1. Назовите классы, которые используются для создания меню. Перечислите свойства и методы.

MenuStrip ToolStripMenuItem ContextMenuStrip:

Свойства:

* Dock – прикрепляет меню к одной из сторон формы
* LayoutStyle – задает ориентацию панели меню на форме
* ShowItemToolTips – указывает, будут ли отображаться всплывающие подсказки отдельных элементов меню
* Strech – позволяет растянуть панель по всей длине контейнера
* TextDirection – задает направление текста в пунктах меню
* DropDownItems: Получает коллекцию элементов выпадающего меню.
* ShortcutKeys: Получает или задает комбинацию клавиш, используемую в качестве быстрого сочетания клавиш для доступа к элементу меню.
* CheckOnClick – при тру позволяет по клику отметить пункт меню
* Checked – будет ли пункт меню отмечен при запуске программы
* CheckState – отмечен ли пункт или нет
* DisplayStyle – будет ли отображаться на элементе текст или изображение или и то и то
* Image – само изображение

Методы:

* Add: Добавляет элемент в коллекцию элементов меню, выпадающего или контекстного меню.
* Remove: Удаляет указанный элемент из коллекции элементов меню, выпадающего или контекстного меню.

1. Что может содержать строка состояния? Какие есть методы управления строкой состояния?

Информацию о состоянии приложения, такую как текущее действие, прогресс выполнения задачи, сообщения об ошибках и др. Методы управления строкой состояния - установку текста, цвета фона, отображения, скрытия строки, добавления, удаления.

1. Что такое регулярные выражения? Где и как их можно использовать?

Регулярные выражения - это последовательности символов, используемые для поиска и сопоставления шаблонов в тексте. Они могут использоваться для проверки и фильтрации текста, поиска и замены строк, извлечения данных из структурированных документов и многое другое.

1. Что такое привязки (якоря) в RegEx? Приведите примеры.

Привязки (якоря) в RegEx - это символы, используемые для указания позиции в тексте, к которой должен быть применен шаблон. Примеры привязок включают "^", который указывает на начало строки, и "$", который указывает на конец строки.

1. Зачем используют конструкции группирования? Приведите примеры

Конструкции группирования позволяют группировать части шаблона в отдельные блоки, чтобы применять к ним определенные операции или повторения. Например, группирование может быть использовано для создания альтернативных вариантов сопоставления, таких как "apple|orange", или для извлечения подстрок из текста, которые соответствуют определенному шаблону.

1. Что такое квантор или множители? Приведите примеры

Кванторы (множители) в RegEx используются для указания количества повторений для частей шаблона. Например, "\*" означает, что предыдущий символ или группа могут быть повторены ноль или более раз, а "+" означает, что они должны быть повторены один или более раз.

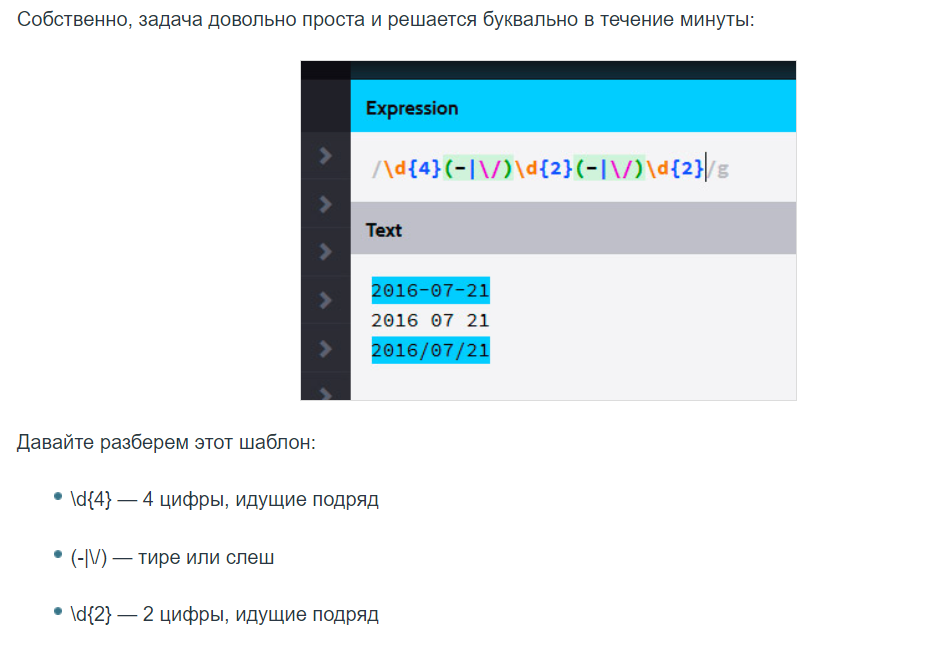
1. Напишите регулярное выражение для проверки номера MTC (Velcom)

