# Strings

# Declaração e concatenação de uma string

Strings são sequências de caracteres usadas para representar texto, delimitadas por aspas duplas. As strings podem ser concatenadas utilizando o **+.**

```
string variavel = "texto";

string variavel2 = "texto2";

Console.WriteLine("texto" + "texto2");
```

# Declaração e concatenação de uma string

```
string linguagem = "C#";
string expressao = "Olá mundo ";

Console.WriteLine(expressao + linguagem);
```

Variáveis simples sendo declaradas com aspas duplas.

Se quisermos agrupar mais de um texto, usamos a concatenação. Aqui, ao concatenar as strings, temos um único texto "Olá mundo C#"

# Interpolação de strings

Para evitar muitas concatenações, podemos interpolar strings, utilizando o “$” antes do texto e as “{}” para indicar variáveis.

```
Console.Write("Olá! Digite seu nome: ");
string nome = Console.ReadLine();

Console.Write("Agora, digite sua idade: ");
string idade = Console.ReadLine();

Console.Write("Agora, digite seu telefone: ");
string telefone = Console.ReadLine();

Console.WriteLine("Nome: " + nome + ", idade: " + idade + ", telefone: " + telefone);
Console.WriteLine($"Nome: {nome}, idade: {idade}, telefone: {telefone}");
```

Na primeira linha, estamos concatenando variáveis do jeito tradicional, enquanto na segunda estamos interpolando as strings. Perceba que, com a interpolação, evitamos confusões associadas ao uso de vários “+” na mesma frase.

# Sequências de escape

Existem alguns caracteres especiais ao trabalharmos com strings, como as aspas (“”) e a contra-barra(\). Para representar esses caracteres especiais, utilizamos as sequências de escape. Cada sequência de escape representa algo diferente.

```
var frase = "Olá, \nmeu email é \"iasmin@email.com\"";
Console.WriteLine(frase);
```

Estão sendo usadas duas sequências de escape diferentes: **\n** e **\”**. Ao final, a frase será impressa assim:

```
Olá,
meu email é “iasmin@email.com”
```

# Sequências de escape

Na tabela abaixo, estão listadas as sequências de escape mais utilizadas quando trabalhamos com strings:

| Sequência | Descrição |
|---|---|
| \n | Passa para a linha de baixo |
| \t | Tabula a saída |
| \\ | Representa a contra-barra |
| \’ | Representa as aspas simples |
| \” | Representa as aspas duplas |
| \b | Apaga o caractere anterior |

# Verbatim Literal

Além das sequências de escape, existe um outro recurso no C# para trabalhar com caracteres especiais: o **verbatim literal**. Com ele, é possível escrever o **texto exatamente como queremos que seja impresso**, sem precisar usar escapes. Para usar o verbatim, adicionamos um "**@**" antes da string.

```
var frase = "Olá, \nmeu email é \"iasmin@email.com\"";
var frase2 = @"Olá,
meu email é ""iasmin@email.com""";

var caminho = "C:\\Users\\adria\\Documents\\requisicao.txt";
var caminho2 = @"C:\Users\adria\Documents\requisicao.txt";
```

Ao lado, vemos a diferença entre usar escapes e o verbatim. No verbatim, para colocar aspas no texto, usamos duas aspas seguidas (**""**). Essa é uma das poucas exceções onde precisamos de expressões para representar caracteres especiais. Fora isso, ele facilita muito ao lidar com caminhos ou textos que teriam muitos escapes.

# Métodos de manipulação

Existem diversos métodos para trabalharmos com strings, transformando-as para o que melhor se adapta ao nosso trabalho. A seguir, estão listados os principais métodos de manipulação de strings de C#:

- Length(): Retorna o número de caracteres da string.
- ToLower() e toUpper(): Converte todos os caracteres da string para minúsculas ou maiúsculas, respectivamente.
- Contains(“texto”): Verifica se a string contém o texto especificado.

# Métodos de manipulação

- StartsWith("texto") e EndsWith("texto"): Verifica se a string começa ou termina com o texto indicado.
- IndexOf("x"): Retorna a posição da primeira ocorrência do caractere ou texto. Se não encontrar, retorna -1.
- Substring(inicio, tamanho): Retorna uma parte da string, começando do índice informado (parâmetro `inicio`), com a quantidade de caracteres desejada (parâmetro `tamanho`).
- Replace("antigo", "novo"): Substitui todas as ocorrências do texto antigo pelo novo.

