Advanced Application Development (CSE-214)

week-5 : Angular forms, Dependency Injection, RxJS

Dr. Alper ÖZCAN

Akdeniz University

alper.ozcan@gmail.com

What Does Immutability Mean?

```
public class Example {
    public static void main(String[] args) {
        String s = "Immutable";
        s.replace("table", "able");
        System.out.println(s);
    }
}
```

What Does Immutability Mean?

Which of the following statements about `String` immutability is **true**?

A) `String` objects cannot be changed after creation.

B) `String` allows modification through `append()`.

C) `String` is mutable if declared as `final`.

D) `String` objects change their value in place.

What Does Immutability Mean?

When you create a String, it is **stored in memory**.

If you change the value of the String, **a new object is created**, and the reference is updated.

The **original String** "Hello" **remains unchanged**.

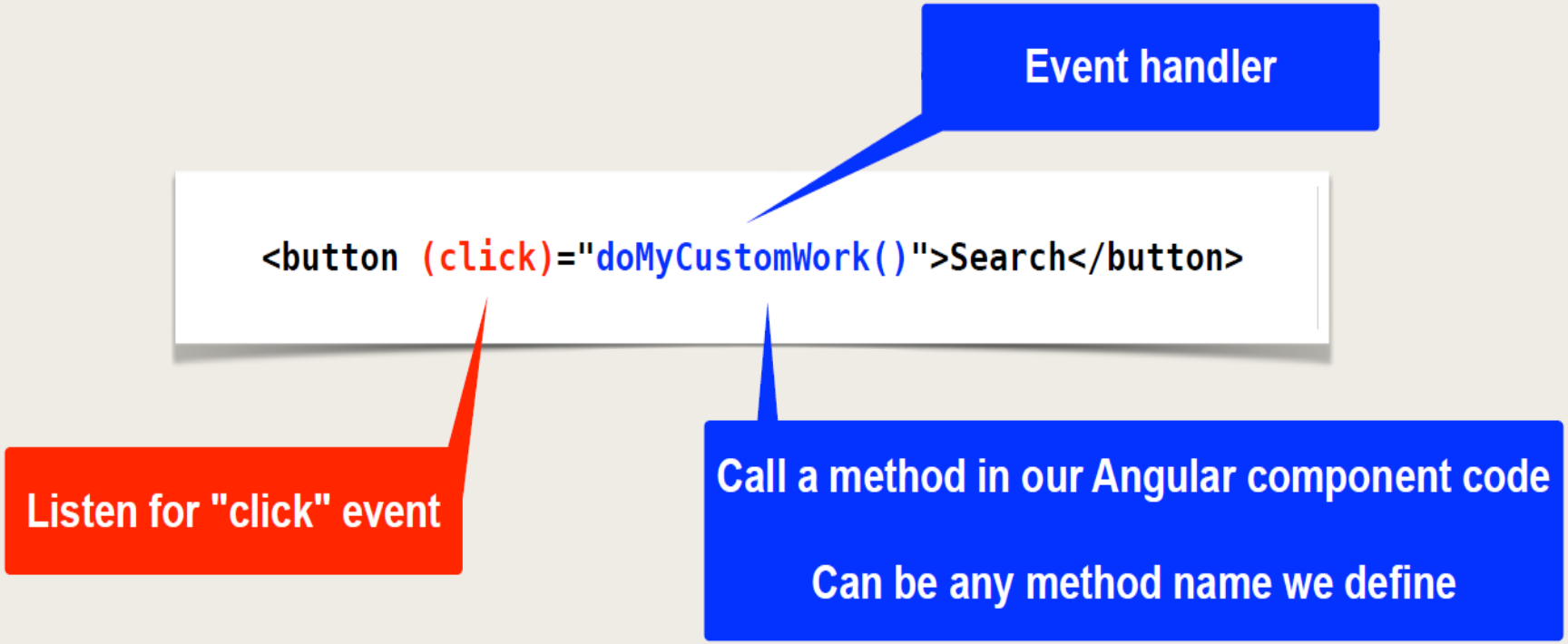
```
String str = "Hello";
str = str + " World"; // A new String object is created
```

Difference between String and StringBuilder

- String is **immutable**: Every modification creates a **new object**.
- StringBuilder is **mutable**: Modifications occur on the **same object**, improving performance.

Event Binding

- In Angular, you can listen for events with "event binding"



Event Binding

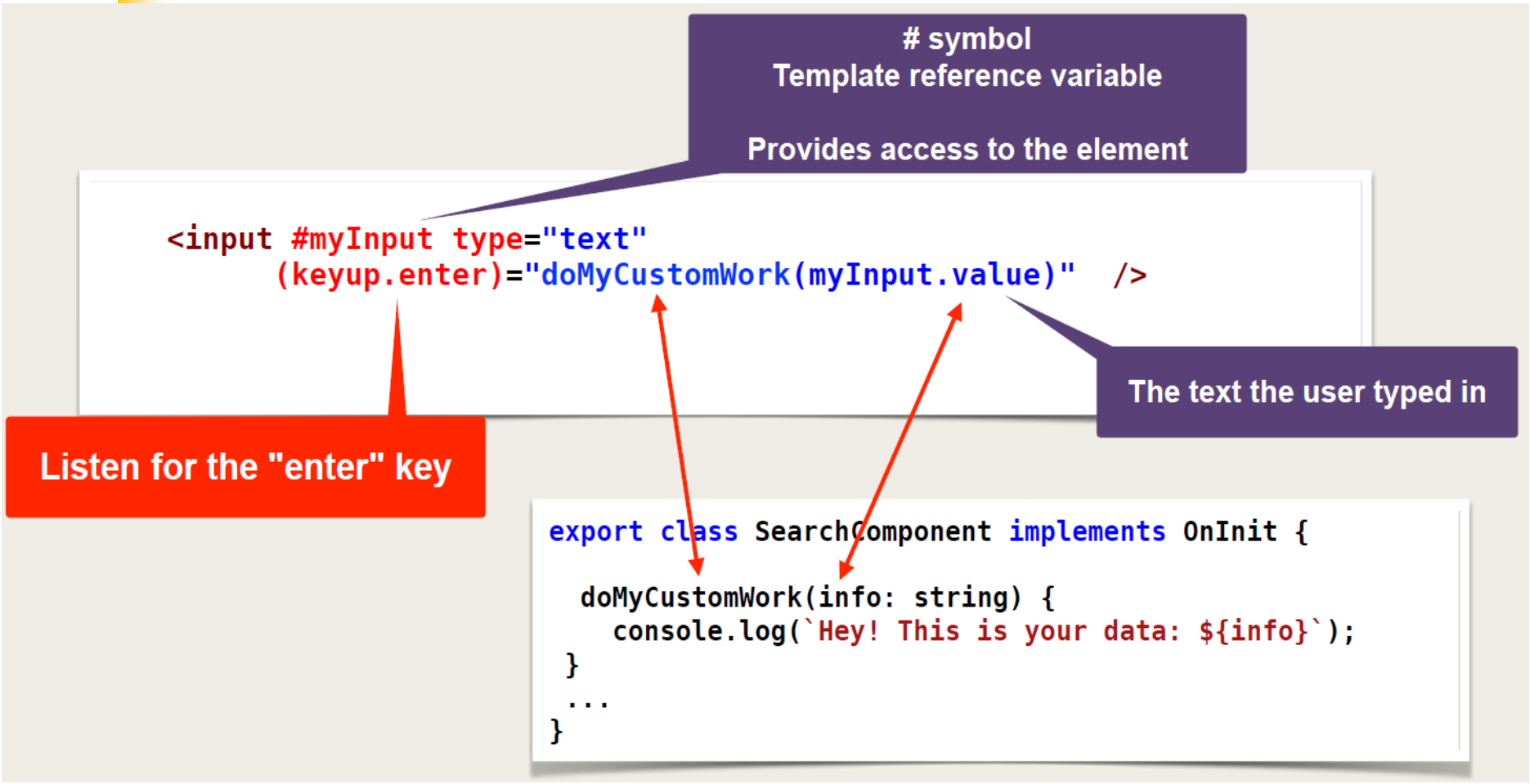
Call a method in our Angular component code

```
<button (click)="doMyCustomWork()">Search</button>
```

Listen for "click" event

```
export class SearchComponent implements OnInit {

  doMyCustomWork() {
    console.log(`Hey! You pushed my button!`);
  }

  ...
}
```

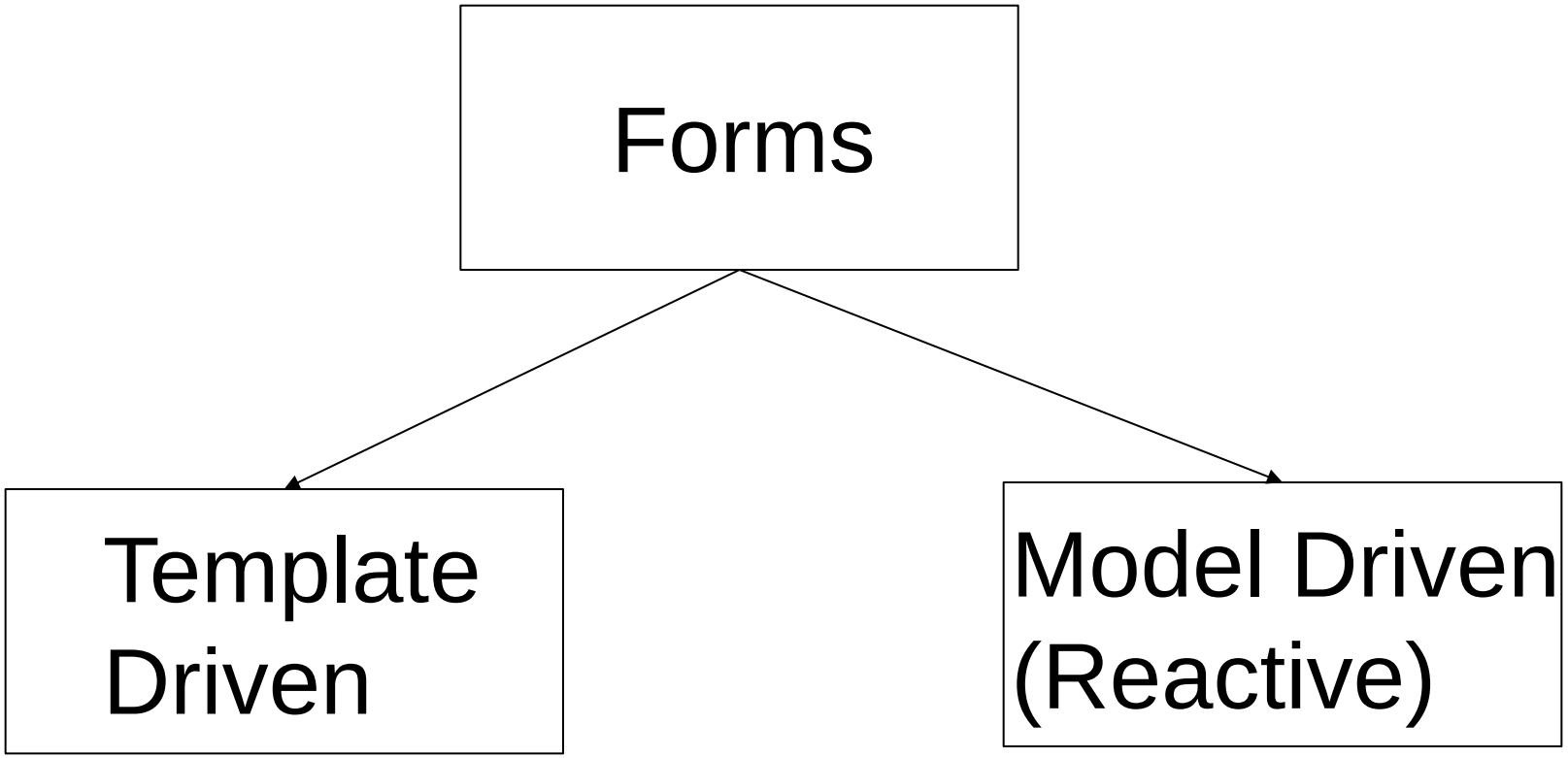
Event Binding



Angular Forms

- Forms play a crucial role in web applications, enabling users to input data.
- Handling user input with forms is the cornerstone of many common applications. Applications use forms to enable (1) **users to log in**, (2) **to update a profile**, (3) **to enter sensitive information**, and (4) **to perform many other data-entry tasks**.
- Angular provides two different approaches to handling user input through forms: **Model Driven (Reactive)** and **Template-driven**.
- Both **(1) capture user input events from the view, (2) validate the user input, (3) create a form model and (4) data model to update, and (5) provide a way to track changes**.

Angular Forms

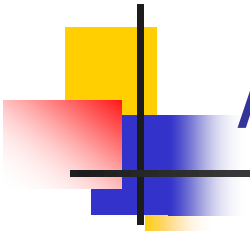


Angular provides two ways to create forms: **Template-Driven** and **Model-Driven (Reactive)**

Angular Forms

•**Model-Driven (Reactive) forms** provide direct, explicit access to the underlying forms object model. Compared **to template-driven forms**, they are **more robust and more scalable, reusable, and testable**. If **forms** are a **key part of your application**, or you're already using reactive patterns for building your application, use reactive forms.

•**Template-driven forms** rely on **directives** in the **template** to **create** and **manipulate the underlying object model**. They are useful for **adding a simple form to an app**, such as an **email list signup form**.



Angular Forms

Angular provides two ways to manage forms:

1. **Template-driven Forms**
    1. Uses Angular directives in the template (HTML) to manage form data.
    2. Relies on two-way data binding **([(ngModel)])** to update the component model.
    3. **Simpler** and **suitable** for **basic forms**.
    4. Uses **FormsModule**.

2. **Reactive (Model-driven) Forms**
    1. Uses **FormGroup, FormControl, and FormArray** to manage form state and validation.
    2. The form logic is defined in the TypeScript component instead of the template.
    3. Provides better **scalability** and **testability**.
    4. Uses **ReactiveFormsModule**.

