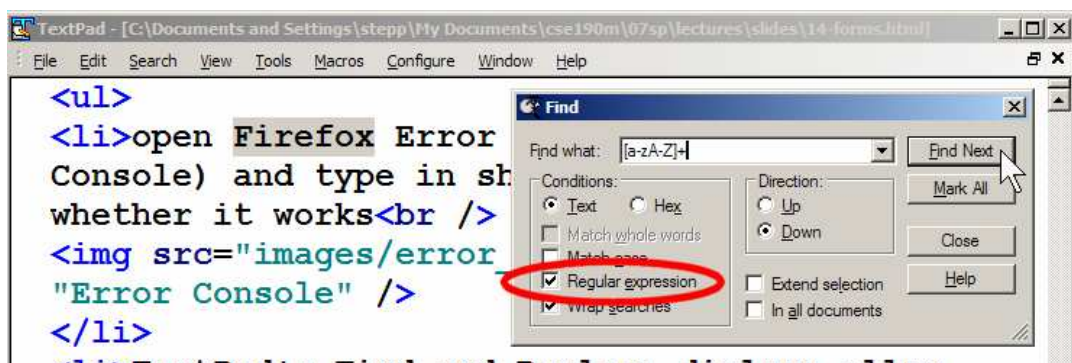
What's a Regular Expression?

A regular expression (RegEx): describes a search pattern of text typically made up from special characters called metacharacters:

- You can test whether a string matches the expression pattern
- You can use a RegEx to search/replace characters in a string
- It's very powerful, but tough to read

Regular expressions occur in many places and masks, not alone in code or languages environments:

Text editors, shells or search tools like Grep<sup>1</sup> allow also RegEx in search/replace functions, like the following screen-shot figure out.




1: Find with RegEx

Let's jump to history and the beginning of RegEx with Perl. Perl was a horribly flawed and very useful scripting language, based on UNIX shell scripting and C, that helped lead to many other

---

<sup>1</sup> Global Regular Expression Print / Parser

better languages. Perl was and is also excellent for string/file/text processing because it built regular expressions directly into the language as a first-class data type.


 Many command-line shell Linux/Mac tools (ed, vi, grep, egrep, sed, awk) support Regular Expressions, for e.g. Grep:


```
grep -e "[pP]hone.*206[0-9]{7}" contacts.txt
>> phone { 206-685-2181}
```

Grep is a tool that originated from the UNIX world during the 1970's. It can search through files and folders (directories in UNIX) and check which lines in those files match a given regular expression. Grep will output the filenames and the line numbers or the actual lines that matched the regular expression.

## 1.2 RegEx out of the Box

As you already know the tool is split up into the toolbar across the top, the editor or code part in the centre and the output window at the bottom or the interface part on the right. Change that in the menu /View at our own style.

 In maXbox you will execute the RegEx as a script, libraries and units are already built.

 Before this starter code will work you will need to download maXbox from the website. It can be down-loaded from <http://www.softwareschule.ch/maxbox.htm> (you'll find the download maxbox3.zip on the top left of the page). Once the download has finished, unzip the file, making sure that you preserve the folder structure as it is. If you double-click maxbox3.exe the box opens a default demo program. Test it with F9 / F2 or press **Compile** and you should hear a sound. So far so good now we'll open the example:


```
309_regex_powertester2.txt
```

If you can't find the two files try also the zip-file (48100 bytes) loaded from:  
[http://www.softwareschule.ch/examples/309\\_regex\\_powertester2.txt](http://www.softwareschule.ch/examples/309_regex_powertester2.txt)

Now let's take a look at the code of this first part project. Our first line is

```
01 program RegEx_Power_Tester_TRegx;
```

We name it, means the program's name is above.

 This example requires two objects from the classes: TRegExpr and TPerlRegEx of PerlRegEx so the second one is from the well known PCRE Lib. TPerlRegEx is a Delphi VCL wrapper around the open source PCRE library, which implements Perl-Compatible Regular Expressions.

This version of TPerlRegEx is compatible with the TPerlRegEx class (PCRE 7.9) in the RegularExpressionsCore unit in Delphi XE. In fact, the unit in Delphi XE and maXbox3 is derived from the version of TPerlRegEx that we are using now.

