In Angular, **validators** are functions used to enforce rules on form controls, ensuring that user input meets specific criteria. Angular provides both **built-in validators** and the ability to create **custom validators**.

Built-in Validators

Angular has several predefined validators that can be applied to form controls:

- `Validators.required`: Ensures the field is not empty.
- `Validators.min(length)` / `Validators.max(length)`: Sets minimum and maximum values.
- `Validators.minLength(length)` / `Validators.maxLength(length)`: Defines character length constraints.
- `Validators.email`: Validates email format.
- `Validators.pattern(regex)`: Ensures input matches a specific pattern.

Custom Validators

If built-in validators don't meet your needs, you can create custom ones. A custom validator is a function that returns an error object if validation fails or `null` if it passes. For example:

import { AbstractControl, ValidationErrors, ValidatorFn } from '@angular/forms';

```
export function passwordValidator(): ValidatorFn {
        return    (control:    AbstractControl):
ValidationErrors | null => {
    const value = control.value;
    const hasUppercase = /[A-Z]/.test(value);
    const hasNumber = /[0-9]/.test(value);
     return hasUppercase && hasNumber ? null : {
passwordStrength: true };
  };
}
```

Applying Validators

Validators can be applied in **Reactive Forms** like this:

typescript

```
import   {   FormControl,   Validators   }   from
'@angular/forms';

const   emailControl   =   new   FormControl('',
[Validators.required, Validators.email]);


export class AppComponent {
  passwordForm: FormGroup;


  constructor(private fb: FormBuilder) {

    this.passwordForm = this.fb.group({

              password:  ['',  [Validators.required,
passwordValidator()]],

      confirmPassword: ['', Validators.required]

      }, { validator: this.passwordMatchValidator
});

  }


  passwordMatchValidator(form: FormGroup) {

    const password = form.get('password')?.value;

                       const     confirmPassword    =
form.get('confirmPassword')?.value;

    return password === confirmPassword ? null : {
passwordMismatch: true };

  }
```

```
onSubmit() {

  if (this.passwordForm.valid) {

    alert('Form Submitted Successfully');

  } else {

    alert('Please check the form for errors');

  }

}
```

In Angular, these terms are used to describe the state of form controls:

- **Touched**: A form control is considered "touched" when the user has interacted with it, such as clicking or focusing on it and then moving away. This helps in triggering validation messages when the user has left an input field without entering valid data.

- **Untouched**: A form control remains "untouched" if the user has not interacted with it yet. This is useful for determining whether a field should display validation errors immediately or wait until the user interacts with it.

- **Dirty**: A form control becomes "dirty" when the user has modified its value. This helps track whether the user has made changes to the form.

These states are useful for managing form validation and user interactions in Angular applications. You can read more about them here. Would you like an example of how to use these states in code?