Applying Regular Expressions

Steve Gordon

.NET Engineer and Microsoft MVP

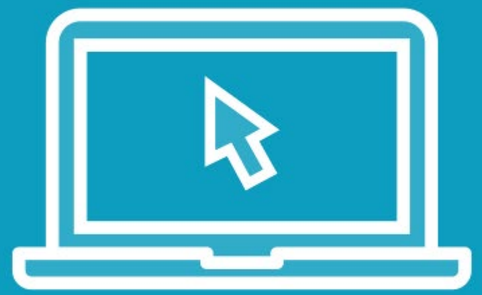
@stevejgordon www.stevejgordon.co.uk

Overview

Define and use regular expressions

- Considerations for using regex
- Learn about syntax of regex patterns
- Match and capture text using regex

Demo



Focus on how and where to apply regular expressions

Use Regex.Split

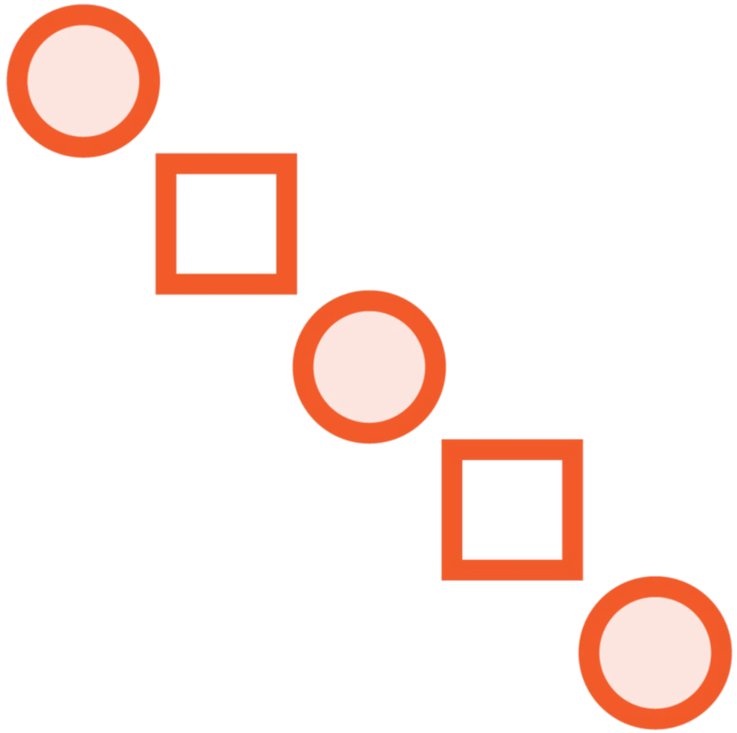
Measure the performance implications of regex using benchmarks

Warning!

Regex is not always the best choice for working with strings.

Regex has overhead when compared to raw string manipulation.