Let's do a first RegEx now. We want to check if a name is a valid Pascal name like a syntax checker does. We use straight forward a function in the box:

```
732 if ExecRegExpr('^[a-zA-Z_][a-zA-Z0-9_]*','pascal_name_kon')
    then writeln('pascal name valid') else writeln('pascal name invalid');
```

This is a useful global function:

```
function ExecRegExpr (const ARegExpr, AInputStr: string): boolean;
```

It is true if a string `AInputString` matches regular expression `ARegExpr` and it will raise an exception if syntax errors in `ARegExpr` are done.

Now let's analyse our first RegEx step by step `^[a-zA-Z][a-zA-Z0-9_].*`:

<code>^</code>	matches the beginning of a line; <code>\$</code> the end
<code>[a-z]</code>	matches all twenty six small characters from 'a' to 'z'
<code>[a-zA-Z_]</code>	matches any letter with underscore
<code>[a-zA-Z0-9_]</code>	matches any letter or digit with underscore
<code>.*</code> (a dot)	matches any character except <code>\n</code>
<code>.*</code>	means 0 or more occurrences
<code>[ ]</code>	group characters into a character set;

A lot of rules for the beginning, and they look ugly for novices, but really they are very simple (well, usually simple ;)), handy and powerful tool too. You can validate e-mail addresses; extract phone numbers or ZIP codes from web-pages or documents, search for complex patterns in log files and all you can imagine! Rules (templates) can be changed without your program recompilation! This can be especially useful for user input validation in DBMS and web projects. Try the next one:

```
if ExecRegExpr('M[ae][iy]e?r.*[be]', 'Mairhuberu')
    then writeln('regex maierhuber true') else writeln('regex maierhuber false');
```

? Means 0 or 1 occurrences

Any item of a regular expression may be followed by another type of metacharacters – called iterators. Using this characters you can specify number of occurrences of previous characters so inside `[ ]`, most modifier keys act as normal characters:

`/what[.!?*]*/` matches "what", "what.", "what!", "what?\*\*\*!", ..

So a character class is a way of matching 1 character in the string being searched to any of a number of characters in the search pattern. Character classes are defined using square brackets. Thus `[135]` matches any of 1, 3, or 5. A range of characters (based on ASCII order) can be used in a character class: `[0-7]` matches any digit between 0 and 7, and `[a-z]` matches and small (but not capital) letter.



Note that the hyphen is being used as a metacharacter here. To match a literal hyphen in a character class, it needs to be the first character. So `[-135]` matches any of -, 1, 3, or 5. `[-0-9]` matches any digit or the hyphen.

What if we want to define a certain place? An assertion is a statement about the position of the match pattern within a string. The most common assertions are `^`, which signifies the beginning of a string, and `$`, which signifies the end of the string.

For example search all empty or blank lines: Search empty lines: `^ ^$`

This is how we can assert a valid port number with `^` and `$`:

```
745 if ExecRegExpr('^(?:\d\d?\d?\d?\d?)$',':80009')
    then writeln('regex port true') else writeln('regex port false');
```

There are 3 main operators that use regular expressions:

1. **Matching** (which returns TRUE if a match is found and FALSE if no match is found.
2. **Substitution**, which substitutes one pattern of characters for another within a string
3. **Split**, which separates a string into a series of substrings

If you want to match a certain number of repeats of a group of characters, you can group the characters within parentheses. For example, `/(cat){3}/` matches 3 reps of “cat” in a row: “catcatcat”. However, `/cat{3}/` matches “ca” followed by 3 t’s: “cattt”.

And things go on. To negate or reject a character class, that is, to match any character EXCEPT what is in the class, use the caret `^` as the first symbol in the class. `[^0-9]` matches any character that isn’t a digit. `[^-0-9]` matches any character that isn’t a hyphen or a digit.