Template-driven Forms Example

```html
<!-- app.component.html -->
<form #myForm="ngForm" (ngSubmit)="onSubmit(myForm)">
  <input type="text" name="username" [(ngModel)]="user.username" required />
  <button type="submit" [disabled]="myForm.invalid">Submit</button>
</form>
```

```typescript
// app.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  user = { username: '' };

  onSubmit(form: any) {
    console.log(form.value);
  }
}
```

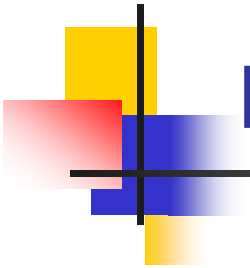
- Uses [(ngModel)] for two-way data binding
- #myForm="ngForm" creates a reference to the form
- The ngForm directive tracks form state and validity.

Reactive (Model-driven) Forms Example

```html
<!-- app.component.html -->
<form [formGroup]="userForm" (ngSubmit)="onSubmit()">
  <input type="text" formControlName="username" />
  <button type="submit" [disabled]="userForm.invalid">Submit</button>
</form>
```

```ts
// app.component.ts
import { Component } from '@angular/core';
import { FormGroup, FormControl, Validators } from '@angular/forms';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  userForm = new FormGroup({
    username: new FormControl('', Validators.required)
  });

  onSubmit() {
    console.log(this.userForm.value);
  }
}
```

- Uses **FormGroup** to group form controls.
- Uses **FormControl** to define individual fields.
- No reliance on **ngModel**, keeping the data model pure
- **Form state** and **data** are **managed separately**.
- The form exists as an **immutable object** (**FormGroup** instance), and updates only happen when we **explicitly retrieve the values** (this.userForm.value).
- Angular **does NOT modify the component state directly**.



Reactive (Model-driven) Forms

No reliance on **ngModel**, keeping the data model pure

In **Reactive Forms**, Angular **does not use ngModel** for **binding form values to the component's data model**. Instead, the form is **controlled programmatically** using **FormGroup** and **FormControl**.

"keeping the data model pure" mean?

It means that the **data model (i.e., the component's state) is not directly modified by the UI**. Instead, **form updates** happen **immutably—each change** creates a **new state** rather than **mutating the existing one**.

How is it different from Template-driven Forms?

1. Template-driven Forms (Mutable)

- Uses **[(ngModel)]** (**two-way binding**).
- The model **(user.username)** is **directly updated** when the user types.
- This means UI changes immediately affect the component's data

```html
<input type="text" [(ngModel)]="user.username" />
```

```typescript
user = { username: '' };  // This gets updated directly from UI
```

How is it different from Template-driven Forms?

2. Reactive Forms (Immutable)

- Uses **FormControl** and **FormGroup**, without **ngModel**.
- The form **does not directly modify the model** in real time.
- Instead, changes are captured as an event (**this.userForm.value**), keeping the original object unchanged.

```html
<input type="text" formControlName="username" />
```

Better Performance: Angular doesn't need to track changes like with ngModel.

Scalability: Large applications benefit from unidirectional data flow, making it easier to manage complex forms.

```typescript
userForm = new FormGroup({
  username: new FormControl('')
});

onSubmit() {
  console.log(this.userForm.value);  // Form data is retrieved when needed
}
```

Form States in Angular

Angular tracks form states to help with validation and UI feedback.

Form State	Description
Pristine	The form/control has **not been modified** yet.
Dirty	The form/control has been **modified by the user.**
Untouched	The form/control **has not been focused (touched).**
Touched	The form/control has been **focused and blurred.**
Valid	The form/control **passes all validations.**
Invalid	The form/control **fails at least one validation.**

Example: Using Form States in UI

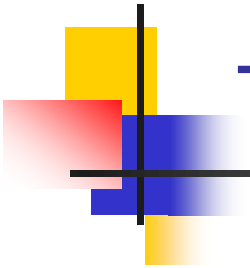
```html
<form [formGroup]="userForm">
  <input type="text" formControlName="username" />

  <div *ngIf="userForm.controls.username.touched && userForm.controls.username.invalid">
    <small>Username is required!</small>
  </div>
</form>
```

How Form States Work

1. Initially, the form is **pristine** and **untouched**.
2. If the user types something, the form becomes **dirty**.
3. If the user clicks in the field and then clicks away, it becomes **touched**.
4. If validation fails, the form becomes **invalid**.

Template-driven vs. Reactive Forms

Feature	Template-driven Forms	Reactive Forms
Form Model	Defined in the **template (HTML)**	Defined in the **component (TypeScript)**
Data Binding	Uses **two-way binding** (`[(ngModel)]`)	Uses **immutable form objects** (`FormGroup` , `FormControl`)
Validation	Uses **directives** like `required`	Uses **Validators** in TypeScript
Form State Tracking	Done automatically via `ngForm`	Manually controlled via `FormGroup`
Best For	Simple forms	Complex forms with dynamic fields
Performance	Less efficient due to two-way binding overhead	More efficient with immutable data



Template-driven vs. Reactive Forms

- **Use Template-driven Forms** for **simple** and **small** applications.

- **Use Reactive Forms** for **scalability, maintainability, and better control** over form state and validation.

- **Reactive Forms keep the data model pure** because:

  - The component state is **not mutated** directly by UI inputs.
  - Form updates happen **immutably**, and changes are explicitly **retrieved**.
  - This results in **predictable data flow** and **better performance**.

FormsModule (For Template-driven Forms)

- Provides directives like **ngForm** and **ngModel**.
- Must be imported in **app.module.ts**

```
import { FormsModule } from '@angular/forms';
@NgModule({
  imports: [FormsModule]
})
```

ngForm Directive

- Automatically creates a FormGroup for a <form> element.
- Tracks form state and validity.

```
<!-- app.component.html -->
<form #myForm="ngForm" (ngSubmit)="onSubmit(myForm)">
  <input type="text" name="username" [(ngModel)]="user.username" required />
  <button type="submit" [disabled]="myForm.invalid">Submit</button>
</form>
```

ngModel Directive

- Binds form input to a property in the component.

- Can be used in two ways:

  - **One-way binding**: [ngModel]="user.username" (data flows from the component to the template)

  - **Two-way binding**: [(ngModel)]="user.username" (data flows **both** ways—changes in the UI update the component, and vice versa)

```
<!-- app.component.html -->
<form #myForm="ngForm" (ngSubmit)="onSubmit(myForm)">
  <input type="text" name="username" [(ngModel)]="user.username" required />
  <button type="submit" [disabled]="myForm.invalid">Submit</button>
</form>
```

#myForm="ngForm"

- Creates a reference to the form.
- Allows access to form state and validation status.

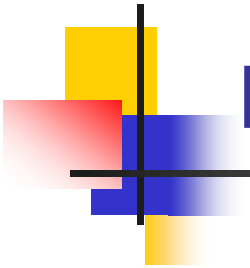
```
<!-- app.component.html -->
<form #myForm="ngForm" (ngSubmit)="onSubmit(myForm)">
  <input type="text" name="username" [(ngModel)]="user.username" required />
  <button type="submit" [disabled]="myForm.invalid">Submit</button>
</form>
```

FormGroup (For Reactive Forms)

- A collection of FormControls.

```
userForm = new FormGroup({
  username: new FormControl('')
});
```

```
// app.component.ts
import { Component } from '@angular/core';
import { FormGroup, FormControl, Validators } from '@angular/forms';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html'
})
export class AppComponent {
  userForm = new FormGroup({
    username: new FormControl('', Validators.required)
  });

  onSubmit() {
    console.log(this.userForm.value);
  }
}
```



FormControl

- Represents a single form input.
- Can have validators

```
username = new FormControl('', Validators.required);
```

FormBuilder Class

A more concise way to define FormGroup

```
import { FormBuilder } from '@angular/forms';

constructor(private fb: FormBuilder) {}

userForm = this.fb.group({
  username: ['']
});
```

FormArray

Manages a dynamic array of form controls

```
skills = new FormArray([
  new FormControl('Angular'),
  new FormControl('React')
]);
```

Template-driven Forms (Set up)

- To set up a Template Driven Form in Angular, follow these steps:
- 1. Import the **FormsModule** in your Angular module to enable template-driven forms.
- 2. Use the **ngForm** directive in the HTML form tag to declare the form.
- 3. Utilize various form input directives such as **ngModel** to bind form controls to properties in the component.

FormsModule

FormsModule is a core module that **provides** the **infrastructure** for **building** and **working** with **template-driven** forms. It's an essential part of Angular's form-handling capabilities

```
import { FormsModule } from '@angular/forms';
```

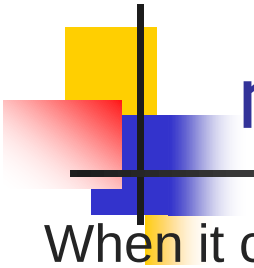
#myForm=”ngForm” creates a **reference to the form** and **assigns** it to the **variable** `**myForm**` which can be used in the component code.

ngForm directive

- The **ngForm directive** creates an Angular **form instance** and **provides form-related functionality**. It also tracks the **form's validation status** and **controls the submission behavior**.

```
<!-- app.component.html -->
<form #myForm="ngForm" (ngSubmit)="onSubmit(myForm)">
  <label for="name">Name:</label>
  <input type="text" id="name" name="name" ngModel required>

  <label for="email">Email:</label>
  <input type="email" id="email" name="email" ngModel required email>

  <button type="submit">Submit</button>
</form>
```

```
// app.component.ts
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  onSubmit(form: NgForm) {
    // Form submission logic
    console.log(form.value);
  }
}
```



ngModel

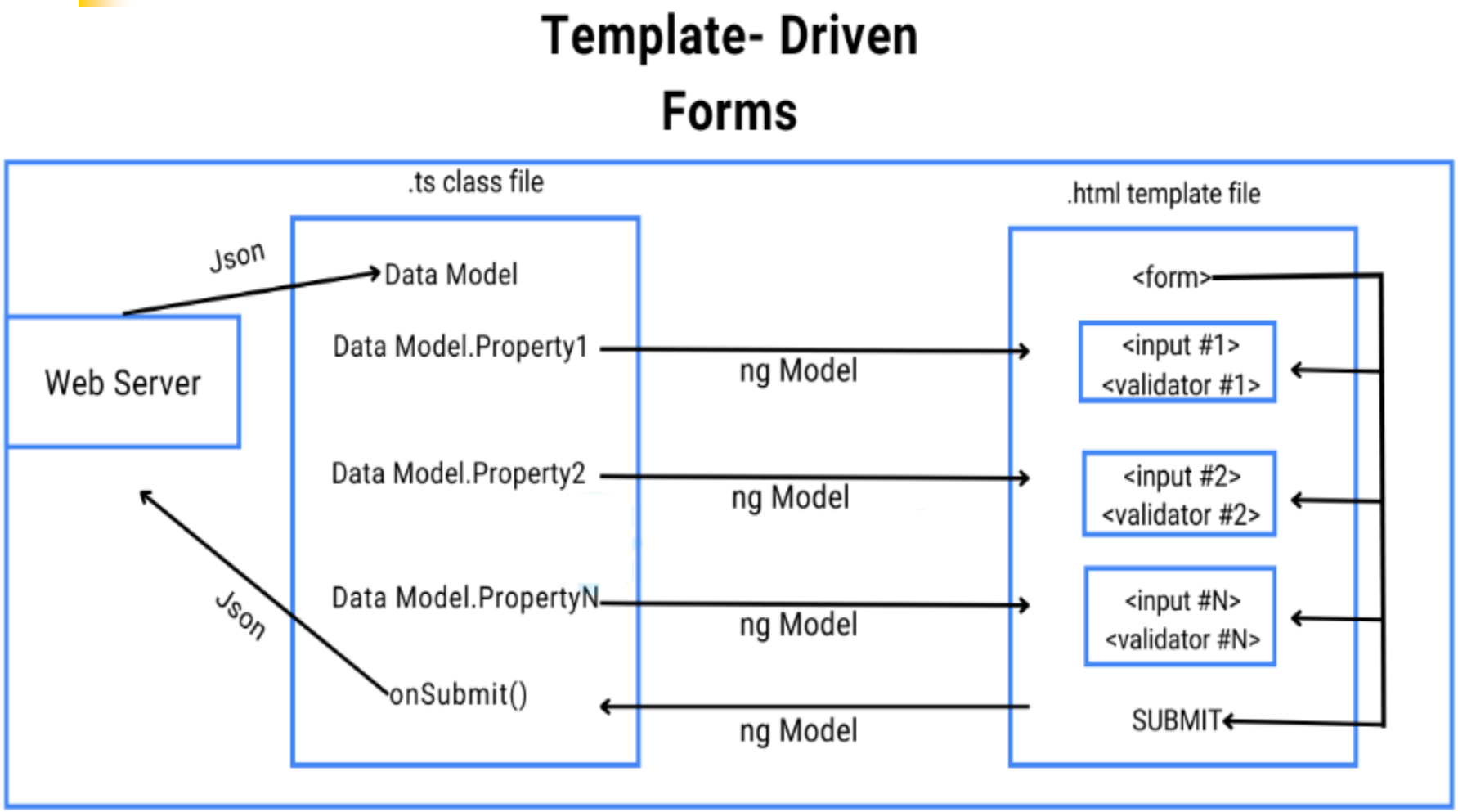
When it comes to connecting the template and the model, there are several approaches available. **ngModel** provides three distinct syntaxes to address this situation.

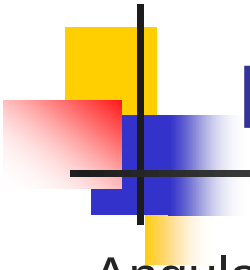
[(ngModel)] : This is known as two-way binding. It establishes a synchronization between the template form control and a property in the component. Changes in either the template or the component will instantly update the other.