^375(25|29|33|44)\d{7}$

Это регулярное выражение начинается с символов **^375**, которые соответствуют коду страны Беларусь, и затем следует один из четырех возможных кодов операторов мобильной связи - **25**, **29**, **33** или **44**. Далее должны идти семь цифр для номера телефона. Символ **$** означает конец строки.

1. Напишите регулярное выражение для проверки паспорта.

Регулярное выражение для проверки паспорта может зависеть от страны, которая выпускает паспорт. Для паспорта Республики Беларусь можно использовать следующее выражение: ^[A-Z]{2}\d{7}$

1. Напишите регулярное выражение для проверки даты.
2. Напишите регулярное выражение для проверки УДКАа

^\d{3}\.\d{4}$

1. Напишите регулярное выражение для проверки Фамилии Ааа

^[a-z]{2,}$

1. Напишите регулярное выражение для проверки пароля, логина.

/(?=.\*[0-9])(?=.\*[!@#$%^&\*])(?=.\*[a-z])(?=.\*[A-Z])[0-9a-zA-Z!@#$%^&\*]{6,}/g

Вот пример на [**regex101**](https://regex101.com/r/dT8sD6/1). Можете попробовать написать свои пароли и проверить работу регулярного выражения на соответствие своим требованиям.

Пояснение:

* (?=.\*[0-9]) - строка содержит хотя бы одно число;
* (?=.\*[!@#$%^&\*]) - строка содержит хотя бы один спецсимвол;
* (?=.\*[a-z]) - строка содержит хотя бы одну латинскую букву в нижнем регистре;
* (?=.\*[A-Z]) - строка содержит хотя бы одну латинскую букву в верхнем регистре;
* [0-9a-zA-Z!@#$%^&\*]{6,} - строка состоит не менее, чем из 6 вышеупомянутых символов.

Логин:

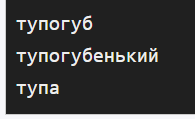
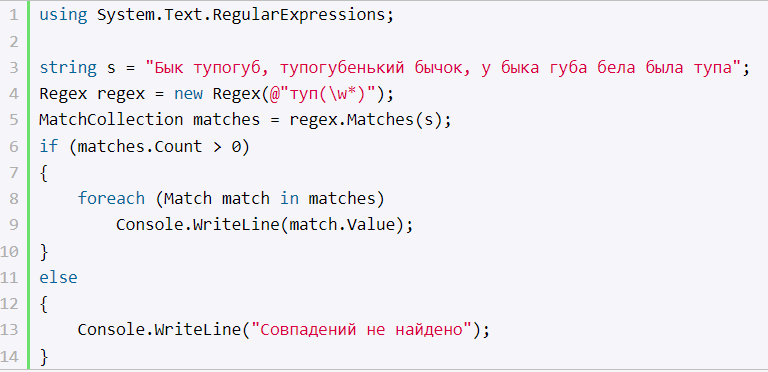


Пароль:



Регулярные выражения представляют эффективный и гибкий метод по обработке больших текстов, позволяя в то же время существенно уменьшить объемы кода по сравнению с использованием стандартных операций со строками.

**System.Text.RegularExpressions**. класс Regex



Здесь мы находим в искомой строке все словоформы слова "туп". В конструктор объекта Regex передается регулярное выражение для поиска. Далее мы разберем некоторые элементы синтаксиса регулярных выражений, а пока достаточно знать, что выражение туп(\w\*) обозначает, найти все слова, которые имеют корень "туп" и после которого может стоять различное количество символов.