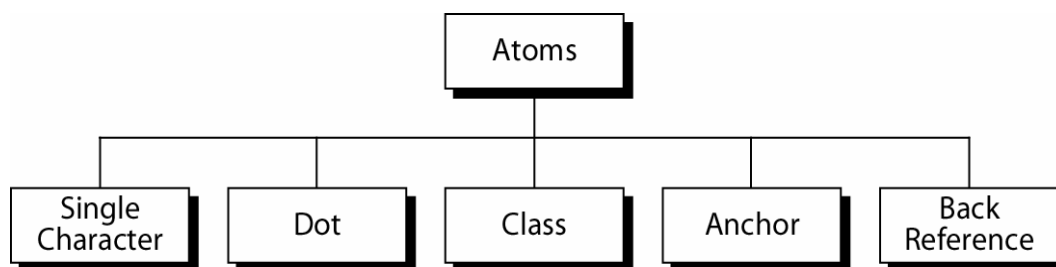
Now its time to reflect:

RE Metacharacter	Matches...
<code>^</code>	beginning of line
<code>\$</code>	end of line
<code>\char</code>	Escape the meaning of <i>char</i> following it
<code>[^]</code>	One character <u>not</u> in the set
<code>\&lt;</code>	Beginning of word anchor
<code>\&gt;</code>	End of word anchor
<code>()</code> or <code>\()</code>	Tags matched characters to be used later (max = 9)
<code> </code> or <code>\ </code>	Or grouping
<code>x{m}</code>	Repetition of character <i>x</i> , <i>m</i> times ( <i>x,m</i> = integer)
<code>x{m,}</code>	Repetition of character <i>x</i> , at least <i>m</i> times
<code>x{m,n}</code>	Repetition of character <i>x</i> between <i>m</i> and <i>n</i> times

## 2. Overview of Matches

You can specify a series of alternatives for a pattern using `|` to separate them, so that `fee|fie|foe` will match any of “fee”, “fie”, or “foe” in the target string (as would `f(e|i|o)e`). The first alternative includes everything from the last pattern delimiter (`"'`, `"[`, or the beginning of the pattern) up to the first `|`, and the last alternative contains everything from the last `|` to the next pattern delimiter.

For this reason, it’s common practice to include alternatives in parentheses, to minimize confusion about where they start and end.



Next a few examples to see the atoms:

```

rex:= '(no)+.*'; //Print all lines containing one or more consecutive
occurrences of the pattern "no".

rex:= '.*S(h|u).*'; //Print all lines containing the uppercase letter "S",
followed by either "h" or "u".

rex:= '.*\.[^0][^0].*'; //Print all lines ending with a period and exactly two
non-zero numbers.
  
```

```
rex:= '.*[0-9]{6}\.***'; //all lines at least 6 consecutive numbers follow by a
period.
```

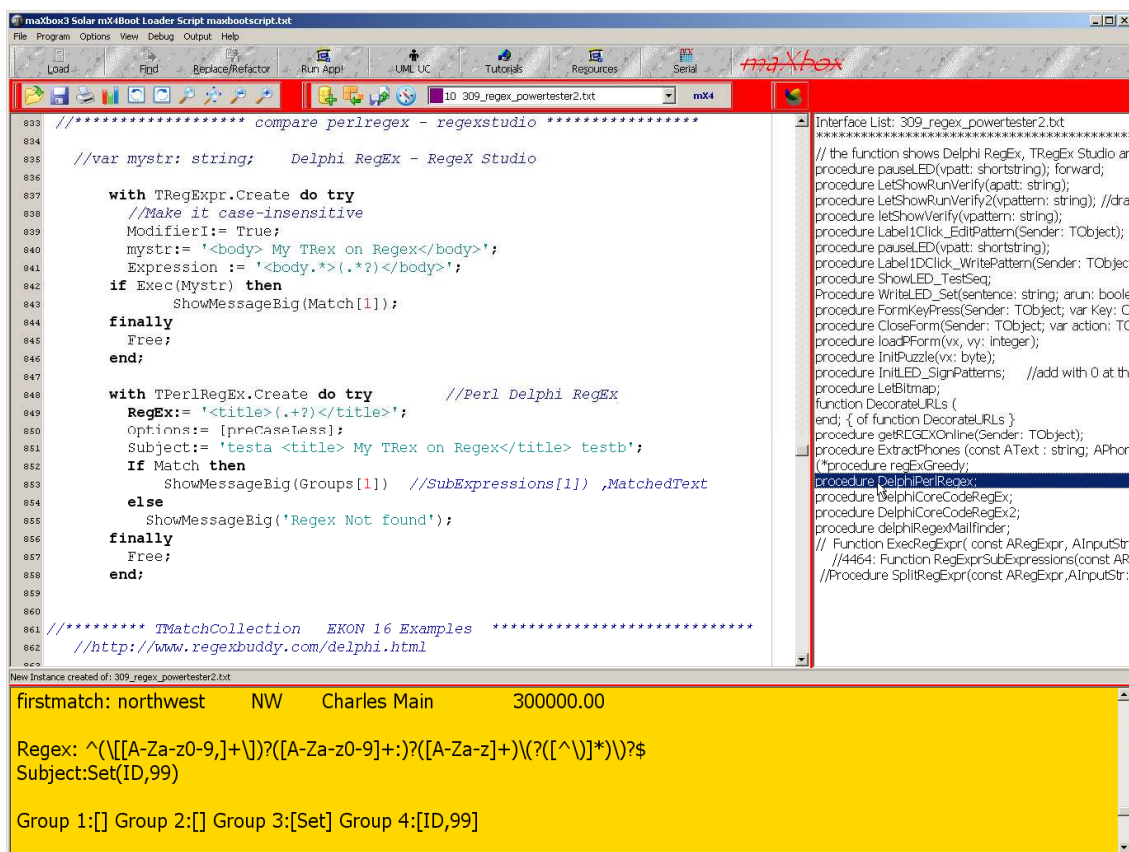
Next we want to see how the objects in the box work:

The static versions of the methods are provided for convenience, and should only be used for one off matches, if you are matching in a loop or repeating the same search often then you should create an 'instance' of the TRegEx record and use the non static methods.

The RegEx unit defines TRegEx and TMatch as records. That way you don't have to explicitly create and destroy them. Internally, TRegEx uses TPerlRegEx to do the heavy lifting.

TPerlRegEx is a class that needs to be created and destroyed like any other class. If you look at the TRegEx source code, you'll notice that it uses an interface to destroy the TPerlRegEx instance when TRegEx goes out of scope. Interfaces are reference counted in Delphi, making them usable for automatic memory management.

☞ The XE interface to PCRE is a layer of units based on contributions from various people, the PCRE API header translations in RegularExpressionsAPI.pas



3: The two Classes in compare

Clients are seen in picture 3 as the two classes do the same task. After creating the object RegEx we set the options:

These are the most important options you can specify:

- **preCaseLess** Tries to match the regex without paying attention to case. If set, 'Bye' will match 'Bye', 'bye', 'BYE' and even 'byE', 'bYe', etc. Otherwise, only 'Bye' will be matched. Equivalent to Perl's /i modifier.
- **preMultiLine** The ^ (beginning of string) and \$ (ending of string) regex operators will also match right after and right before a newline in the Subject string. This effectively treats one string with multiple lines as multiple strings. Equivalent to Perl's /m modifier.

- **preSingleLine** Normally, dot (.) matches anything but a newline (\n). With preSingleLine, dot (.) will match anything, including newlines. This allows a multiline string to be regarded as a single entity. Equivalent to Perl's /s modifier.
- Note that preMultiLine and preSingleLine can be used together.
- **preExtended** Allow regex to contain extra whitespace, newlines and Perl-style comments, all of which will be filtered out. This is sometimes called "free-spacing mode".
- **preAnchored** Allows the RegEx to match only at the start of the subject or right after the previous match.
- **preUngreedy** Repeat operators (?, \*, +, {num,num}) are greedy by default, i.e. they try to match as many characters as possible. Set preUngreedy to use ungreedy repeat operators by default, i.e. they try to match as few characters as possible.



Greedy is a strange operator or option. A slight explanation about "greediness".

"Greedy" takes as many as possible; "non-greedy" takes as few as possible. For example, 'b+' and 'b\*' applied to string 'abbbbc' return 'bbbbb', 'b+?' returns 'b', 'b\*?' returns an empty string, 'b{2,3}?' returns 'bb', 'b{2,3}' returns 'bbb'.

The regular expression engine does "greedy" matching by default!