[ngModel] : This is known as one-way binding from the component to the template. It allows setting an initial value for the template form control based on a property in the component.

ngModel : This is a standalone directive that provides services for template-driven forms. It's often used in combination with other bindings. It allows reference to the form control, which can be useful for validation and other purposes.

Template Driven Forms





Form States

Angular forms have different states such as **pristine**, **dirty**, **touched**, and **invalid**. These classes indicate the validity of the form elements

pristine / dirty: pristine means untouched—no value has been entered yet. Once the user starts typing, the control becomes **dirty**

touched / untouched: These relate to focus. If a user has clicked into a form field, it's **touched**; if they haven't, it's **untouched**

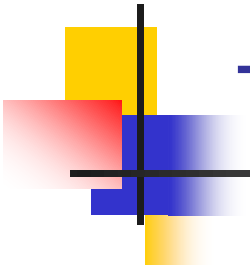
valid: A boolean that tells you if the input conforms to your **validation** rules. If you've said "Emails only!" and the user types "Just a name," **valid** will be false

Form States Example

```
<form [formGroup]="registrationForm" (ngSubmit)="onSubmit()">
  <label for="firstName">First Name</label>
  <input type="text" id="firstName" formControlName="firstName">
  <div *ngIf="firstName.invalid && (firstName.dirty || firstName.touched)">
    <small *ngIf="firstName.errors.required">First Name is required.</small>
  </div>

  <label for="lastName">Last Name</label>
  <input type="text" id="lastName" formControlName="lastName">
  <div *ngIf="lastName.invalid && (lastName.dirty || lastName.touched)">
    <small *ngIf="lastName.errors.required">Last Name is required.</small>
  </div>

  <label for="email">Email</label>
  <input type="email" id="email" formControlName="email">
  <div *ngIf="email.invalid && (email.dirty || email.touched)">
    <small *ngIf="email.errors.required">Email is required.</small>
    <small *ngIf="email.errors.email">Invalid email format.</small>
  </div>

  <button type="submit" [disabled]="registrationForm.invalid">Submit</button>
</form>
```

Form States Example

```
import { Component } from '@angular/core';
import { FormBuilder, FormGroup, Validators } from '@angular/forms';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  registrationForm: FormGroup;

  constructor(private formBuilder: FormBuilder) {
    this.registrationForm = this.formBuilder.group({
      firstName: ['', Validators.required],
      lastName: ['', Validators.required],
      email: ['', [Validators.required, Validators.email]],
    });
  }

  get firstName() {
    return this.registrationForm.get('firstName');
  }

  get lastName() {
    return this.registrationForm.get('lastName');
  }

  get email() {
    return this.registrationForm.get('email');
  }

  onSubmit() {
    console.log(this.registrationForm.value);
  }
}
```



Template Driven Forms

Template Driven Forms are used to bind the data to the component class using **ngModel**.

There is very less of effort required from the developer to pass information from the component class to the template.

Model-Driven (Reactive) Forms

Angular Model-Driven (Reactive) Forms are an effective tool for managing form state & validation in Angular applications by synchronizing form controls with model data. These forms take advantage of the capabilities of RxJS observables to provide dynamic updates, strong validation, and seamless integration with component functionality, hence improving the user experience and developer efficiency.

Reactive forms provide **synchronous** access to the data model, **immutability** through observable operators, and change tracking through observable streams.

They maintain form-state integrity by returning a new state for each modification, using an explicit and immutable approach.

Built on **observable** streams, they provide **synchronous** access to form inputs and values, which improves form state management.

ReactiveFormsModule

We will create a **Form**, using **FormGroup, FormControl, FormBuilder** class, etc. The first step is to import **ReactiveFormsModule** in the **app.module.ts** as listed below

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import {ReactiveFormsModule} from '@angular/forms';
import { AppComponent } from './app.component';

@NgModule({
 declarations: [
 AppComponent
 ],
 imports: [
 BrowserModule, ReactiveFormsModule
 ],
 providers: [],
 bootstrap: [AppComponent]
})
export class AppModule { }
```

FormControl, FormArray, FormGroup

Some important classes need to be imported inside the component like **FormControl**, and **FormArray** listed below

```
import { Component } from '@angular/core';
import {FormArray, FormControl, FormGroup} from '@angular/forms';


@Component({
 selector: 'app-root',
 template: ``
})
export class AppComponent {
 title = 'Reactive Forms';


}
```

FormControl

FormControl tracks the value and validation status of an individual form control.

FormControl is a single form control element, such as an input, select, or textarea. It is used to create forms that collect a single piece of data, such as a user's name, email address, or password.

```
emailControl: FormControl = new FormControl('');
```

On the template, we can use this FormControl using

```
<input [formControl]="emailControl" type="text" placeholder="What is your email?">
```

FormGroup (1)

FormGroup tracks the same values and status for a collection of form controls.

We can think of **FormControl** as a child property of **FormGroup**. The syntax used for **FormGroup** is as follows. **FormGroup** can contain any number of **FormControls** or other **FormGroups**.

```
registrationForm: FormGroup;
Codeium: Refactor | Explain | Generate JSDoc | X
constructor(private formBuilder: FormBuilder) {
  this.registrationForm = this.formBuilder.group({
    firstName: ['', Validators.required],
    lastName: ['', Validators.required],
    email: ['', [Validators.required, Validators.email]],
    items: this.formBuilder.array([])
  });
```

FormGroup (2)

•**Combines controls:** It allows you to group multiple **FormControls** into one unit.

•**Tracks their state: FormGroup** watches the state of its controls like whether all the fields in a section are valid).

•**Values and validity:** You can check the overall form value or validation status in one place, rather than individually.

```
registrationForm: FormGroup;
Codeium: Refactor | Explain | Generate JSDoc | X
constructor(private formBuilder: FormBuilder) {
  this.registrationForm = this.formBuilder.group({
    firstName: ['', Validators.required],
    lastName: ['', Validators.required],
    email: ['', [Validators.required, Validators.email]],
    items: this.formBuilder.array([])
  });
```

FormGroup (3)

```
registrationForm: FormGroup;
Codeium: Refactor | Explain | Generate JSDoc | ×
constructor(private formBuilder: FormBuilder) {
  this.registrationForm = this.formBuilder.group({
    firstName: ['', Validators.required],
    lastName: ['', Validators.required],
    email: ['', [Validators.required, Validators.email]],
    items: this.formBuilder.array([])
  });
```

```
<form [formGroup]="registrationForm" (ngSubmit)="onSubmit()">
  <div>
    <label for="firstName">First Name</label>
    <input type="text" id="firstName" formControlName="firstName">
    <div *ngIf="firstName!.invalid && (firstName!.dirty || firstName!.touched)">
      <small *ngIf="firstName!.errors?.['required']">First Name is required.</small>
    </div>
  </div>

  <div>
    <label for="lastName">Last Name</label>
    <input type="text" id="lastName" formControlName="lastName">
    <div *ngIf="lastName!.invalid && (lastName!.dirty || lastName!.touched)">
      <small *ngIf="lastName!.errors?.['required']">Last Name is required.</small>
    </div>
  </div>

  <div>
    <label for="email">Email</label>
    <input type="email" id="email" formControlName="email">
    <div *ngIf="email!.invalid && (email!.dirty || email!.touched)">
      <small *ngIf="email!.errors?.['required']">Email is required.</small>
      <small *ngIf="email!.errors?.['email']">Invalid email format.</small>
    </div>
  </div>
```

FormBuilder

To simplify the syntax of **FormGroup** and **FormControl**, we use **FormBuilder**. This helps reduce the length of our code and optimize it.

```
registrationForm: FormGroup;
Codeium: Refactor | Explain | Generate JSDoc | X
constructor(private formBuilder: FormBuilder) {
  this.registrationForm = this.formBuilder.group({
    firstName: ['', Validators.required],
    lastName: ['', Validators.required],
    email: ['', [Validators.required, Validators.email]],
    items: this.formBuilder.array([])
  });
```

FormArray

FormArray tracks the same values and status for an array of form controls.
If you want to add fields in the form dynamically with the help of an array, **FormArray** is used. Like other classes, it is imported in the component like this

```
constructor(private formBuilder: FormBuilder) {
  this.registrationForm = this.formBuilder.group({
    firstName: ['', Validators.required],
    lastName: ['', Validators.required],
    email: ['', [Validators.required, Validators.email]],
    items: this.formBuilder.array([])
  });
```

```
<div formArrayName="items">
  <div *ngFor="let item of items.controls; let i=index">
    <div [formGroupName]="i">
      <input formControlName="itemEmail" placeholder="Additional Email">
      <input formControlName="itemPassword" type="password" placeholder="Password">
      <button type="button" (click)="removeItem(i)">Remove</button>
    </div>
  </div>
  <button type="button" (click)="addItem()">Add Item</button>
</div>
```

Creating Model-Driven (Reactive) Forms

Step 1: Import ReactiveFormsModule in the App Module

Step 2: Import FormControl, and FormGroup classes in the component class

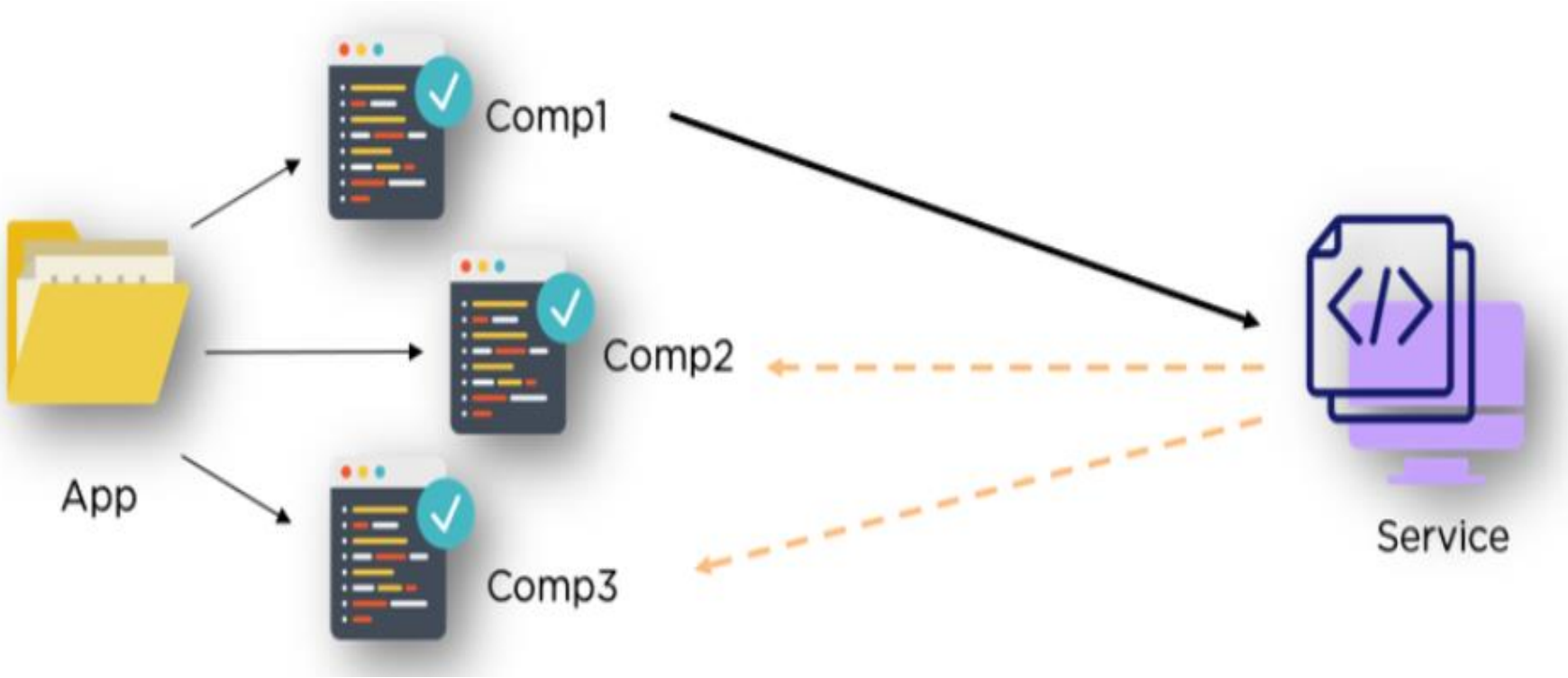
Step 3: Create a FormGroup class with details like email, password

Step 4: On the template, create a Submit button with the function to implement submit in the class.

Angular Forms (Mutability of the data model)

- The change-tracking method plays a role in the efficiency of your application.
- **Reactive forms** keep the data model pure by providing it as an **immutable data structure**.
- Each time a **change is triggered on the data model**, the **FormControl instance returns a new data model** rather than **updating the existing data model**.
- This gives you the ability to track **unique changes to the data model through the control's observable**.
- **Change detection** is **more efficient** because it **only needs to update on unique changes.**
- Because **data updates follow reactive patterns**, you can integrate with **observable operators to transform data**.

Services



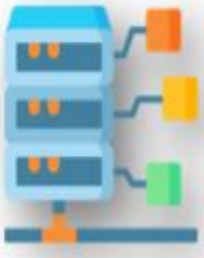
Services

A Service is a Class

Decorated with
@Injectibe

They share the same
piece of code

Hold the business logic



Interact with the
backend

Share data among
components

Services are singleton

Registered on modules
or components

Services

- Services in Angular are essentially classes that encapsulate specific functionalities, separate from the component logic.
- Service's primary objective is to share data, logic, or functionalities across different components in an application.
- Services can also be used to interact with a backend server to perform operations like fetching data, posting data, etc.

Services

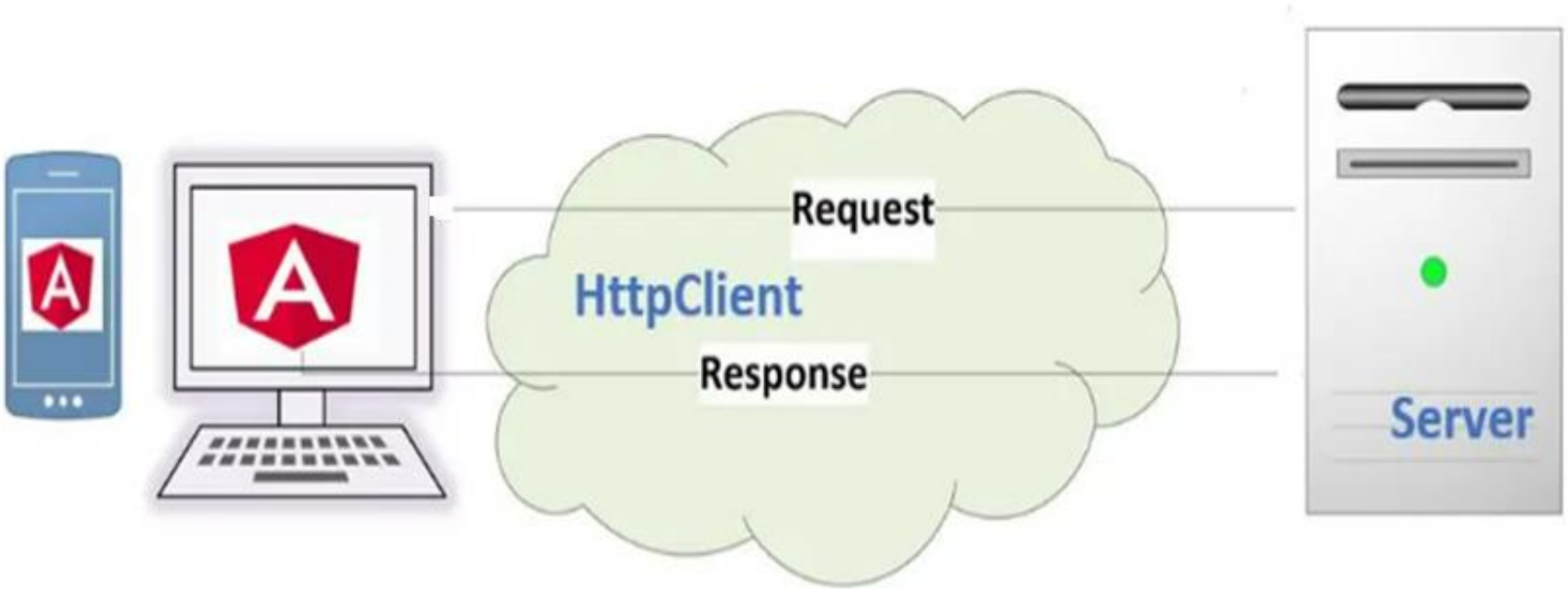
Here's an example of a user **service** (user.service.ts)