Advice



Consider whether using regex simplifies code maintenance

Regex has some performance overhead

- Extra memory usage
- Potentially slower processing

Regex is best suited to complex pattern matching

- Can reduce lines of code vs. manual string manipulation

A trade-off exists between complexity and performance

Anchors

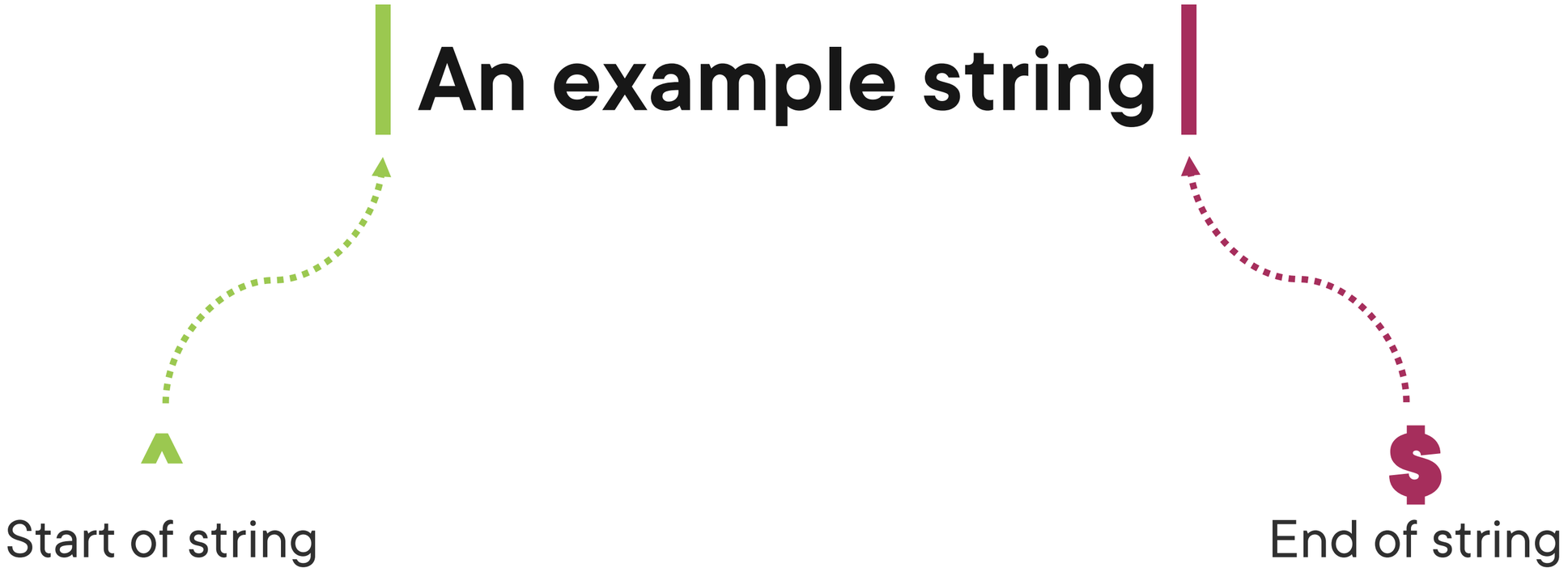
Anchors

Metacharacters that specify where a match must occur within a string

Atomic zero-width assertions

- Do not cause the regex engine to advance or consume characters within a string

Common Anchors



Multiline Mode

^

Multiline Mode Off

Match must occur at the beginning of the string.

Multiline Mode On

Match must occur at the beginning of each line.

$

Multiline Mode Off

Match must occur at the end of a string, or before the \n at the end of a string.

Multiline Mode On

Match must occur at the each line, or before \n at the end of a line.

```
var pattern = "^Ca";
var match = Regex.Match("Cats", pattern);
match = Regex.Match("Dogs", pattern);
match = Regex.Match("   Cats", pattern);
match = Regex.Match("Animals: Cats", pattern);
match = Regex.Match("Canary", pattern);


var patternTwo = "^Cats$";
match = Regex.Match("Cats | Dogs", patternTwo);
match = Regex.Match("Cats", patternTwo);
match = Regex.Match("   Cats", patternTwo);
match = Regex.Match(“cats", patternTwo);
```

True

False

False

False

True

False

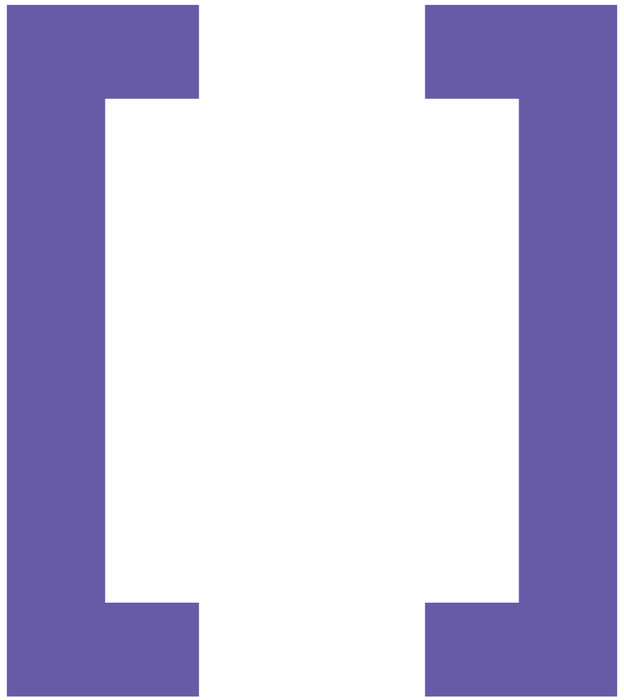
True

False

False

Character Classes

Character Classes



A character class defines a set of characters

- Any character in the set can occur within the input string for a match to succeed