A typical RegEx client session looks like this:

```
848 with TPerlRegex.Create do try           //Perl Delphi RegEx
849     RegEx:= '<title>(.*?)</title>';
850     Options:= [preCaseLess];
851     Subject:= 'testa <title> My TRex on Regex</title> testb';
852     If Match then
853         ShowMessageBig(Groups[1]) //SubExpressions[1]) ,MatchedText
854     else
855         ShowMessageBig('Regex Not found');
856 finally
857     Free;
858 end;
```

Subject is the RegEx and the string on which Match will try to match RegEx. Match attempts to match the regular expression specified in the RegEx property on the string specified in the Subject property. If Compile has not yet been called, Match will do so for you.

Call MatchAgain to attempt to match the RegEx on the remainder of the subject string after a successful call to Match.



**Compile:** Before it can be used, the regular expression needs to be compiled. Match will call Compile automatically if you did not do so. If the regular expression will be applied in time-critical code, you may wish to compile it during your application's initialization. You may also want to call Study to further optimize the execution of the RegEx.

Let's have a look at the RegEx itself and the magic behind:

```
'<title>(.*?)</title>'
```

What's about this <title>, it must be a global identifier to find or still exists.

+? one or more ("non-greedy"), similar to {1,}?

It captures everything between the first <title> and the first </title> that follows. This is usually what you want to do with large sequences in a group.



☞ **Greedy names:** Greedy matching can cause problems with the use of quantifiers. Imagine that you have a long DNA sequence and you try to match `/ATG(.*)TAG/`. The `“.*”` matches 0 or more of any character. Greedy matching causes this to take the entire sequence between the first ATG and the last TAG. This could be a very long matched sequence.

☞ Note that Regular expressions don't work very well with nested delimiters or other tree-like data structures, such as are found in an HTML table or an XML document. We will discuss alternatives later in a course.



So far we have learned little about RegEx and the past with a TRex eating words as a book output to us;-). Now it's time to run your program at first with F9 (if you haven't done yet) and learn something about the `309_regex_powertester2.txt` with many code snippets to explore.

One of them is a song finder in the appendix to get a song list from an mp3 file and play them!



4: Mastering

There are plenty more regular expression tricks and operations, which can be found in Programming Perl, or, for the truly devoted, *Mastering Regular Expressions*. Next we enter part two of the insider information about the implementation.



5: Enter a RegEx building

### 1.3 RegEx in Delphi and maXbox

The `TPerlRegEx` class aims at providing any Delphi, Java or C++Builder developer with the same, powerful regular expression capabilities provided by the Perl programming language community, created by Larry Wall.

It is implemented as a wrapper around the open source `PCRE` library.

The regular expression engine in Delphi XE is PCRE (Perl Compatible Regular Expression). It's a fast and compliant (with generally accepted RegEx syntax) engine which has been around for

many years. Users of earlier versions of Delphi can use it with `TPerlRegEx`, a Delphi class wrapper around it.

`TRegEx` is a record for convenience with a bunch of methods and static class methods for matching with regular expressions.

I've always used the `RegularExpressionsCore` unit rather than the higher level stuff because the core unit is compatible with the unit that Jan Goyvaerts has provided for free for years. That was my introduction to regular expressions. So I forgot about the other unit. I guess there's either a bug or it just doesn't work the way one might expect.