```
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root',
})
export class UserService {
  users = ['John', 'Doe', 'Smith'];
  getUsers() {
    return this.users;
  }
}
```

This **service** has a users array and a getUsers method to return the user list. The **@Injectable decorator** marks it as a service that can be **injected** into other classes.

Built-in Services (HttpClient)

The **HttpClient service** is an **Angular module** that allows us to **make HTTP requests** to a server. It is a powerful service that can be used to interact with RESTful APIs and retrieve data from a backend server.



Create Angular service to call REST APIs (HttpClient)

- REST client provided by Angular:
  - **`HttpClient`** ... part of **`HttpClientModule`**
- Add support in the application module

File: src/app/app.module.ts

```
...
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  declarations: [
    AppComponent
 ],
  imports: [
    BrowserModule,
    HttpClientModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Support for HttpClientModule

Create Angular service to call REST APIs (HttpClient)

File: src/app/services/product.service.ts

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Product } from '../common/product';
import { Observable } from 'rxjs';
import { map } from 'rxjs/operators';

@Injectable({
  providedIn: 'root'
})
export class ProductService {

  private baseUrl = 'http://localhost:8080/api/products';

  constructor(private httpClient: HttpClient) { }

  getProductList(): Observable<Product[]> {
    return this.httpClient.get<GetResponse>(this.baseUrl).pipe(
      map(response => response._embedded.products)
    );
  }
}

interface GetResponse {
  _embedded: {
    products: Product[];
  }
}
```

Our service can be injected into other classes / components

Inject httpClient

Unwraps the JSON from Spring Data REST _embedded entry

Returns an observable

Map the JSON data from Spring Data REST to Product array

Subscribe to data (call the service)

File: src/app/components/product-list/product-list.component.ts

```
import { Component, OnInit } from '@angular/core';
import { ProductService } from 'src/app/services/product.service';
import { Product } from 'src/app/common/product';

@Component({
  selector: 'app-product-list',
  templateUrl: './product-list.component.html',
 styleUrls: ['./product-list.component.css']
})
export class ProductListComponent implements OnInit {

  products: Product[];

  constructor(private productService: ProductService) { }

  ngOnInit() {
    this.listProducts();
  }

  listProducts() {
    this.productService.getProductList().subscribe(
      data => {
        this.products = data;
      }
    )
  }
}
```

Method is invoked once you "subscribe"

Assign results to the Product array

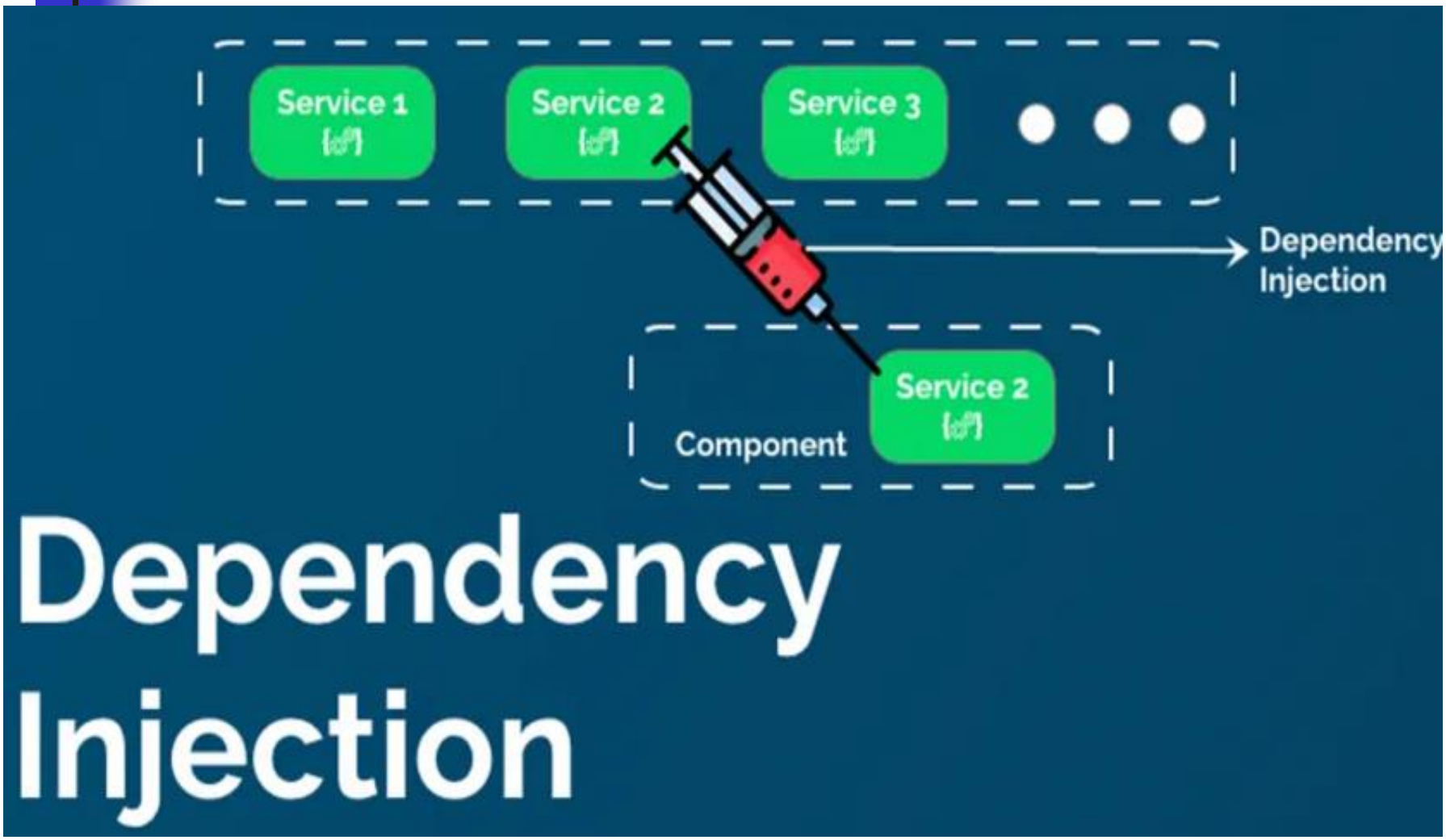
Key points about services

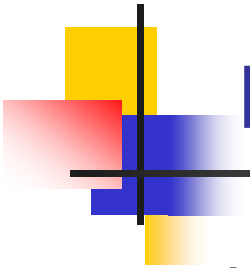
- **Single Responsibility Principle:** Each service should be built around a specific functionality following the **Single Responsibility Principle**. For example, you might have one service for **user authentication** and **another** for **data retrieval**.

- **Dependency Injection: Services** can be **injected** into **components**, which makes your code more **modular** and **reusable**. You don’t have to create an instance of a service using the new keyword. Instead, **Angular's dependency injection system** takes care of creating and providing the instances wherever needed.

- **Singleton Services:** By default, services are **singletons** in Angular. This means that Angular will create only one instance of your service and reuse it wherever that service is needed. This is useful for **sharing data** and **functionality**

Dependency Injection





Dependency Injection

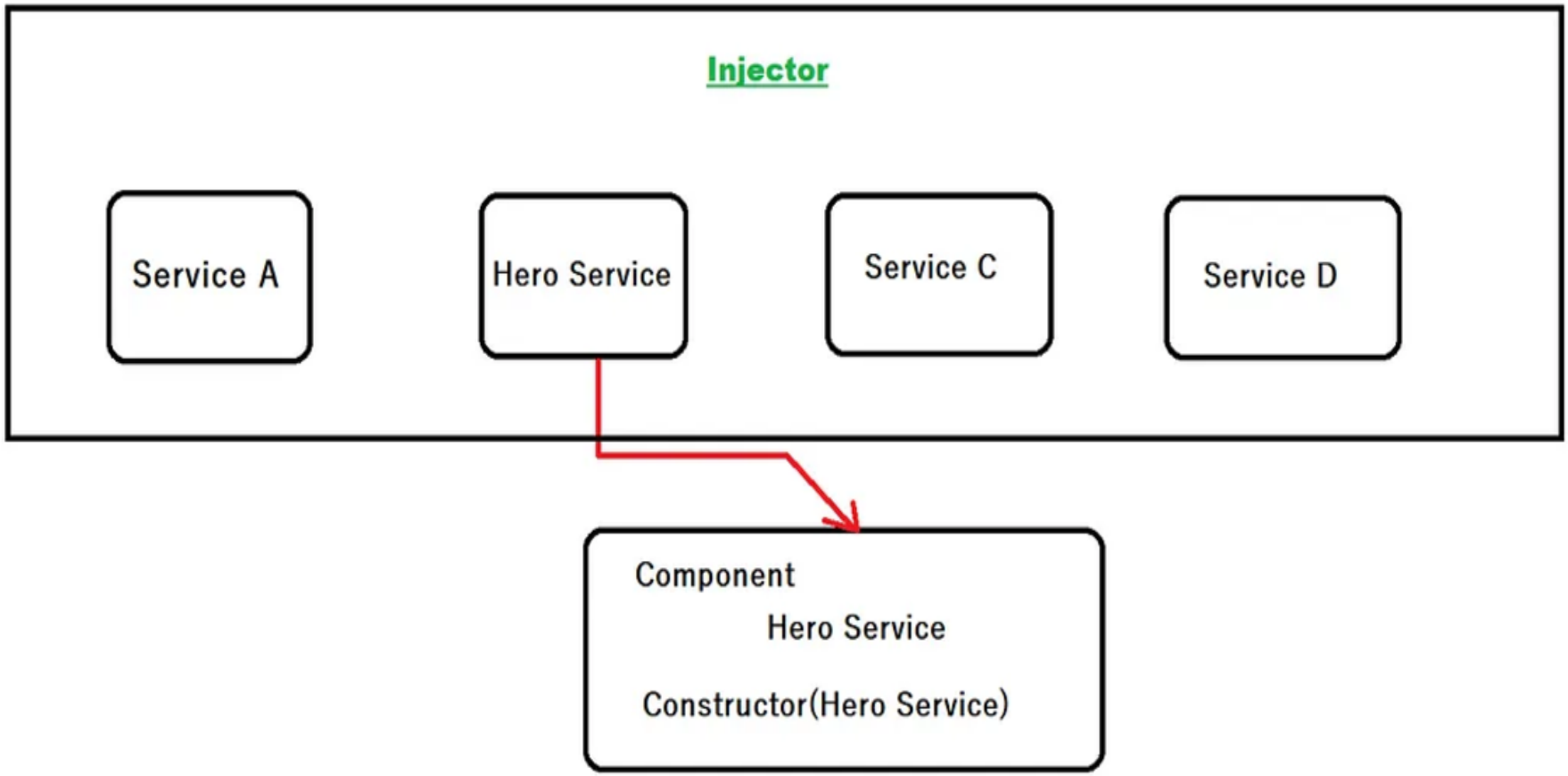
Dependency injection, or **DI**, is a **design pattern** in which a class requests dependencies from external sources rather than creating them.

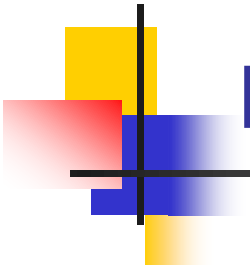
With **Dependency Injection** your class **receives its dependencies** from **external sources(container) rather than creating them itself**.

Dependency injection (DI) in Angular provides a way to create and inject dependencies into components or services, making it easier to manage the relationships between different parts of our application.

Dependency injection helps improve **modularity**, **maintainability**, and **testability** of our Angular applications by promoting **loose coupling** between **components** and their **dependencies**

Dependency Injection





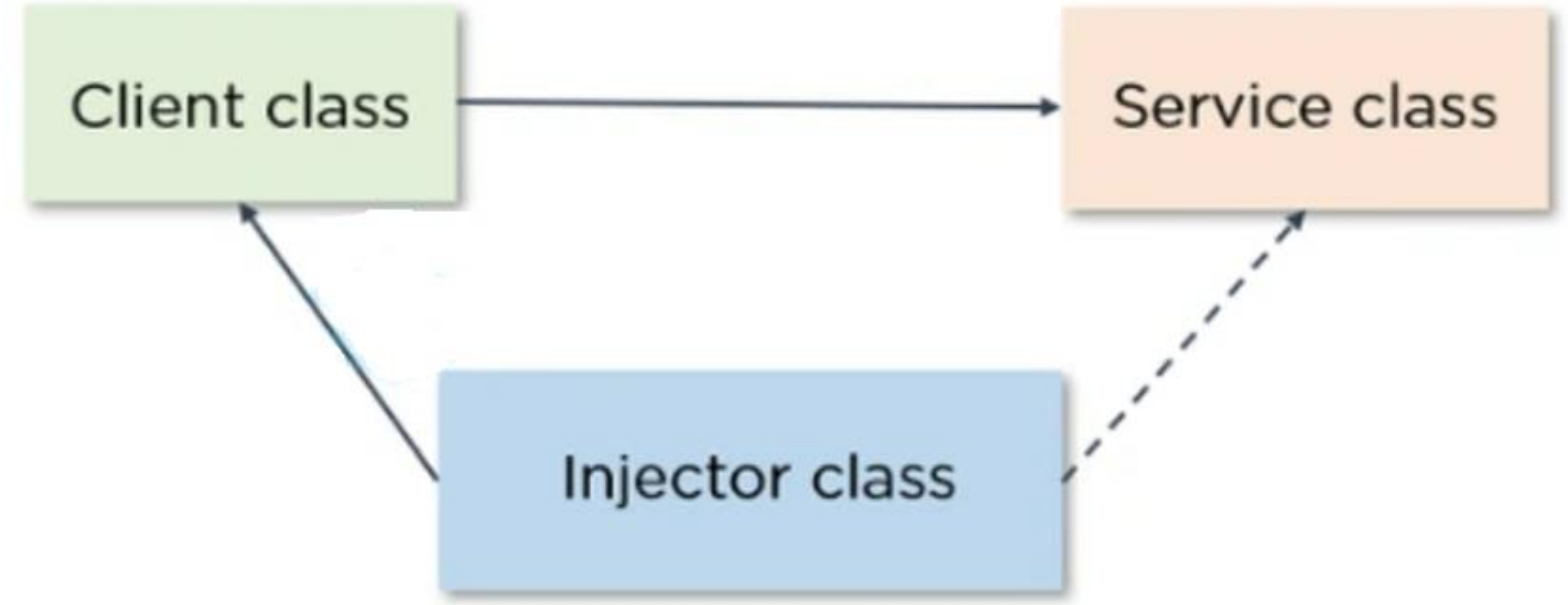
Dependency Injection Key Concepts

Dependency: In the context of Angular, a **dependency** refers to any **service**, **object**, or **value** that a **component** needs to perform its tasks. This can include **HTTP services, data services, configuration settings**

Service: A service in Angular is a class that provides some functionality or data. Services are typically used to **share data or logic between components**, and they are **injectable**, meaning they can be injected into other parts of our application.

Injector: Angular's injector is responsible for **creating** and **managing** instances of services and **injecting** them into components or other services. It maintains a **registry of providers**, which are **instructions** on how to create **instances of services**.

Dependency Injection



1.**Client Class** - This is the dependent class, which depends on the service class.
2.**Service Class** - Class that provides the service to the client class.
3.**Injector Class** - Injects the service class object into the client class.

1. Provider Registration in Angular

Providers are responsible for creating and delivering dependencies. There are different ways to register a provider in Angular

1.1 Registering a Provider in a Component:

- You can register a provider inside the providers array of a component
- LoggerService is provided only to AppComponent and its child components.