# Métodos de manipulação

- Trim(),  TrimStart() e TrimEnd(): Remove espaços em branco do início e fim da string (`Trim()`), só do início (`TrimStart()`), ou só do fim (`TrimEnd()`).
- Split(‘ ’): Divide a string em partes com base no caractere separador, como espaço ou vírgula.
- Join(", ", array): Junta os elementos de um array em uma única string, separados pelo texto indicado (neste caso, vírgula e espaço).

alura

# REGEX

# Regex

Regex é a abreviação de “Regular Expressions”, ou “**Expressões regulares**”.

As expressões regulares são utilizadas para **descrever padrões que ocorrem em textos**. Uma vez que identificamos e descrevemos esses padrões, podemos utilizá-los na nossa linguagem de programação.

Em C#, utilizamos regex ao trabalhar com a classe `System.Text.RegularExpressions`. Ela possui várias funcionalidades que permitem validar, buscar ou substituir dados em textos, manipulando strings de forma sofisticada.

# Padrões de Chaves PIX

Imagine que você quer descrever os padrões associados a chaves pix. Conseguimos identificá-los visualmente:

- CPF:  XXX.XXX.XXX-XX
- CNPJ: XX.XXX.XXX/XXXX-XX
- Telefone: (XX)XXXXX-XXXX
- Email: xxxxxxxx@xxx.xx

Mas para representar esses padrões com código, precisaremos utilizar símbolos especiais

# Caracteres especiais

| Símbolo | Descrição |
|---|---|
| . | Caractere, exceto quebra de linha |
| \d | Dígito (0-9) |
| \D | Caractere que não é um dígito |
| \w | Caractere alfanumérico |
| \W | Caractere não alfanumérico |
| \s | Espaço em branco |
| \S | Caractere que não é espaço em branco |
| ^ | Início da string |
| $ | Fim da string |

Ao lado, temos caracteres especiais que representam padrões. Por exemplo, se eu uso um “\d”, estou indicando para o meu regex que tenho um caractere que é algum número de 0 a 9. Toda vez que queremos usar um caractere especial, devemos escapá-lo com “\”

# Quantificadores

| Símbolo | Descrição |
|---|---|
| * | 0 ou mais ocorrências do padrão anterior |
| + | 1 ou mais ocorrências do padrão anterior |
| ? | 0 ou 1 ocorrência do padrão anterior |
| {n} | Exatamente n ocorrências do padrão anterior |
| {n,} | n ou mais ocorrências do padrão anterior |
| {n,m} | Entre n e m ocorrências do padrão anterior |

Já nessa tabela, temos quantificadores, que vão indicar quantas vezes um padrão pode ocorrer. Por exemplo: ao invés de escrever “\d\d\d”, representando um número com 3 dígitos, podemos escrever “\d{3}”.

# Padrões de Chaves PIX

Utilizando os caracteres especiais e quantificadores, já conseguimos descrever alguns dos padrões de chaves pix:

- CPF - XXX.XXX.XXX-XX ⇒ `^\d{3}\.\d{3}\.\d{3}-\d{2}$`
- CNPJ - XX.XXX.XXX/XXXX-XX ⇒ `^\d{2}\.\d{3}\.\d{3}/d{4}-\d{2}$`
- Telefone: (XX)XXXXX-XXXX ⇒ `^\(\d{2}\)\d{4,5}-\d{4}$`

Repare que sempre indicamos o início (^) e fim($) da string. A maioria dos padrões usa somente os caracteres especiais e quantificadores. Quando queremos representar exatamente o caractere “.” , devemos utilizá-lo escapado com “\”, já que ele sozinho é um caractere especial.