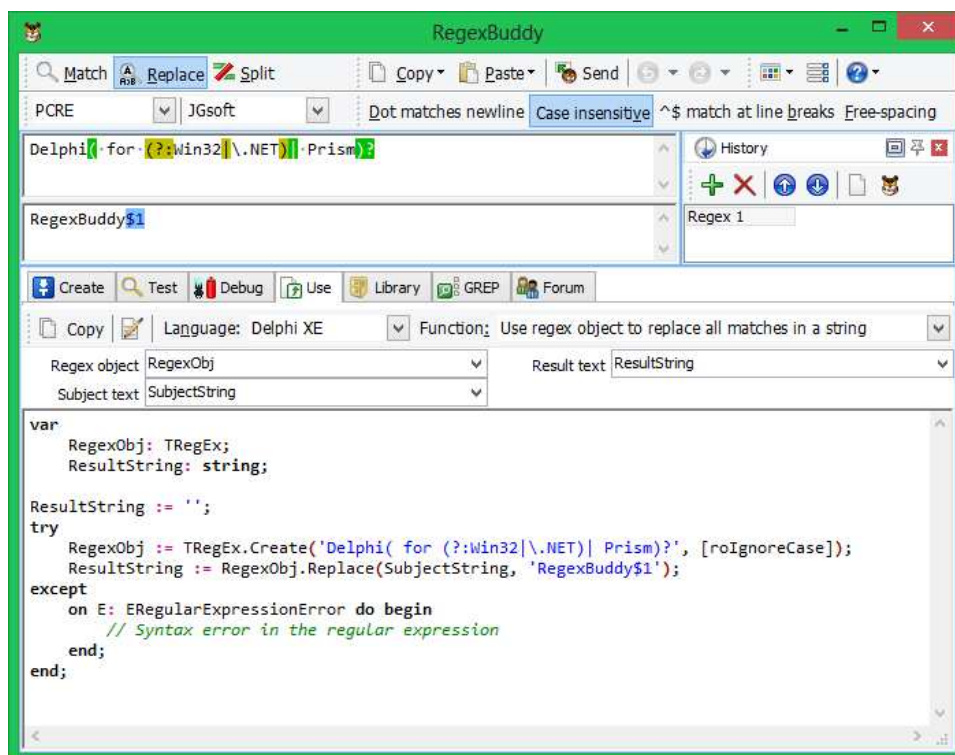
For new code written in Delphi XE, you should definitely use the `RegEx` unit that is part of Delphi rather than one of the many 3rd party units that may be available. But if you're dealing with UTF-8 data, use the `RegularExpressionsCore` unit to avoid needless UTF-8 to UTF-16 to UTF-8 conversions.

☞ The procedure `Study procedure Study;`

Allows studying the `RegEx`. Studying takes time, but will make the execution of the `RegEx` a lot faster. Call `study` if you will be using the same `RegEx` many times. `Study` will also call `Compile` if this had not yet been done.

Depending on what the user entered in `Edit1` and `Memo1`, `RegEx` might end up being a pretty complicated regular expression that will be applied to the memo text a great many times. This makes it worthwhile to spend a little extra time studying the regular expression (later on more).

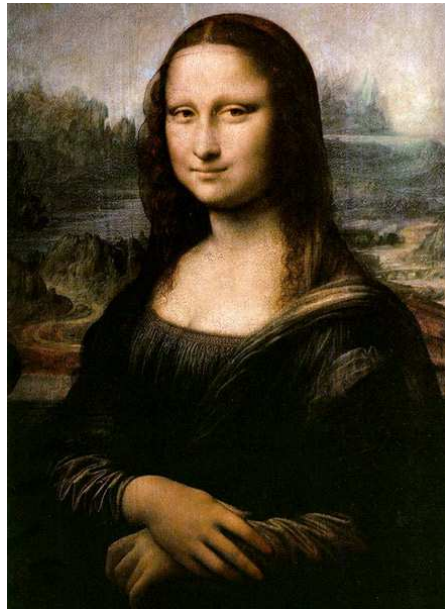
By the way there's another tool: Compose and analyze `RegEx` patterns with `RegexBuddy`'s easy-to-grasp `RegEx` blocks and intuitive `RegEx` tree, instead of or in combination with the traditional `RegEx` syntax. Developed by the author of the website <http://www.regular-expressions.info/>, `RegexBuddy` makes learning and using regular expressions easier than ever.



6: The `RegexBuddy` and the GUI

Conclusion: A regular expression (`RegEx` or `regexp` for short) is a special text string for describing a search pattern. You can think of regular expressions as wildcards on steroids. You are probably familiar with wildcard notations such as `*.txt` to find all text files in a file manager. The `RegEx` equivalent is `.*\.txt$`.





7: The Secret behind this Regular Expression!?

Study method example:

Depending on what the user entered in `Edit1` and `Mem1`, `RegEx` might end up being a pretty complicated regular expression that will be applied to the memo text a great many times. This makes it worthwhile to spend a little extra time studying the regular expression.

```
31 with PerlRegEx1 do begin
    RegEx:= Edit1.Text;
    Study;
    Subject:= Mem1.Lines.Text;
    Replacement:= Edit2.Text;
    ReplaceAll;
    Mem1.Lines.Text:= Subject;
end;
```



Try reformat phone numbers from 206-685-2181 format to (206) 685.2181 format to get data back:

You can use back-references when replacing text. Text "captured" in `()` is given an internal number; use `\number` to refer to it elsewhere in the pattern `\0` is the overall pattern, `\1` is the first parenthetical capture, `\2` the second, ...