```
import { Component } from '@angular/core';
import { LoggerService } from './logger.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  providers: [LoggerService] // Registering service at the component level
})
export class AppComponent {
  constructor(private logger: LoggerService) {
    this.logger.log('AppComponent initialized!');
  }
}
```

1. Provider Registration in Angular

1.2 Registering a Provider in a Module (Singleton Service):

- If you want a service to be shared across the entire application, you can provide it in the **@Injectable decorator** or inside **app.module.ts**
- This means that LoggerService is registered as a singleton service throughout the application.

Using providedIn **in Service**:

```
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root' // This makes LoggerService available globally
})
export class LoggerService {
  log(message: string) {
    console.log(message);
  }
}
```

1. Provider Registration in Angular

1.2 Registering a Provider in a Module (Singleton Service):

Registering in app.module.ts
Alternatively, you can register a service explicitly in the module:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
import { LoggerService } from './logger.service';

@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule],
  providers: [LoggerService], // Service registered at the module level
  bootstrap: [AppComponent]
})
export class AppModule { }
```

2. Using @Inject() Decorator

Step 1: Define an Injection Token for Configuration

We create a new InjectionToken to store our configuration object.

```
import { InjectionToken } from '@angular/core';

// Define an injection token for app configuration
export const APP_CONFIG = new InjectionToken<AppConfig>('app.config');

// Define the interface for the configuration object
export interface AppConfig {
  apiEndpoint: string;
  enableLogging: boolean;
}
```

2. Using @Inject() Decorator

Step 2: Provide the Configuration in app.module.ts

register the configuration object inside providers

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
import { APP_CONFIG, AppConfig } from './app.config';

// Create a configuration object
const appConfig: AppConfig = {
  apiEndpoint: 'https://myapi.com',
  enableLogging: true
};

@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule],
  providers: [
    { provide: APP_CONFIG, useValue: appConfig } // Provide the configuration object
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

2. Using @Inject() Decorator

Step 3: Inject the Configuration Object Using @Inject()

in our component, we inject the APP_CONFIG token and use its values.

```
import { Component, Inject } from '@angular/core';
import { APP_CONFIG, AppConfig } from './app.config';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  constructor(@Inject(APP_CONFIG) private config: AppConfig) {
    console.log('API Endpoint:', this.config.apiEndpoint);
    console.log('Logging Enabled:', this.config.enableLogging);
  }
}
```

3. Using Injection Tokens in Angular

Injection Tokens are used when we need to inject values that are not classes, such as configuration objects, API URLs, or primitive values.

3.1 Defining an Injection Token

```
import { InjectionToken } from '@angular/core';

// Creating an injection token
export const API_URL = new InjectionToken<string>('apiUrl');
```

3.2 Providing the Injection Token in a Module

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
import { API_URL } from './api.token';

@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule],
  providers: [
    { provide: API_URL, useValue: 'https://api.example.com' } // Providing a value for the tok
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

3. Using Injection Tokens in Angular

3.3 Injecting the Token in a Component

```
import { Component, Inject } from '@angular/core';
import { API_URL } from './api.token';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  constructor(@Inject(API_URL) private apiUrl: string) {
    console.log('API URL:', this.apiUrl); // Logs 'https://api.example.com'
  }
}
```

1.Lambda Functions (Arrow Functions)

lambda functions refer to **arrow functions** (=>). These are a modern way to define functions with a shorter syntax compared to traditional function expressions.

```
function add(a: number, b: number): number {
  return a + b;
}
```

This is equivalent to:

```
const add = (a: number, b: number): number => {
  return a + b;
};

console.log(add(5, 3)); // Output: 8
```

2.Lambda Functions - setTimeout()

Arrow functions help preserve ***this*** when working with setTimeout()

```
export class AppComponent {
  message = 'Hello, Angular!';

  constructor() {
    setTimeout(() => {
      console.log(this.message); // Works correctly
    }, 1000);
  }
}
```

If we used a normal function:

```
setTimeout(function () {
  console.log(this.message); // ❌ `this` is undefined
}, 1000);
```

This fails because ***this*** inside function () {} refers to the global object, not the AppComponent class.

How **this** Works in a Regular Function ?

When you define a function using the **function** keyword:

```
function myFunction() {
  console.log(this); // `this` refers to the global object (`window` in browsers)
}
myFunction();
```

- this is **determined at runtime** based on how the function is called.
- It **does not automatically refer to the class instance** in object-oriented programming.
- **this** refers to the **global object (window in browsers)**.

```
setTimeout(function () {
  console.log(this.message); // ❌ `this` is undefined
}, 1000);
```

window.message is **not defined**

Normal Function - setTimeout()

How to Fix It in normal function ? : **Solution 1**. you need to explicitly bind this to the function.

```
setTimeout(function () {
  console.log(this.message); // ✅ Works correctly
}.bind(this), 1000);
```

.bind(this) forces **this** to refer to **the class instance** instead of **the global object**.

Solution 2: Assign **this** to a Variable

```
let that = this;
setTimeout(function () {
  console.log(that.message); // ✅ Works correctly
}, 1000);
```

- let that = this; stores the correct reference to this.
- Inside the function, that.message correctly refers to the class property.

3. Lambda Functions – (for Event Handling)

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  template: `<button (click)="onClick()">Click Me</button>`
})
export class AppComponent {
  message = 'Hello';

  onClick = () => {
    console.log(this.message); // Works correctly because of lexical `this`
  };
}
```

The arrow function **automatically binds** this to the class instance (AppComponent)

The () before => represents the **function parameters** (empty () means no parameters).

4. Lambda Functions – Using map()

Transforming data in RxJS (map(), filter(), etc.).

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';
import { map } from 'rxjs/operators';

@Injectable({
  providedIn: 'root'
})
export class DataService {
  private apiUrl = 'https://jsonplaceholder.typicode.com/users';

  constructor(private http: HttpClient) {}

  getUserNames(): Observable<string[]> {
    return this.http.get<any[]>(this.apiUrl).pipe(
      map(users => users.map(user => user.name)) // Arrow function inside map()
    );
  }
}
```

pipe() – What Does It Do?

The **pipe()** method in RxJS is used to **combine multiple operators** in a readable and chainable way.

```
return this.http.get<any[]>(this.apiUrl).pipe(
  map(users => users.map(user => user.name))
);
```

- Allows multiple transformations in a **single chain**.
- Used with operators like map(), filter(), catchError(), etc

- `this.http.get<any[]>(this.apiUrl)`:
  - **Performs an HTTP GET request** to fetch an array of users (`any[]`).
  - Returns an **observable** (`Observable<any[]>`).
- `.pipe(...)`:
  - **Transforms the observable's output** before it is passed to the subscriber.
  - The function inside `pipe()` modifies the response using **RxJS operators** (e.g., `map()`).

map() (RxJS Operator)

The map() operator in RxJS **transforms** the emitted values from an observable.

```
map(users => users.map(user => user.name))
```

- The first map() (**RxJS operator**) **takes the array of users** from the HTTP response.
- It **returns a new array** containing only the name property of each user.

Before Transformation (Fetched Data from API)

```
[
  { "id": 1, "name": "Alice", "email": "alice@example.com" },
  { "id": 2, "name": "Bob", "email": "bob@example.com" }
]
```

After Applying map()

```
["Alice", "Bob"]
```

Why Use map()? Extracts only **necessary data** from the API response.

=> (Arrow Function)

1. First Arrow Function:

```
users => users.map(user => user.name)
```

- `users` is the parameter (the array from API response).
- The function **returns a new array** where each item is transformed using `.map()`.

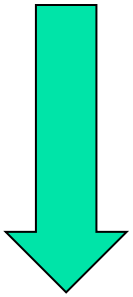
2. Second Arrow Function Inside .map()

```
user => user.name
```

- `user` represents each object inside the `users` array.
- It extracts only the `name` property.

Lambda Functions – Using map()

```
map(users => users.map(user => user.name))
```



Equivalent Code Using Regular Functions

```
map(function(users) {
  return users.map(function(user) {
    return user.name;
  });
})
```

Lambda Functions – Using map()

```
export class DataService {
  private apiUrl = 'https://jsonplaceholder.typicode.com/users';

  constructor(private http: HttpClient) {}

  getUserNames(): Observable<string[]> {
    return this.http.get<any[]>(this.apiUrl).pipe(
      map(users => users.map(user => user.name)) // Transforms response to array of names
    );
  }
}
```

1. **Fetches data from API** (http.get()).
2. **Processes the response**:
   - Uses pipe() to apply transformations.
   - Uses map() to extract name values from objects.
3. **Returns an observable** that emits only user names.