Метод Matches класса Regex принимает строку, к которой надо применить регулярные выражения, и возвращает коллекцию найденных совпадений.

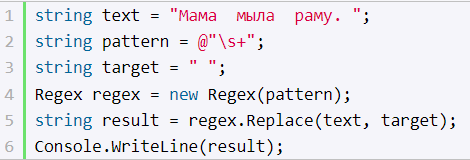
Каждый элемент такой коллекции представляет объект **Match**. Его свойство Value возвращает найденное совпадение.

* **^**: соответствие должно начинаться в начале строки (например, выражение @"^пр\w\*" соответствует слову "привет" в строке "привет мир")
* **$**: конец строки (например, выражение @"\w\*ир$" соответствует слову "мир" в строке "привет мир", так как часть "ир" находится в самом конце)
* **.**: знак точки определяет любой одиночный символ (например, выражение "м.р" соответствует слову "мир" или "мор")
* **\***: предыдущий символ повторяется 0 и более раз
* **+**: предыдущий символ повторяется 1 и более раз
* **?**: предыдущий символ повторяется 0 или 1 раз
* **\s**: соответствует любому пробельному символу
* **\S**: соответствует любому символу, не являющемуся пробелом
* **\w**: соответствует любому алфавитно-цифровому символу
* **\W**: соответствует любому не алфавитно-цифровому символу
* **\d**: соответствует любой десятичной цифре
* **\D**: соответствует любому символу, не являющемуся десятичной цифрой

Класс Regex предоставляет статический метод **IsMatch**, который позволяет проверить входную строку с шаблоном на соответствие:

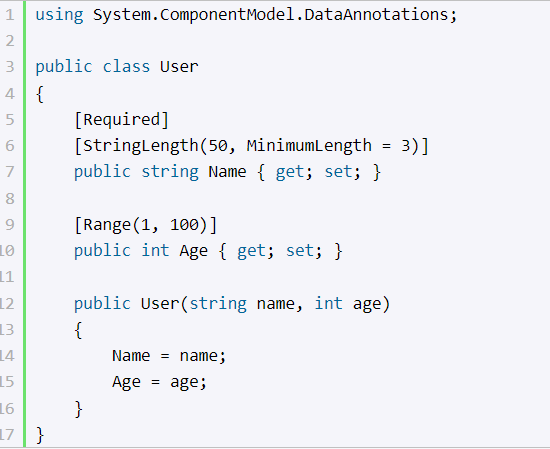


Класс Regex имеет метод Replace, который позволяет заменить строку, соответствующую регулярному выражению, другой строкой:

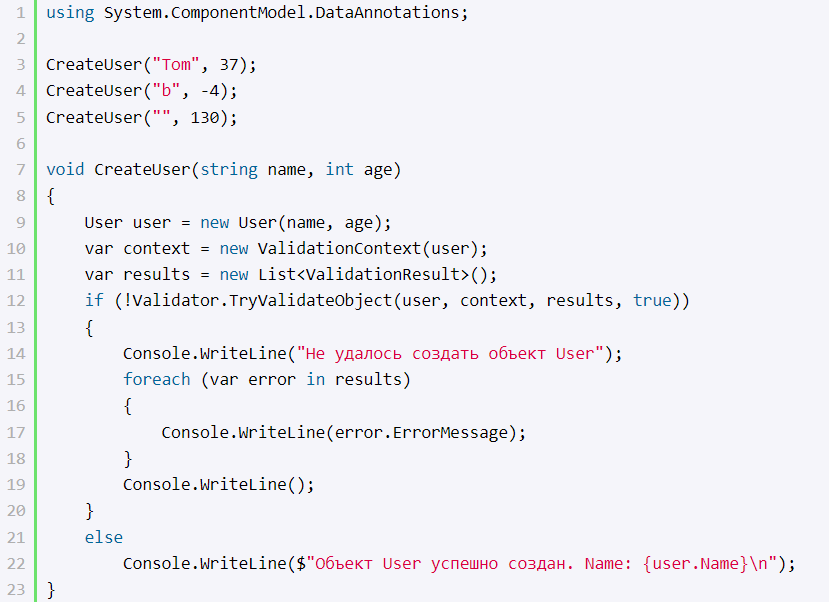


Валидация:

**System.ComponentModel.DataAnnotations**.



В данном случае используются три атрибута: классы RequiredAttribute, StringLengthAttribute и RangeAttribute. В коде необязательно использовать суффикс Attribute, поэтому он, как правило, отбрасывается. Атрибут Required требует обзательного наличия значения. Атрибут StringLength устанавливает максимальную и минимальную длину строки, а атрибут Range устанавливает диапазон приемлемых значений.



Вначале мы создаем контекст валидации - объект **ValidationContext**. В качестве первого параметра в конструктор этого класса передается валидируемый объект, то есть в данном случае объект User.

Собственно валидацию производит класс **Validator** и его метод TryValidateObject(). В этот метод передается валидируемый объект (в данном случае объект user), контекст валидации, список объектов **ValidationResult** и булевый параметр, который указывает, надо ли валидировать все свойства.

Если метод Validator.TryValidateObject() возвращает false, значит объект не проходит валидацию. Если модель не проходит валидацию, то список объектов ValidationResult оказывается заполенным. А каждый объект ValidationResult содержит информацию о возникшей ошибке. Класс ValidationResult имеет два ключевых свойства: MemberNames - список свойств, для которых возникла ошибка, и ErrorMessage - собственно сообщение об ошибке.

Для валидации модели мы можем использовать большой набор встроенных атрибутов. Все эти атрибуты представляют классы, унаследованные от класса **ValidationAttribute**

Имеется довольно большое количество атрибутов. Основные из них:

* **Required**: данный атрибут указывает, что свойство должно быть обязательно установлено, обязательно должно иметь какое-либо значение. Если свойство имеет значение null, то оно не проходит валидацию. Также не проходят валидацию свойства, которые представляют тип string, и которым присваивается пустая строка.

Из его свойств следует отметить свойство AllowEmptyStrings. Если оно имеет значение true, то для строковых свойств разрешено использовать пустые строки:

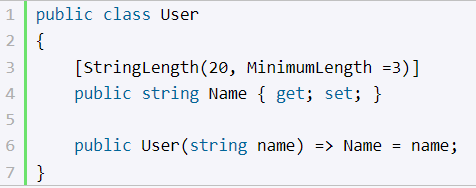


**RegularExpression**: указывает на регулярное выражение, которому должно соответствовать значение свойства. Например, пусть у пользователя определено свойство Phone, которое хранит номер телефона:

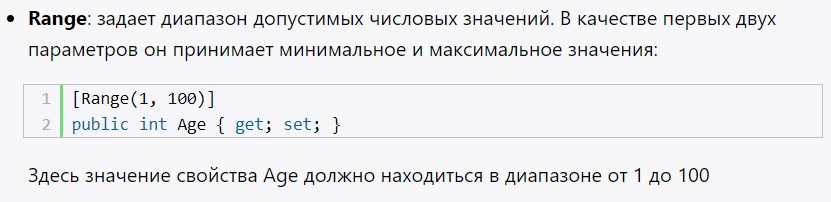


В этом случае номер телефона должен быть в формате "+xxxx-xxx-xxxx", например, +1111-111-2345.

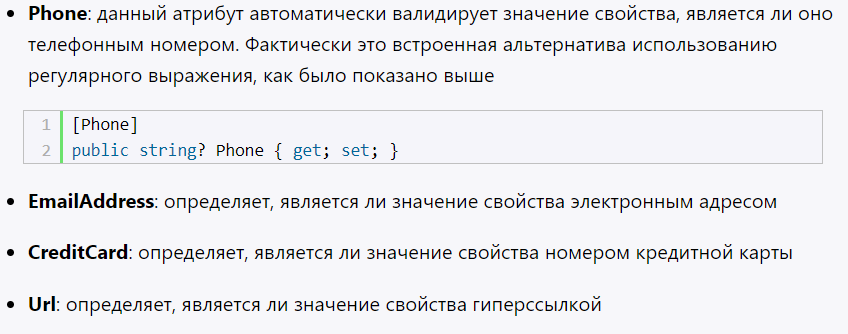
**StringLength**: определяет допустимую длину для строковых свойств. В качестве первого параметра он принимает максимально допустимую длину строки. С помощью дополнительного свойства MinimumLength можно установить минимально допустимую длину строки