Example: "A" surrounded by same character: `/(\. )A\1/`

Example: to swap a last name with a first name:

```
var name = "Durden, Tyler";
    name = name.replace(/(\w+),\s+(\w+)/, "$2 $1");
    // "Tyler Durden")
```



machine33 Solar.msd

File Program Options View Debug Output Help

Load Edit Format Background Color Compiler Lifecycle Toolbox References Send

298\_bitbit\_animation3 bit msd

```

1  /* *****
2  * Project : System Graphic Function Overview
3  * App Name : 298_bitbit_animation3, loc's = 292
4  * Purpose : Demonstrates bitmap manipulation - bitmapulation
5  * Date : 09/12/2012 - 17:07
6  * History : convert bitbit API to maxbox Nov 2011
7  * system save demo for m3.9.2, finished yet!!
8  * animates a self visible autocorrelation with motion control
9  * *****
10
11 Program BitBit_Animation3:
12
13 (BOOL WINAPI MessageKeep(
14     in UINT uType );)
15
16 //TThreadFunction = function(P: Pointer): Longint; stdcall;
17 //Procedure ExecuteThread(afunc: TThreadFunction; var thrOK: boolean);
18
19 Const MILLISECONDS = 100;
20
21
22 function MessageBoxTimeout(hWnd: HWND; lpText, lpCaption: PChar; uType: UINT;
23     wLanguageId: WORD; dwMilliseconds: DWORD): Integer;
24     external 'MessageBoxTimeoutAUser32.dll stdcall';
25
26
27 procedure CloseClick(Sender: TObject; var action: TCloseAction); forward;
28
29 Const
30     BCFMAP = 'examples\citymax.bmp';
31
32
33 machine33 C:\machine33\examples\298_bitbit_animation3.exe Compiled: 2012.11.20 20:12:50
34
35 machine name is: APSN21
36 user name is: max
37 OS Type is: 15
38 15.11.2012 18:18:50 for maxbox3 file
39 Stop Watch CPU Time: 0:0:0.656
40 m3x executed: 15.11.2012 20:12:51 Runtime: 0:0:1.494
41 SysEdit
42 Animation Form being closed
43 Ver: 3.9.6.3 (396). Work Dir: E:\maxbox\maxbox3_back
  
```

Interface List: 298\_bitbit\_animation3 bit

\*\*\*\*\*

Project : System Graphic Function Overview

//TThreadFunction = function(P: Pointer): Longint; stdcall;

//Procedure ExecuteThread(afunc: TThreadFunction; var thrOK: boolean);

procedure MessageBoxTimeout(hWnd: HWND; lpText, lpCaption: PChar; uType: UINT; wLanguageId: WORD; dwMilliseconds: DWORD): Integer;

procedure CloseClick(Sender: TObject; var action: TCloseAction); forward;

procedure FormDrawnMap(const frame: String; var thrOK: boolean);

procedure BitmapFormCreate(Sender: TObject);

procedure PlayGroundForm\_UpdateImage;

procedure PlayGroundForm\_TimerTimer(Sender: TObject);

procedure BitClearClick(Sender: TObject);

procedure CloseClick(Sender: TObject; var action: TCloseAction); forward;

procedure BitClickButton(Sender: TObject);

procedure FormMouseDown(Sender: TObject);

procedure FormMouseMove(Sender: TObject);

procedure FormMouseUp(Sender: TObject);

procedure FormMouseWheel(Sender: TObject);

procedure CallTimer(msec: Integer);

## 1.4 Appendix

### EXAMPLE: Mail Finder

```
procedure delphiRegexMailfinder;
begin
    // Initialize a test string to include some email addresses. This would
    normally be your eMail.
    TestString:= '<one@server.domain.xy>, another@otherserver.xyz';
    PR:= TPerlRegEx.Create;
    try
        PR.RegEx:= '\b[A-Z0-9._%+-]+@[A-Z0-9.-]+\.[A-Z]{2,4}\b';
        PR.Options:= PR.Options + [preCaseLess];
        PR.Compile;
        PR.Subject:= TestString; // <-- tell PR where to look for matches
        if PR.Match then begin
            WriteLn(PR.MatchedText); // Extract first address
            while PR.MatchAgain do
                WriteLn(PR.MatchedText); // Extract subsequent addresses
            end;
        finally
            PR.Free;
        end;
    //ReadLn;
end;
```