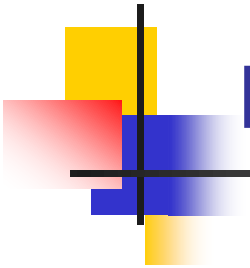
5. Lambda Functions – Using filter ()

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';
import { filter, map } from 'rxjs/operators';

interface User {
  id: number;
  name: string;
  age: number;
}

@Injectable({
  providedIn: 'root'
})
export class UserService {
  private apiUrl = 'https://api.example.com/users';

  constructor(private http: HttpClient) {}

  // Fetch users and filter those older than 30
  getUsersAbove30(): Observable<User[]> {
    return this.http.get<User[]>(this.apiUrl).pipe(
      map(users => users.filter(user => user.age > 30)) // ✅ Filtering inside map
    );
  }
}
```

Lambda Functions – Using filter ()

```
getUsersAbove30(): Observable<User[]> {
  return this.http.get<User[]>(this.apiUrl).pipe(
    map(users => users.filter(user => user.age > 30)) // ✅ Filtering inside map
  );
}
```

- http.get<User[]>(this.apiUrl): Fetches an array of users.
- .pipe(...): Applies **RxJS operators** to process the response.
- map(users => users.filter(user => user.age > 30)):
  - map() transforms the array of users.
  - filter(user => user.age > 30) removes users younger than 30.



Reactive Programming

Reactive Programming is a programming paradigm that deals with **asynchronous data streams** and the propagation of changes.

It provides a way to **handle** and respond to **events, user inputs, and data changes** in a declarative and efficient manner.

Reactive programming is widely used in modern software development, especially in the context of building responsive and interactive user interfaces, as well as in systems that involve real-time data processing.

What is a data stream?

Data Stream is a **sequence of data items** that are **emitted** over time **from a source**.

Data Stream can represent any kind of **asynchronous or event-driven data**, such as user input, network requests, sensor readings, etc.

Data Stream can be **subscribed** to by **observers** that **react to the data items**

Reactive programming is based on the concept of **data streams**, which are **sequences** of **data** that are **emitted over time**.

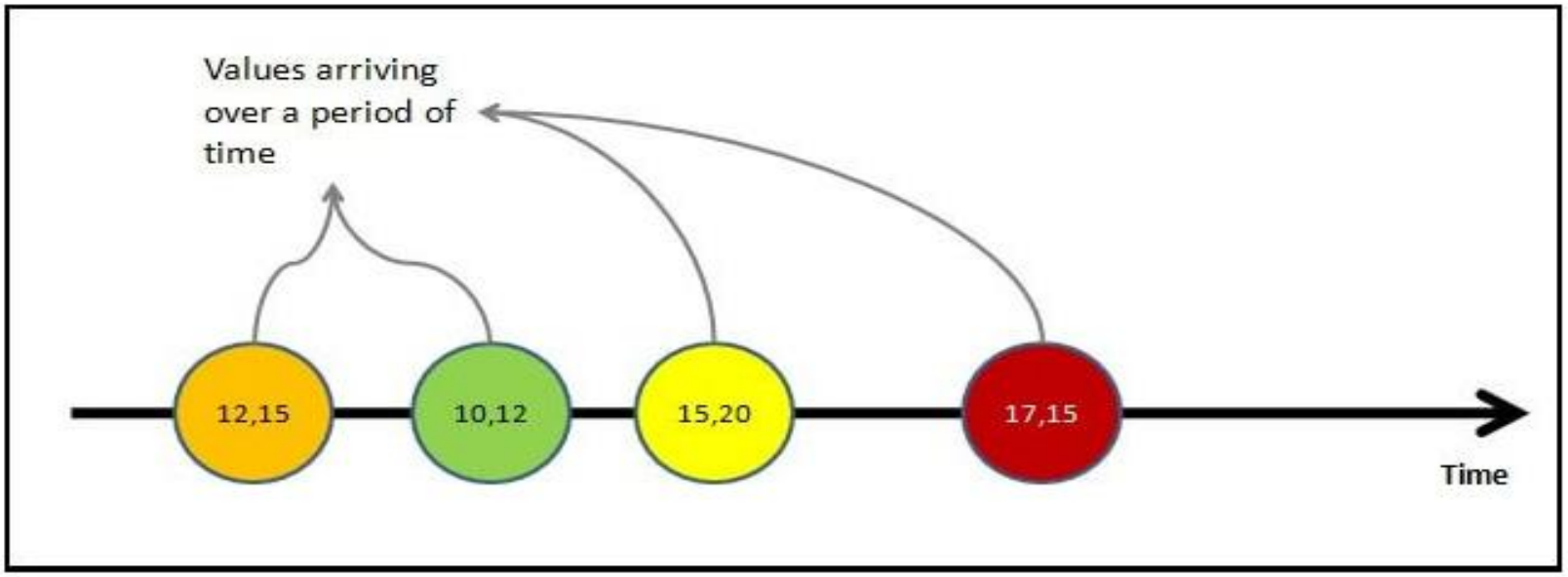
Changes to the data are propagated automatically, applications built using reactive programming can respond quickly to user input and other events. This makes **reactive programming** ideal for building **real-time applications**, such as chat apps, stock trading platforms

Reactive programming often relies on the concept of **data streams**, which are **sequences** of data items that are processed **asynchronously** and **reactively**.

Data stream

Consider the example of a sequence of x and y positions of **mouse click events**. Assume that the user has clicked on the locations (12, 15), (10, 12), (15, 20), and (17, 15) in that order.

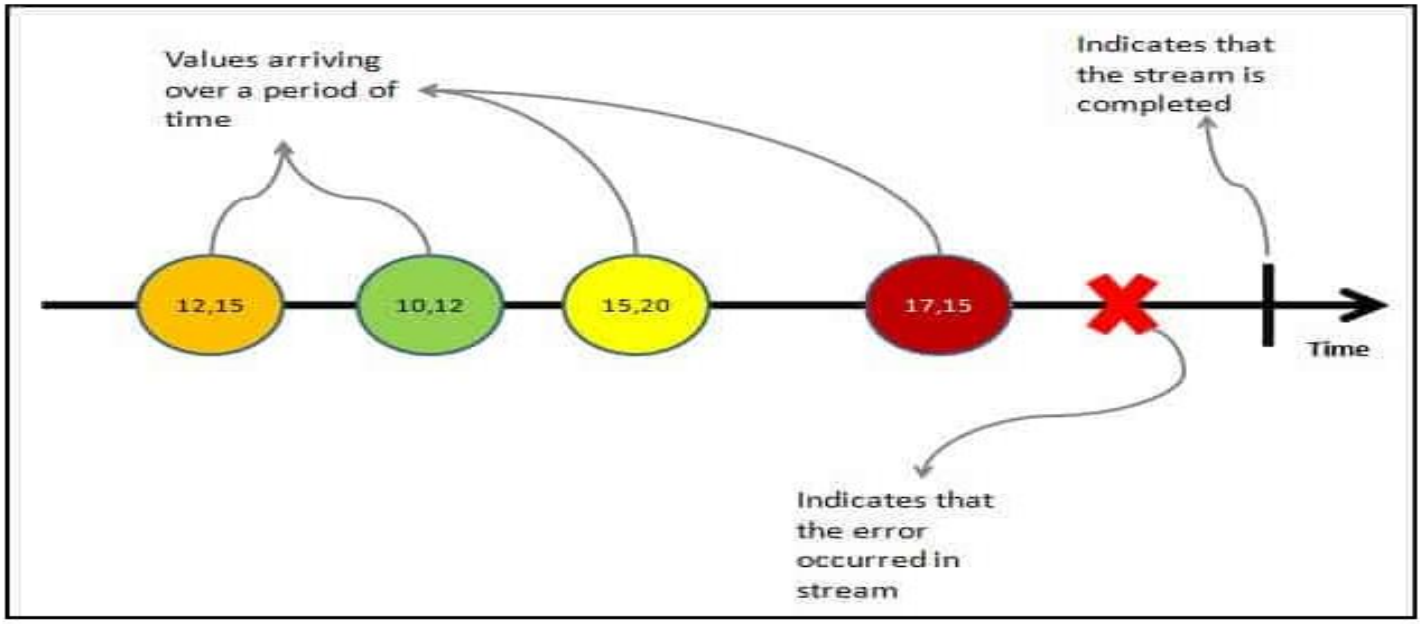
The following diagram shows how the values arrive over a period of time. As you can see, **the stream emits the values as they happen asynchronously.**



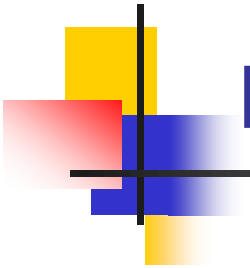
mouse click events as data streams

Data stream

Value is not the only thing that **streams emit**. The stream may complete as the user closes the window or app. Or an error may happen, resulting in the stream's closure. At any point in time, **the stream may emit the following three things**.
Value: i.e., the next value in the stream
Complete: The stream has ended
Error: The error has stopped the stream.



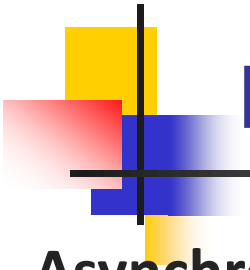
mouse click events as data streams with emit error and complete events



Data stream

The stream of data can be anything. For Example

- Mouse click or Mouse hover events with x & y positions
- Keyboard events like keyup, keydown, keypress, etc
- Form events like value changes etc
- Data that arrives after an HTTP request
- User Notifications
- Measurements from any sensor

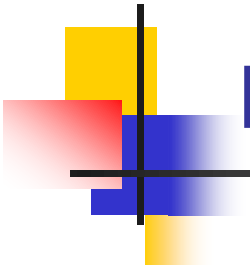


Reactive Programming

Asynchronous and Non-blocking

In **traditional programming (Imperative programming)**, tasks are executed **sequentially**, meaning that **each operation waits for the previous one to complete**. This approach can **lead** to **bottlenecks** and **poor performance**, especially in applications with **high concurrency**.

Reactive programming, on the other hand, allows tasks to be executed **asynchronously**, meaning that operations can be initiated **without waiting for previous tasks to complete**. This **asynchronous** nature enables applications to handle a large number of **concurrent requests without blocking threads**, leading to better **resource utilisation and improved performance.**

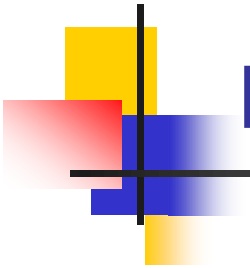


Benefits of Reactive Programming

- **Scalability**: Reactive programming enables applications to scale effortlessly, thanks to its asynchronous and event-driven nature.

- **Responsiveness**: Reactive applications are highly responsive, providing users with real-time feedback and seamless experiences.

- **Resilience**: By embracing asynchronous and non-blocking I/O, reactive applications are more resilient to failures and can gracefully handle errors.



Reactive Programming

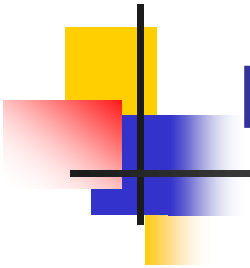
Some examples of when you might use reactive programming include:

•Developing real-time applications, such as **games, chat systems, or financial dashboards, that need to respond quickly to user interactions and changes in data**.

•Developing **event-driven systems**, such as **web services or microservices**, that need to **handle large amounts of data and events in a scalable and efficient** manner.

•Implementing **user interfaces**, where the **state of the UI** is driven by **user interactions and changes in data.**

•Developing systems that **handle multiple data streams**, such as **video and audio, and need to process and respond to these streams in real-time**.

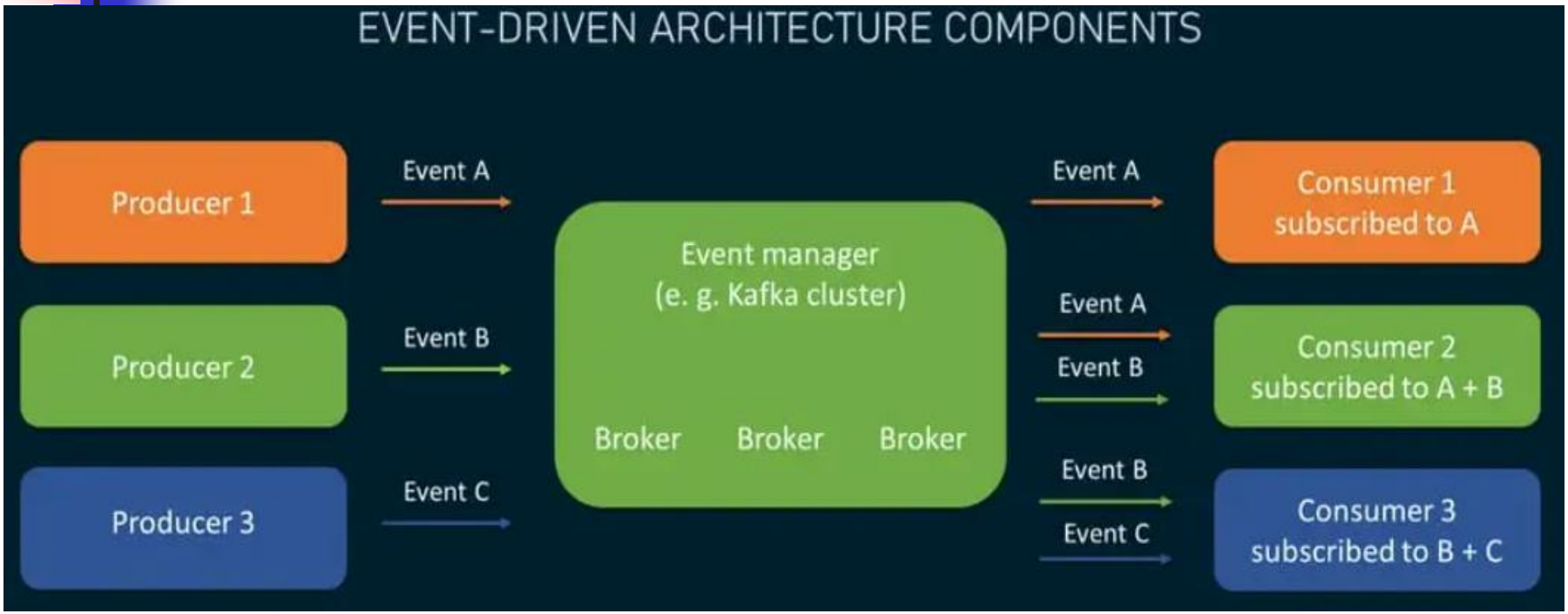


Reactive Programming

Some of the examples where reactive programming is used in Angular

- Reacting to an **HTTP request** in Angular
- **Value changes / Status Changes** in Angular Forms
- **The Router and Forms modules** use **observables** to listen for and respond to user-input events.
- You can define **custom events** that send **observable output** data from a **child to a parent component**.
- The **HTTP module** uses **observables** to **handle AJAX requests** and **responses**.
- **Websockets**

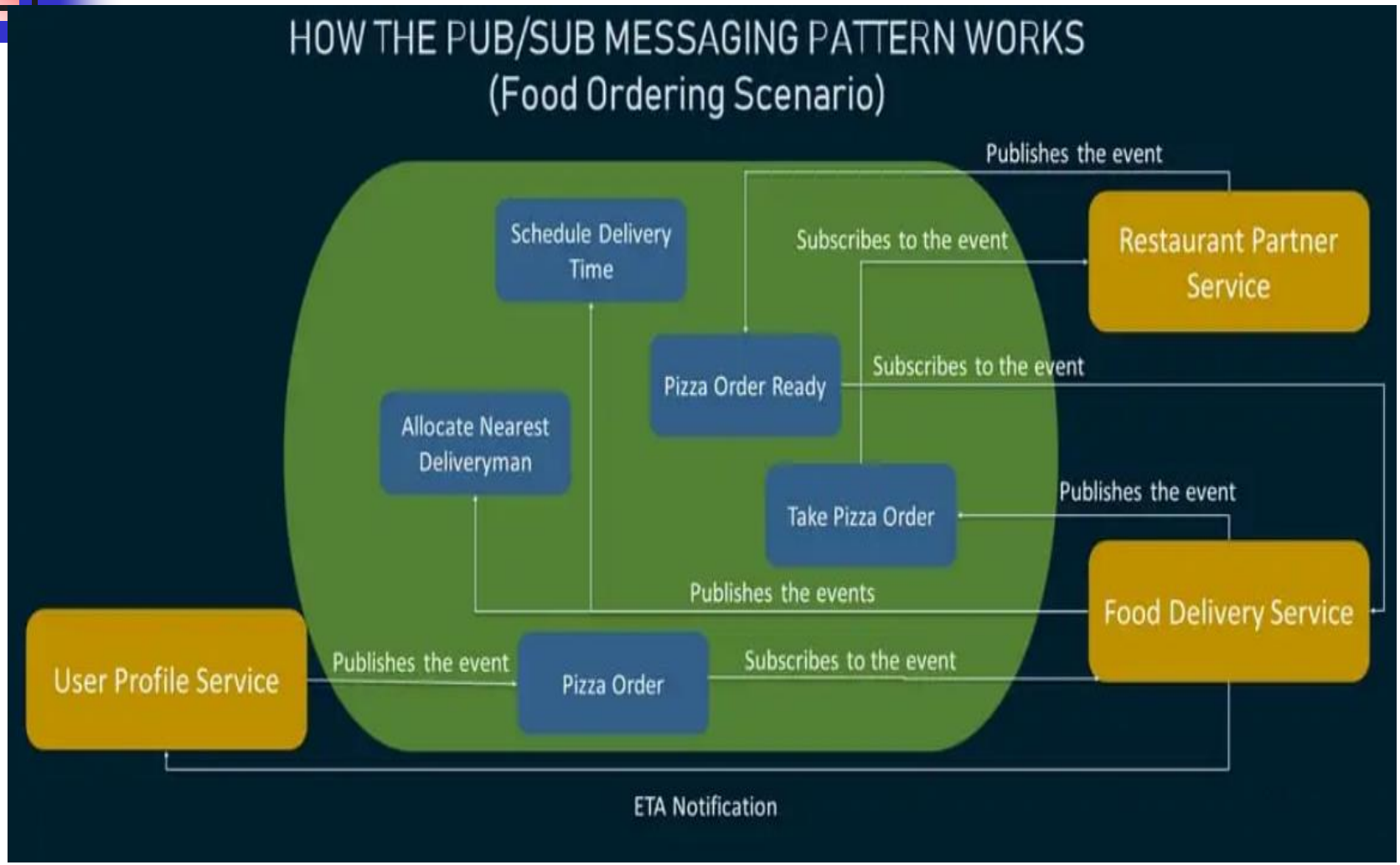
Event-driven Architecture



Reactive programming is designed to handle **complex and event-driven systems**, which are becoming more common in modern software development.

In reactive systems, components communicate by **emitting** and **reacting** to **events**. **Events** can be anything from user inputs to **data changes or system notifications**. This **event-driven** architecture enables **loosely coupled** and **highly scalable systems**, where **components can react to changes in real-time**.

Event-driven Architecture (Example)



monitor the location of the order and provide ETA (Estimated Time of Arrival) notifications for the user

Reactive Programming with RxJS

Reactive Extensions for JavaScript (RxJS) is a Javascript library that allows us to work with **asynchronous data streams**

RxJS is using **Observables**, which makes it easier to compose **asynchronous** or **callback-based code**.

The **RxJs** has two main players

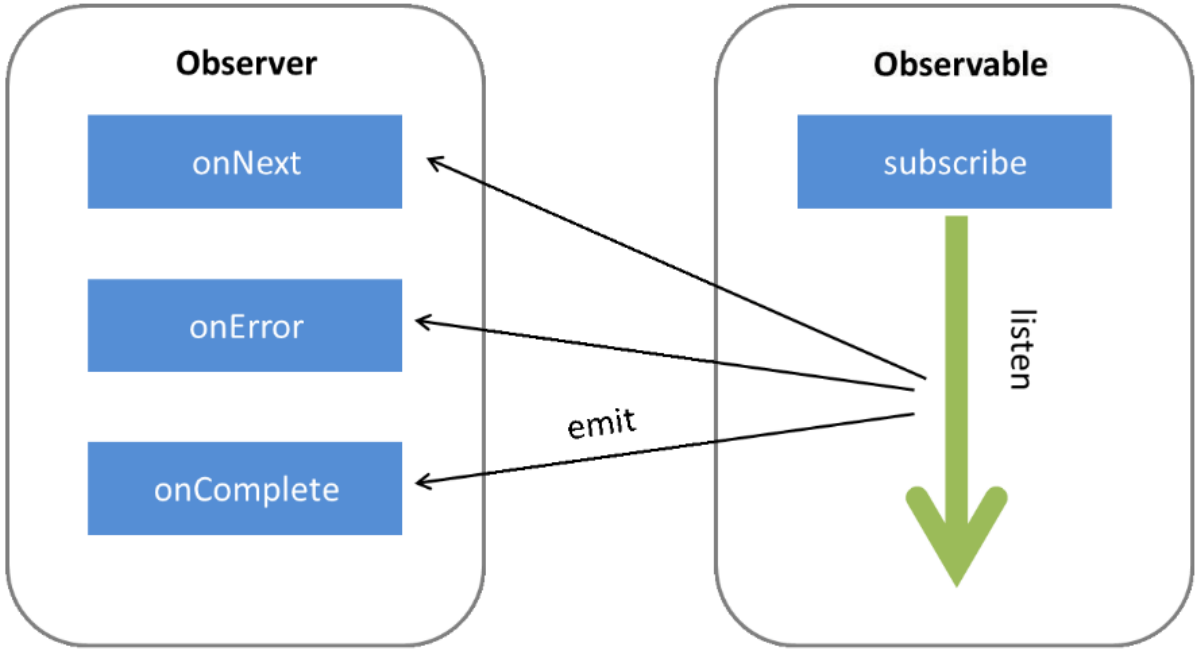
1. **Observable**
2. **Observers (Subscribers)**

To start using **RxJS** in your project, you can install it via npm