Several metacharacters represent common character sets

A specific set or range of characters may be defined within square brackets

- A positive character group defines a set of allowed characters which can occur
- A negative character group defines a set of characters which should not occur

Character Group Examples

Pattern	Explanation
[aeiou]	Matches any lowercase vowel.
[a-z]	Matches any lowercase character in the range ‘a’ through ‘z’.
[0-2]	Match a digit from the range (0, 1 or 2).
[^AEIOU]	Matches any character except uppercase vowels.
[a-zA-Z]	Matches any character in either of its two ranges. Essentially matches any English letter.

Question

Why do we need to use two ranges?

Character Group Ranges

A-Z

Matched based on their Unicode code point

Any code point between the start and end character (inclusive) is a match

Unicode Code Points

Hex	Character
...	
58	X
59	Y
5A	Z
5B	[
5C	\
5D	]

Hex	Character
5E	^
5F	_
60	`
61	a
62	b
63	c
...	

Unicode Code Points

Hex	Character
...	
58	X
59	Y
5A	Z
5B	[
5C	\
5D	]

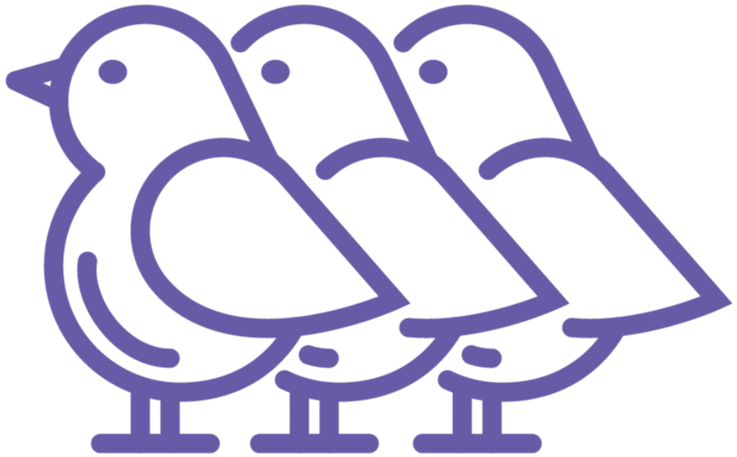
Hex	Character
5E	^
5F	_
60	`
61	a
62	b
63	c
...	

Other Character Classes

Pattern	Matches
.	Any character (wildcard)
\w	Any word character
\W	Any non-word character
\d	Any digit character
\D	Any non-digit character
\s	Any whitespace character
\S	Any non-whitespace character

Quantifiers

Quantifiers



Used to specify how many instances of a character, group or character class must occur within the input

Special metacharacters are available for common quantifiers

Specific numeric values can also be provided

Quantifiers can be either greedy or lazy

- Greedy: Match as much as possible
- Lazy: Match as little as possible

Greedy by default, but can be made lazy by following them with a question mark '?'

Quantifier Examples

Greedy	Lazy	Matches
*	*?	Zero or more times
+	+?	One or more times
?	??	Zero or one time
{3}	{3}?	Exactly 3 times
{3,}	{3,}?	At least 3 times
{3,6}	{3,6}?	Between 3 and 6 times
{,10}	{,10}?	Between zero and 10 times

Groups and Subexpressions

Groups and Subexpressions

Subexpressions are defined within a grouping construct

- Extract substrings within a larger matched string for separate processing
- Group a subexpression to apply a quantifier

May be capturing or non-capturing

- Groups capture by default

Captures are automatically numbered from left to right

Support optional naming of the group

(subexpression)

Matched Subexpressions

Parentheses are used to define a subexpression and capture the match.