# Classe de caracteres

| Símbolo | Descrição |
|---|---|
| [abc] | Qualquer caractere dentro dos colchetes ('a', 'b' ou 'c') |
| [^abc] | Qualquer caractere exceto os que não estejam dentro dos colchetes |
| [a-z] | Qualquer caractere minúsculo de 'a' a 'z' |
| [A-Z] | Qualquer caractere maiúsculo de 'A' a 'Z' |
| [0-9] | Qualquer dígito |
| [a-zA-Z] | Qualquer letra maiúscula ou minúscula |

Em muitos casos, não sabemos a quantidade exata de símbolos do nosso padrão, mas sabemos que existe uma certa estrutura nele. Por exemplo, não sabemos o tamanho exato de um email, mas sabemos que ele terá um “@” no meio. Para que seja possível descrever essas situações, usamos as classes de caracteres.

# Padrões de Chaves PIX

Agora sim! Conseguiremos representar o padrão do email com as classes de caracteres.

- Email: xxxxxxxx@xxx.xx ⇒ **^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$**

Podemos descrever o padrão da seguinte forma:

- ❏ Ele tem um início (^) e um fim($);
- ❏ É dividido em 3 partes: o nome do usuário (o que aparece antes do @), o próprio caractere “@” e o domínio (o que vem depois do @);
- ❏ No nome de usuário, `[a-zA-Z0-9._%+-]` permite letras maiúsculas e minúsculas, números, ponto (.), underline (_), porcentagem (%), mais (+) e hífen (-). Já o `+` indica que um ou mais caracteres dessa lista são obrigatórios.

# Padrões de Chaves PIX

`^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$`

- O domínio é composto de três partes:
  - o nome do domínio (gmail. outlook, empresa), representado por `[a-zA-Z0-9.-]+`. Permite letras, números, ponto (.) e hífen (-). O `+` novamente exige um ou mais caracteres.
  - o próprio ponto separador (`\.`)
  - a extensão do domínio, representada por `[a-zA-Z]{2,}`. Ela permite apenas letras, e o `{2,}` indica que a extensão deve ter pelo menos 2 caracteres (não define limite superior, então .com, .info, .education são válidos).

# Métodos da classe Regex

Para manipular regex em C#, podemos utilizar diversos métodos estáticos da classe Regex, listados abaixo.

| Método | Descrição |
|---|---|
| IsMatch() | Verifica se um padrão existe na string. |
| Match() | Retorna a primeira correspondência do padrão. |
| Matches() | Retorna todas as correspondências do padrão. |
| Replace() | Substitui ocorrências do padrão por uma nova string. |
| Split() | Divide uma string com base em um padrão. |

# Uso da classe Regex

```
string email = "iasmin@alura.com.br";
string padraoEmailAgrupado = @"(^[a-zA-Z0-9._%+-]+)@([a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$)";

string dominio = Regex.Match(email, padraoEmailAgrupado).Groups[2].Value;
Console.WriteLine(dominio);
```

Ao utilizarmos parênteses, agrupamos os padrões identificados e conseguimos recuperar os grupos posteriormente, usando `Groups[1]`, `Groups[2]`... Nesse exemplo, separamos a expressão em dois grupos: o que vinha antes e o que vinha depois do email, e recuperamos apenas o grupo 2, que corresponde ao domínio do email.