```
npm install rxjs
```

```
import { Observable } from 'rxjs';
```

Reactive Programming with RxJS

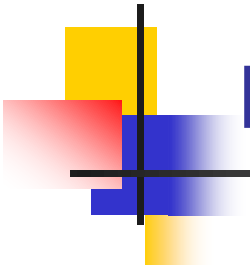


The **Observable** sends data while the **Observer** subscribes to it to receive the data.

The **Observable** fires the data in response to an event. For example, when a user clicks a button or in response to data that is received from a remote server.

On the other hand, the **Observer** has three handles to use the data that it receives:

- **onNext** handles the requested data
- **onError** to handle errors
- **onComplete** which is used when the process ends



Real-World Analogy - YouTube

Observable: Think of the **Observable** as a **YouTube channel** that posts videos. It's a source of data that can **emit** updates (videos, new data) at any time.

Observer: The **Observer** is like your **YouTube account**, which **subscribes** to the channel. It only **receives notifications** about new videos when it subscribes.

next(): This is how the **Observable emits new data** (videos). It **pushes** new videos to the **Observer**.

subscribe(): This is how the **Observer subscribes** to the **Observable** to **start receiving data** (new videos).

Real-World Analogy Example Code

```
// Step 1: Observable (YouTube Channel) - Emits videos
const youtubeChannelObservable = new Observable<string>((observer) => {
  console.log('YouTube Channel Started!');

  observer.next('🎬 Video 1: Introduction to Angular');  // 🔹 Emits Video 1
  observer.next('🎬 Video 2: Understanding RxJS');       // 🔹 Emits Video 2

  setTimeout(() => {
    observer.next('🎬 Video 3: Advanced Observables');   // 🔹 Emits Video 3 after 2s
    observer.complete();  // 🔹 No more videos (Stops)
  }, 2000);
});

// Step 2: Observer (YouTube User) - Listens for new videos
const youtubeUserObserver: Observer<string> = {
  next: (video: string) => console.log(`📢 New Video: ${video}`),  // Logs videos
  error: (error: any) => console.log(`❌ Error: ${error}`),        // Handles errors
  complete: () => console.log('✅ No more videos!'),              // Stops listening
};

// Step 3: Subscribe (User subscribes to channel)
youtubeChannelObservable.subscribe(youtubeUserObserver);
```

Real-World Analogy - Observable

```
// Step 1: Observable (YouTube Channel) - Emits videos
const youtubeChannelObservable = new Observable<string>((observer) => {
  console.log('YouTube Channel Started!');

  observer.next('🎬 Video 1: Introduction to Angular');  // 🔹 Emits Video 1
  observer.next('🎬 Video 2: Understanding RxJS');       // 🔹 Emits Video 2

  setTimeout(() => {
    observer.next('🎬 Video 3: Advanced Observables');   // 🔹 Emits Video 3 after 2s
    observer.complete();  // 🔹 No more videos (Stops)
  }, 2000);
});
```

- Each time observer.next(videoTitle) is called inside the Observable, it **sends data** to the subscriber (Observer).
- The Observer's next() function will be executed **each time** new data is emitted.
- When observer.next(videoTitle) **emits a value**, it **triggers** next(video: string) inside the Observer.

Real-World Analogy - Observer

```
// Step 2: Observer (YouTube User) - Listens for new videos
const youtubeUserObserver: Observer<string> = {
  next: (video: string) => console.log(`📢 New Video: ${video}`),  // Logs videos
  error: (error: any) => console.log(`❌ Error: ${error}`),        // Handles errors
  complete: () => console.log('✅ No more videos!'),               // Stops listening
};
```

- If observer.error() is ever called, the Observer's error(error: any) function runs.
- The Observable **stops emitting new values** and jumps directly to the error handler.

- When observer.complete() is called, the Observer's complete() function runs.
- The Observable **stops sending** new values.

Real-World Analogy - subscription process

```
// Step 3: Subscribe (User subscribes to channel)
youtubeChannelObservable.subscribe(youtubeUserObserver);
```

Connection Established

•The subscribe() method **connects** the Observer (youtubeUserObserver) to the Observable (youtubeChannelObservable).

•Now, the Observer is **ready to receive** data whenever the Observable emits something.

Real-World Analogy Example Code

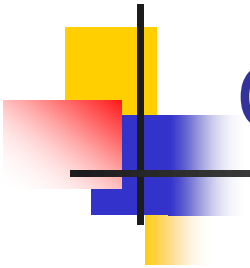
Concept	Role in Observable	Role in Observer
next()	Emits a new value (`observer.next(video)`)	Receives and processes it (`next(video)`)
error()	Signals an error (`observer.error(err)`)	Handles the error (`error(err)`)
complete()	Signals completion (`observer.complete()`)	Runs when no more data is received (`complete()`)
subscribe()	Starts the connection	Observer listens for values

NotificationService Example Code

```
import { Observable } from 'rxjs';

export class NotificationService {
  private notificationObservable: Observable<string>;

  constructor() {
    this.notificationObservable = new Observable<string>((observer) => {
      // Simulating notifications after some delay
      setTimeout(() => observer.next('New message received!'), 2000);
      setTimeout(() => observer.next('Another notification!'), 4000);
      setTimeout(() => observer.complete(), 6000); // Marks completion
    });
  }

  // Function to get the observable
  getNotifications(): Observable<string> {
    return this.notificationObservable;
  }
}
```

NotificationService - Subscribing to the Observable

```
import { Component, OnInit } from '@angular/core';
import { NotificationService } from './notification.service';

@Component({
  selector: 'app-notification',
  template: `<p>{{ message }}</p>`,
})
export class NotificationComponent implements OnInit {
  message = '';

  constructor(private notificationService: NotificationService) {}

  ngOnInit() {
    this.notificationService.getNotifications().subscribe({
      next: (msg) => (this.message = msg), // Receive the emitted value
      complete: () => console.log('No more notifications!'),
    });
  }
}
```

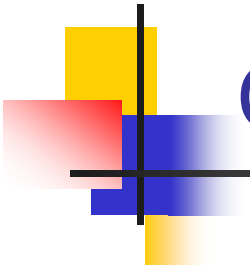


Observable in Angular

- Observable is a function that converts the **ordinary data stream** into an **observable one**. You can think of Observable as a wrapper around the **ordinary data stream**.

- **An observable stream** or simple **Observable emits** the **value from the stream asynchronously**. It emits the **complete** signals when the stream completes or an **error** signal if the stream errors out.

- Observables are declarative. You define an observable function just like any other variable. **The observable starts to emit values** only **when someone subscribes to it**.



Observers (subscribers)

- The **Observable** is only useful if someone **consumes** the value **emitted** by the **observable**. We call them **observers** or **subscribers**.

- The **observers communicate** with the **Observable** using **callbacks**

- The **observer** must subscribe to the **observable** to receive the value from the observable. While subscribing, it optionally passes the three **callbacks (next(), error() & complete())**

Observable Creation

- The **observable constructor** takes the **observer (or subscriber)** as its **argument**. The **subscriber** will run when this **observable's subscribe()** method executes.

- The following example **creates** an **observable of a stream of numbers** 1, 2, 3, 4, 5