`(?:subexpression)`

Non-capturing Subexpressions

A grouping construct which includes a subexpression, but does not capture the matched substring.

```
(?<name>subexpression)
```

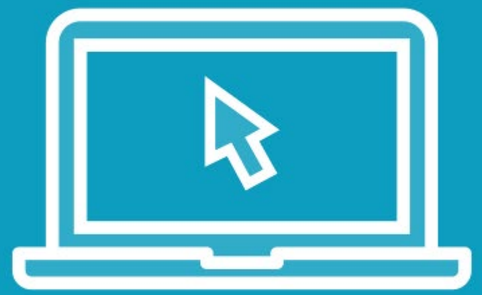
Named Matched Subexpressions

A grouping construct which includes a subexpression, capturing the matched substring with a name that can be used to access it from the match.

Examples

```
var pattern = "(?:Az){3}";
var match = Regex.Match("AzAzAz", pattern); // True
match = Regex.Match("AzAz", pattern); // False
match = Regex.Match("AzThingAzAzAzAz", pattern); // True
```

Demo



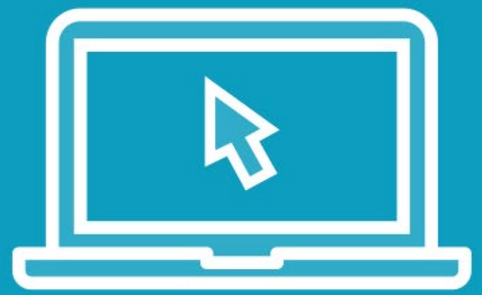
- Apply regex in the data processing application
- Perform complex pattern matching
- Apply anchors
- Apply character classes
- Apply quantifiers
- Apply grouping constructs with subexpressions

```
// Namespace:

System.Text.RegularExpressions

// Definition:

public static Match Match (string input, string pattern);

// Use:

Match result = Regex.Match("An input string","^A.");
```

Regex.Match

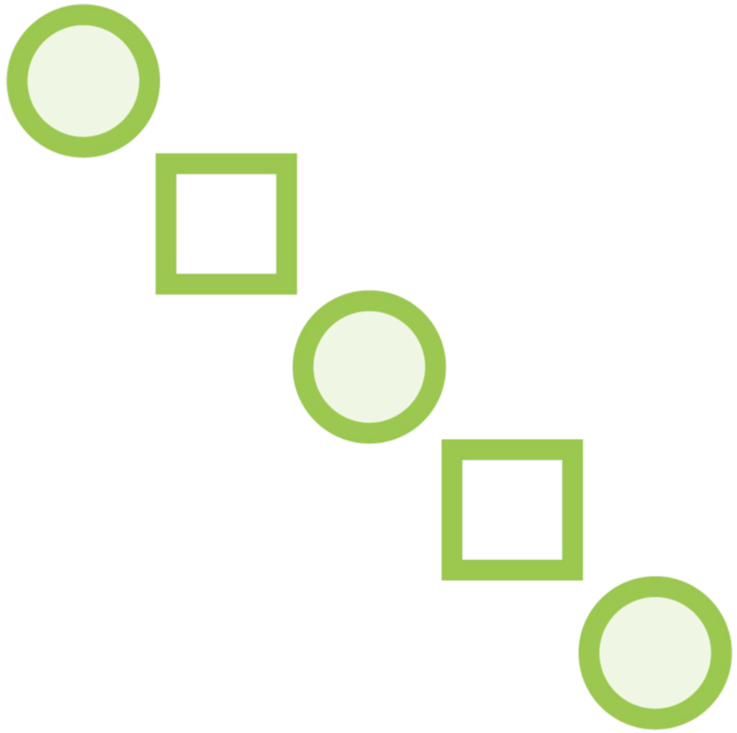
Searches an input string for a substring that matches a regular expression pattern and returns the first occurrence as a single Match object.

Demo



Control captured values from subexpressions

Regex Advantages



Include validation when matching

- Ensure we have seven columns
- Final column is in the expected format

Validation failures result in a failed match

Extract substrings during matching

Regex has additional overhead

- In most applications, this cost is reasonable
- The code we need to maintain is reduced

There are often several ways to achieve the same goal using regular expression patterns.

https://docs.microsoft.com/en-us/dotnet/standard/base-types/regular-expression-language-quick-reference

Microsoft documentation

Up Next:
Applying String Comparisons and Sorting