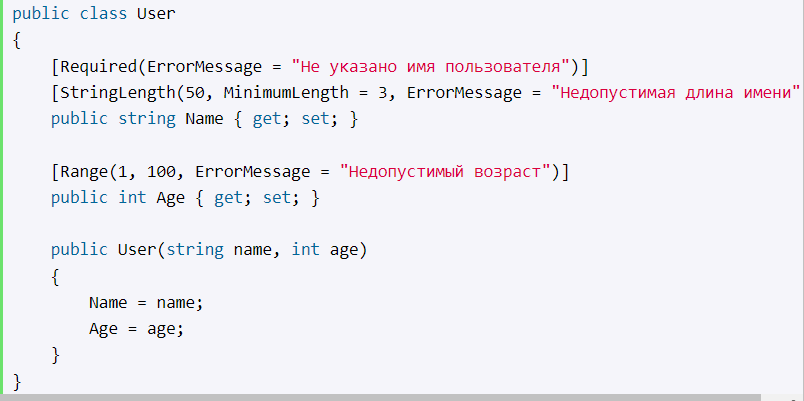
В данном случае значение свойства Name должно иметь длину как минимум в 3 символа и максимум в 20 символов.



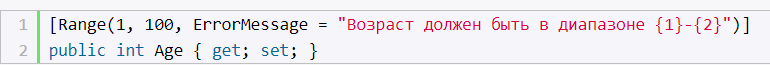




Класс **ValidationAttribute** определяет для атрибутов ряд общих свойств и методов, из которых следует выделить свойство **ErrorMessage**. Это свойство позволяет задать сообщение обошибке. 



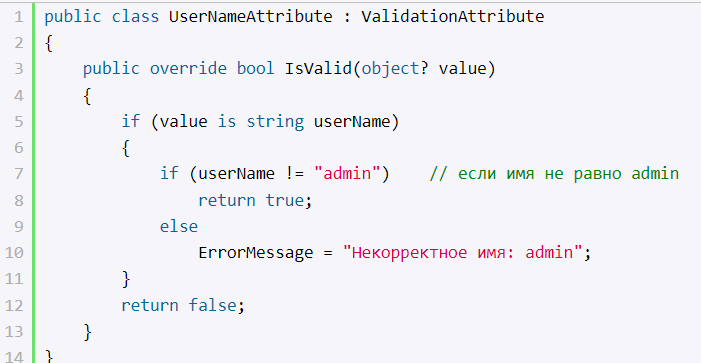
Или



Здесь параметр {0} представлял бы имя свойства - "Age". {1} здесь - первый параметр атрибута - минимальное значение - 1. {2} - максимальное значение 100.

Для создания атрибута нам надо унаследовать свой класс от класса **ValidationAttribute** и реализовать его метод IsValid().

**Атрибут уровня свойства**

****

Данный атрибут будет применяться к строковому свойству, поэтому в метод IsValid(object? value) в качестве value будет передаваться строка. Поэтому в ходе программы мы можем преобразовать значение value к строке.