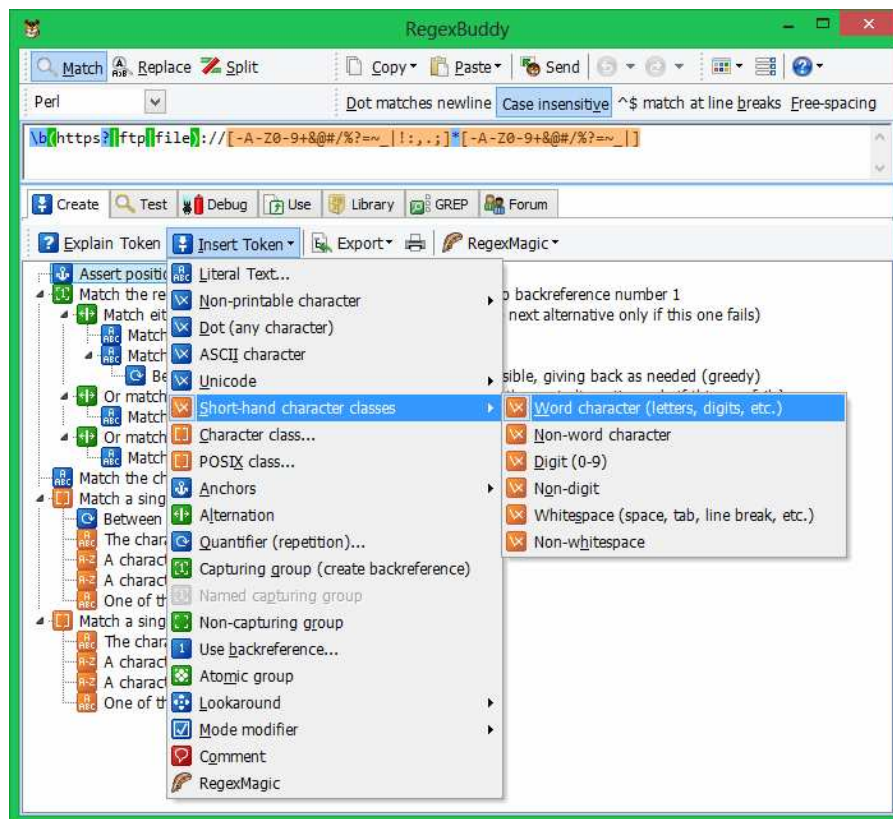
### EXAMPLE: Songfinder

```
with TRegExpr.Create do try
    gstr:= 'Deep Purple';
    modifierS:= false; //non greedy
    Expression:= '#EXTINF:\d{3},'+gstr+' - ([^\n].*)';
    if Exec(fstr) then
        Repeat
            writeln(Format ('Songs of ' +gstr+' : %s', [Match[1]]));
            (*if AnsiCompareText(Match[1], 'Woman') > 0 then begin
                closeMP3;

                PlayMP3('..\EKON_13_14_15\EKON16\06_Woman_From_Tokyo.mp3');
            end;*)
        Until Not ExecNext;
    finally Free;
end;
```

```
//***** Code Finished*****
```

## 1.5 Appendix RegxBuddy in Action



## 1.6 Appendix String RegEx methods

<code>.match(<i>regexp</i>)</code>	returns first match for this string against the given regular expression; if global /g flag is used, returns array of all matches
<code>.replace(<i>regexp</i>, <i>text</i>)</code>	replaces first occurrence of the regular expression with the given text; if global /g flag is used, replaces all occurrences
<code>.search(<i>regexp</i>)</code>	returns first index where the given regular expression occurs
<code>.split(<i>delimiter</i>[, <i>limit</i>])</code>	breaks apart a string into an array of strings using the given regular as the delimiter; returns the array of tokens