# Uso da classe Regex

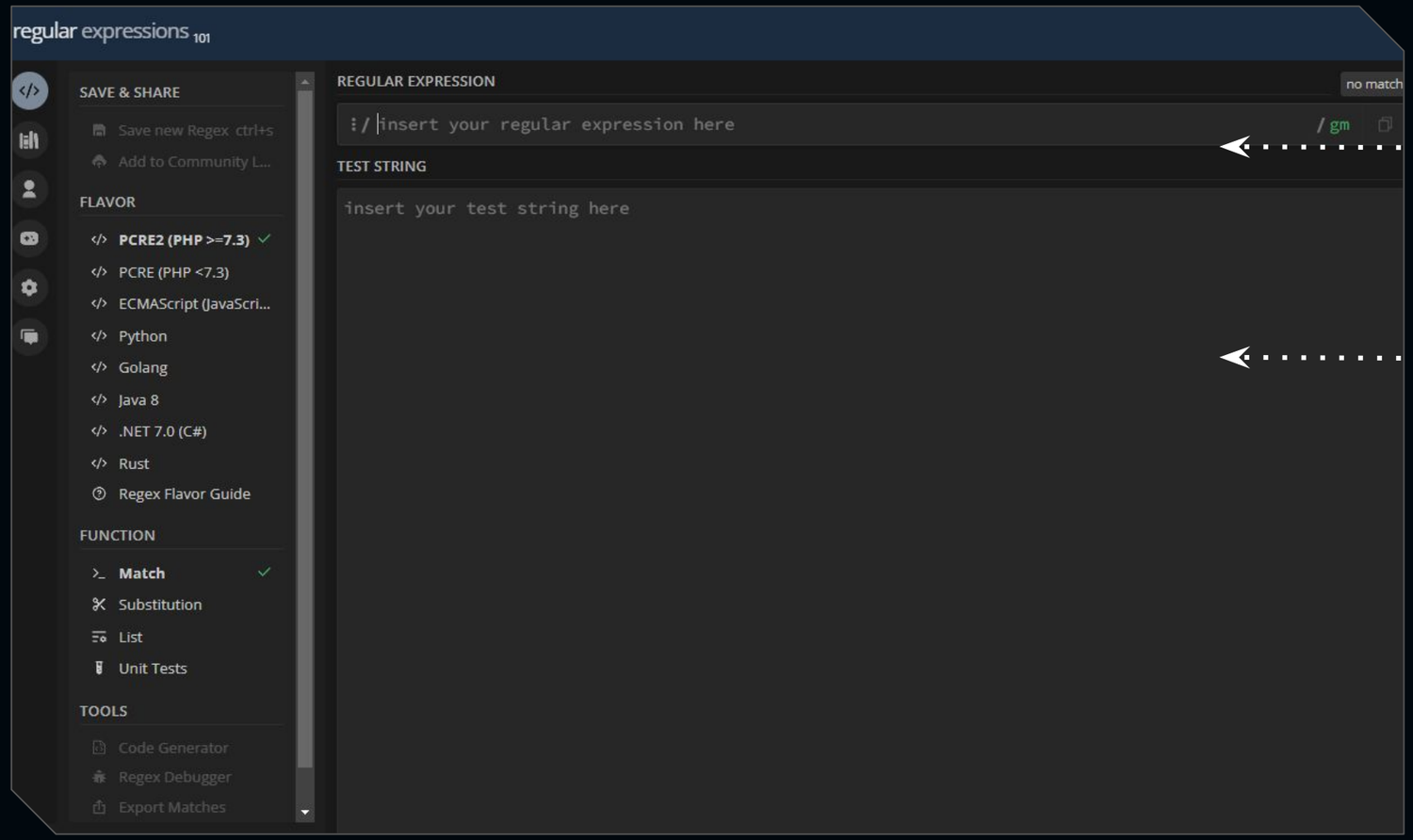
```
string chavePix = Console.ReadLine();

string padraoCPF = @"^\d{3}\.\d{3}\.\d{3}-\d{2}$";
string padraoCNPJ = @"^\d{2}\.\d{3}\.\d{3}/\d{4}-\d{2}$";
string padraoTelefone = @"^\(\d{2}\)\d{4,5}-\d{4}$";
string padraoEmail = @"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$";

string tipoChave;

if (Regex.IsMatch(chavePix, padraoCPF))
    tipoChave = "CPF";
else if (Regex.IsMatch(chavePix, padraoCNPJ))
    tipoChave = "CNPJ";
else if (Regex.IsMatch(chavePix, padraoTelefone))
    tipoChave = "Telefone";
else if (Regex.IsMatch(chavePix, padraoEmail))
    tipoChave = "E-mail";
else
    tipoChave = "Formato inválido";

Console.WriteLine($"Tipo da chave PIX: {tipoChave}");
```

No código, estamos utilizando o método `IsMatch` da classe Regex. Assim, verificamos se a string digitada corresponde a algum dos padrões de chaves pix descritos no código.

alura

# Ferramenta

Podemos usar ferramentas como o Regex101 para testar as expressões regulares antes de passá-las efetivamente para o código.




Campo para inserir uma regex (expressão regular)

Campo para inserir sua string (e descrever os padrões dela)

alura

Compartilhe um resumo de seus novos conhecimentos em suas redes sociais.

**#aprendizadoalura**