****

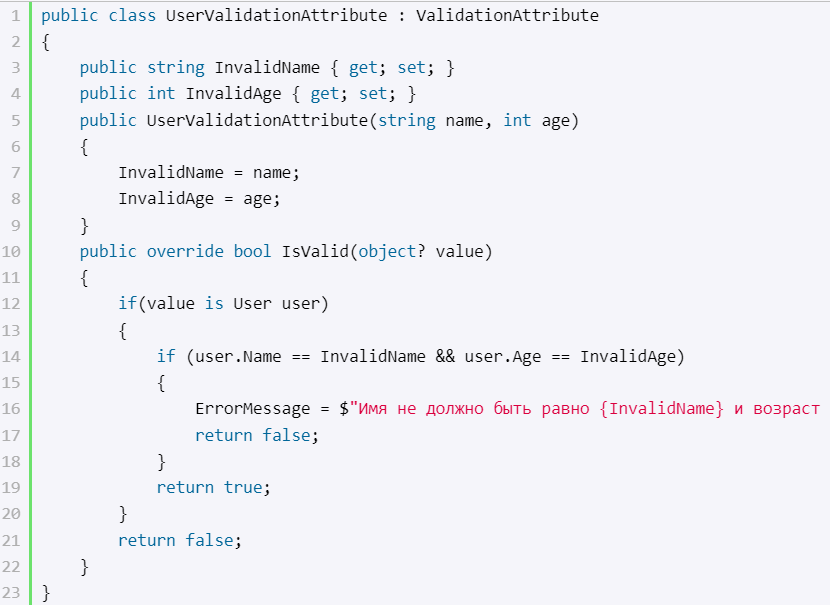
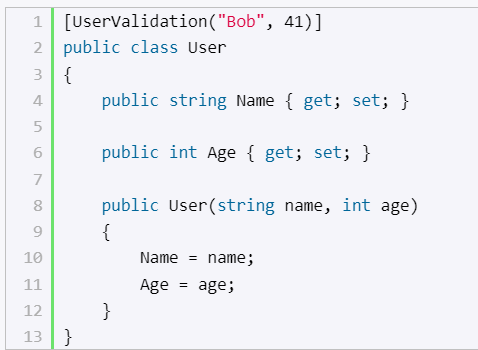
Также, как и другие атрибуты, мы можем использовать эти атрибуты без суффикса **Attribute**.

### Атрибуты валидации уровня класса

****

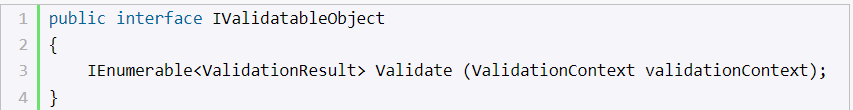
****

### Передача в атрибут значений

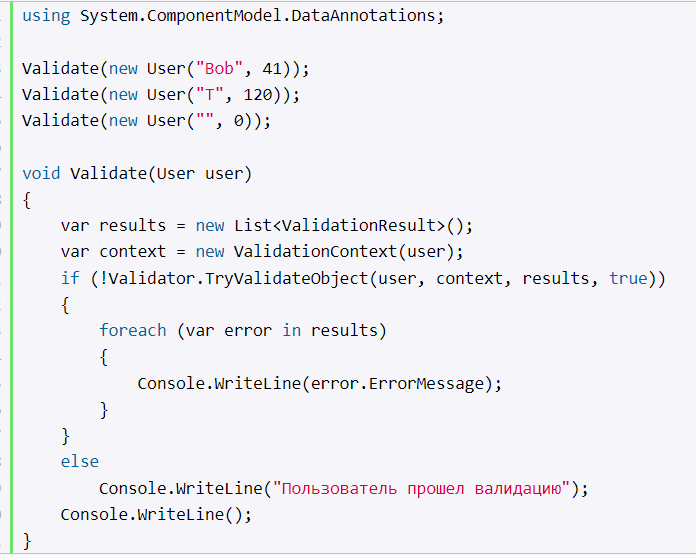
**** ****

## Самовалидация модели

Нам необязательно определять правила валидации модели в виде атрибутов. Мы можем применить к классу интерфейс **IValidatableObject** и реализовать его метод Validate():

****

Метод Validate в качестве параметра получает объект ValidationContext, который собственно и проводит непосредственную валидацию. В качестве результата метод должен возвращать коллекцию объектов ValidationResult, которые представляют результат валидации.

**** ****



"[a-zA-Zа-яА-Я,-;:]{5,50}"

/^[a-z0-9\_-]{3,16}$/

/^#?([a-f0-9]{6}|[a-f0-9]{3})$/

/^(https?:\/\/)?([\da-z\.-]+)\.([a-z\.]{2,6})([\/\w \.-]\*)\*\/?$/

/^(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0- 9]?)$/

/^<([a-z]+)([^(.\*)|\s+\/>)$/