```
obs = new Observable((observer) => {
    console.log("Observable starts")
    observer.next("1")
    observer.next("2")
    observer.next("3")
    observer.next("4")
    observer.next("5")
  })
```

  - The variable obs is now of Type observable.
  - To make the **observable emit values**, we need to **subscribe** to them.

Subscribing to the Observable

The code below shows **subscribing** to an **observable** using the **observer object**. The **next** method is **invoked** whenever the **observable emits data**. It **invokes** the **error** method when **an error occurs** and the **complete** method when the **observable** completes.

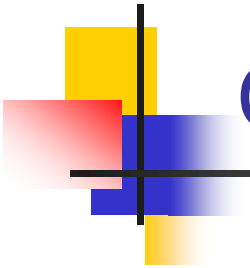
```
ngOnInit() {
 this.obs.subscribe(
  {
   next: (val) => {
    console.log(val);
   }, //next callback
   error: (error) => {
    console.log('error');
   }, //error callback
   complete:() => {
    console.log('Completed');
   } //complete callback
  }
 );
}
```

Subscribing to the Observable

```
export class AppComponent implements OnInit {


  obs = new Observable(observer => {
    console.log('Observable starts');
    observer.next('1');
    observer.next('2');
    observer.next('3');
    observer.next('4');
    observer.next('5');
  });

  ngOnInit() {
    this.obs.subscribe( {
      next: (val) => {
        console.log(val);
      }, //next callback
      error: (error) => {
        console.log('error');
      }, //error callback
      complete:() => {
        console.log('Completed');
      } //complete callback
    }
    );
  }
}
```

- The following example **creates** an **observable of a stream of numbers** 1, 2, 3, 4, 5

- To make the **observable emit values**, we need to **subscribe** to them.



Observable Operators

- The **Operators** are **functions** that operate on an **Observable** and **return a new Observable.**

- The power of **observable** comes from the **operators**. You can use them to **manipulate** the incoming **observable**, **filter it, merge it with another observable, alter the values or subscribe to another observable.**

- You can also **chain each operator one after the other using the pipe**. **Each operator** in the **chain** gets the **observable** from **the previous operator**. It modifies it and creates a **new observable**, which becomes the input for **the next observable**.

Observable Operators

- The following example shows the filer & map operators chained inside a pipe. The filter operator removes all data which is less than or equal to 2 and the map operator multiplies the value by 2.

```
obs.pipe(
 obs = new Observable((observer) => {
   observer.next(1)
   observer.next(2)
   observer.next(3)
   observer.next(4)
   observer.next(5)
   observer.complete()
 }).pipe(
   filter(data => data > 2),   //filter Operator
   map((val) => {return val as number * 2}), //map operator
 )
```

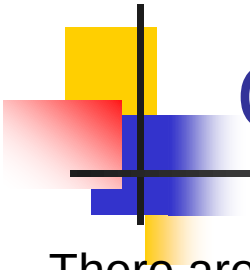
The input stream is [1,2,3,4,5] , while the output is [6, 8, 10].

Unsubscribing from an Observable

- Call the **unsubscribe()** method in the **ngOnDestroy** method.

```
ngOnDestroy() {
  this.obs.unsubscribe();
}
```

When we destroy the component, the observable is unsubscribed and cleaned up.



Observable creation functions

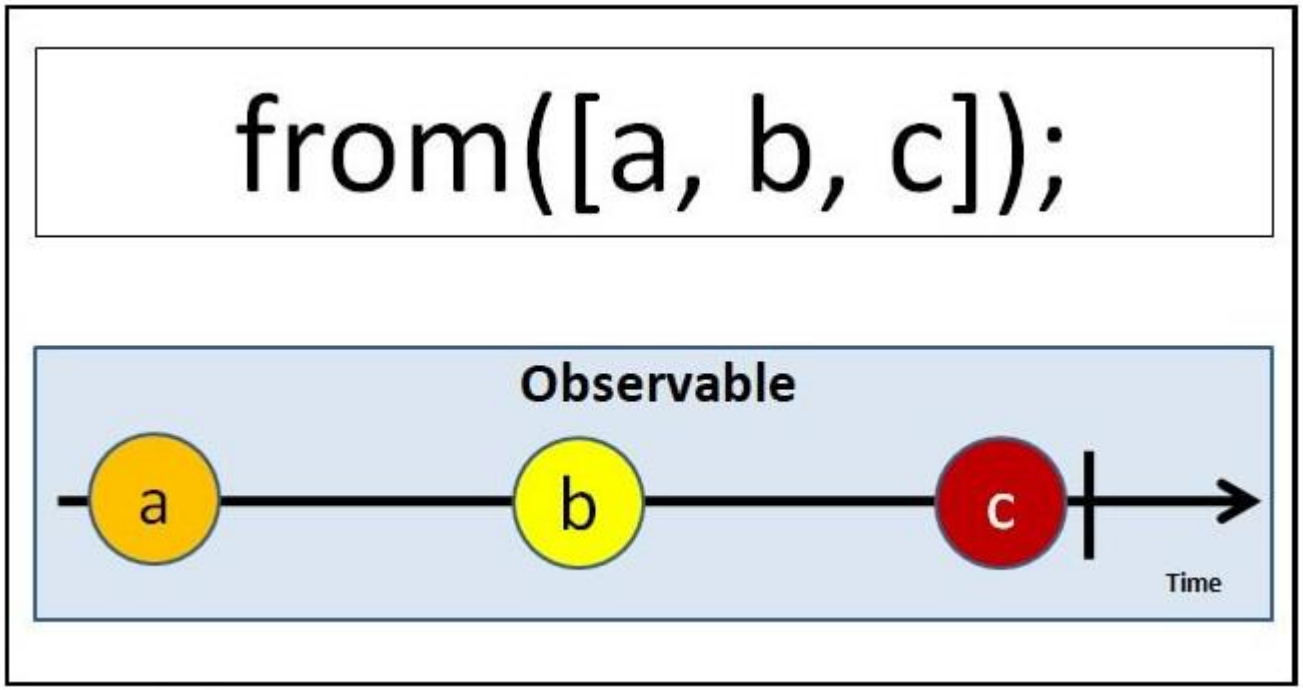
There are a number of functions that are available which you can use to create new observables. These operators help us to create observable from an array, string, any iterable, etc. Here are some of the operators

- create
- defer
- empty
- from
- fromEvent
- interval
- of
- range
- throw
- timer

All the creation related operators are part of the RxJs core library. You can import it from the ‘rxjs’ library

From Operator

From Operator takes only one argument that can be iterated and converts it into an observable.



To use **from** you need to import it from rxjs library as shown below.

```
import { from } from 'rxjs';
```

Using Observable Map

To use map first, we need to import it from the rxjs/operators library.

```
import { map } from 'rxjs/operators'
```

Next, create an observable from an array of numbers as shown below.

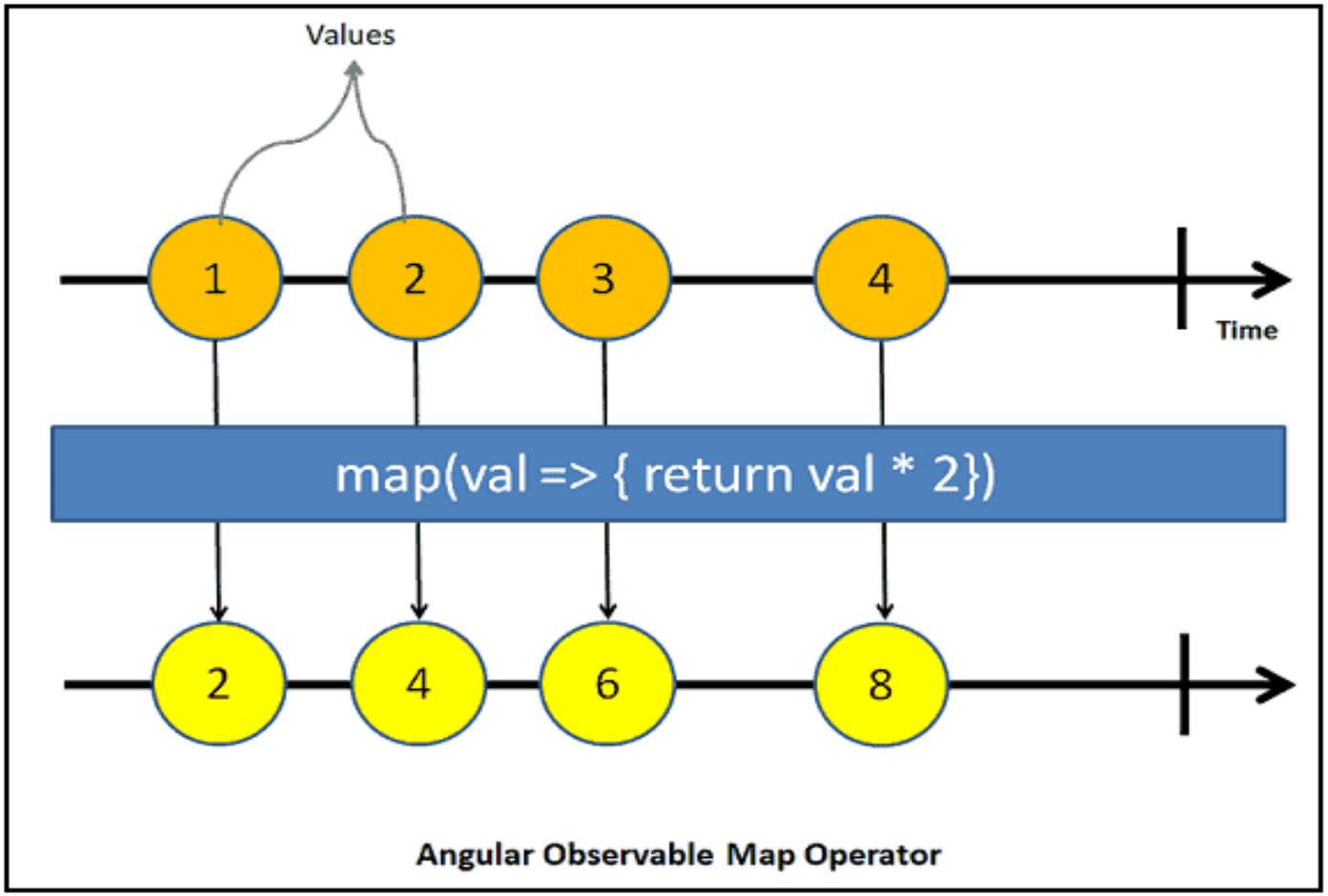
```
srcArray = from([1, 2, 3, 4]);
```

Use the pipe method to invoke the map method. map operator accepts only one argument val and returns it multiplied by 2. Finally, we subscribe and print the result in the console. The output is 2,4,6,8

```
multiplyBy2() {
  this.srcArray
  .pipe(map(val => { return val * 2}))
  .subscribe(val => { console.log(val)})
}
```

Using Map Operator

The following image explains how values from the source observable (i.e.1,2,3,4) go through the map which transforms it into new values by multiplying it by 2.



Using Observable

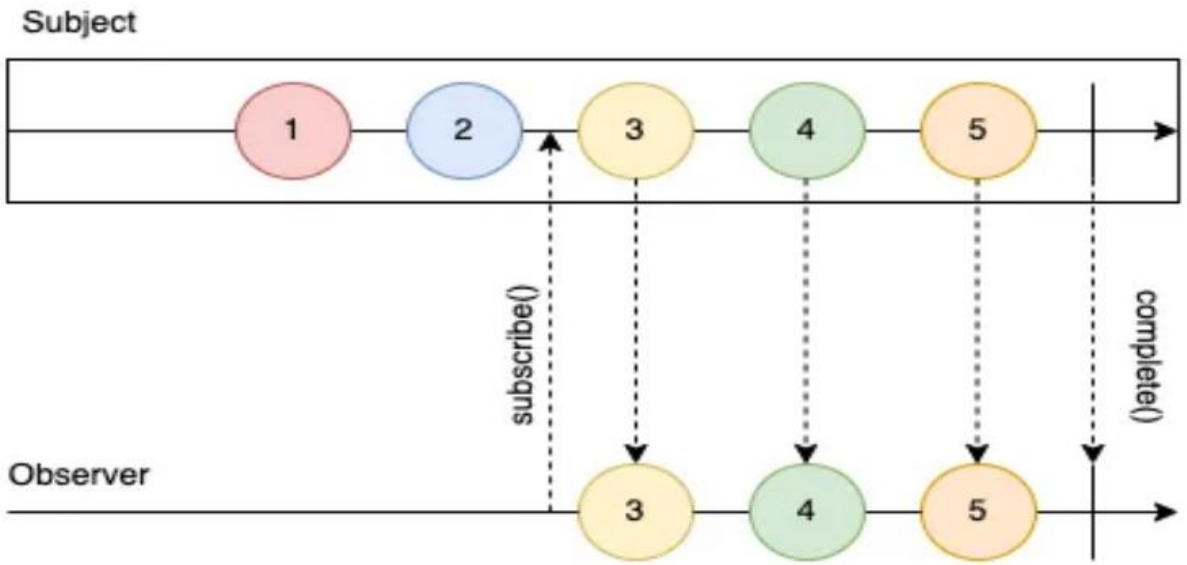
To create an observable from any event, first, we need to get the reference to DOM element using the viewchild & ElementRef. For example the following code gets the reference to the button element with the id #button

```
<div>
    <button #button (click)="buttonClick()">Click me</button>
</div>
```

```
@ViewChild('button', { static: true }) button!: ElementRef;
buttonSubscription!: Subscription;
```

```
buttonClick() {
  const clickObservable = fromEvent(this.button.nativeElement, 'click');
  this.buttonSubscription = clickObservable.subscribe(
    res => console.log(res)
  );
}
```

Subject

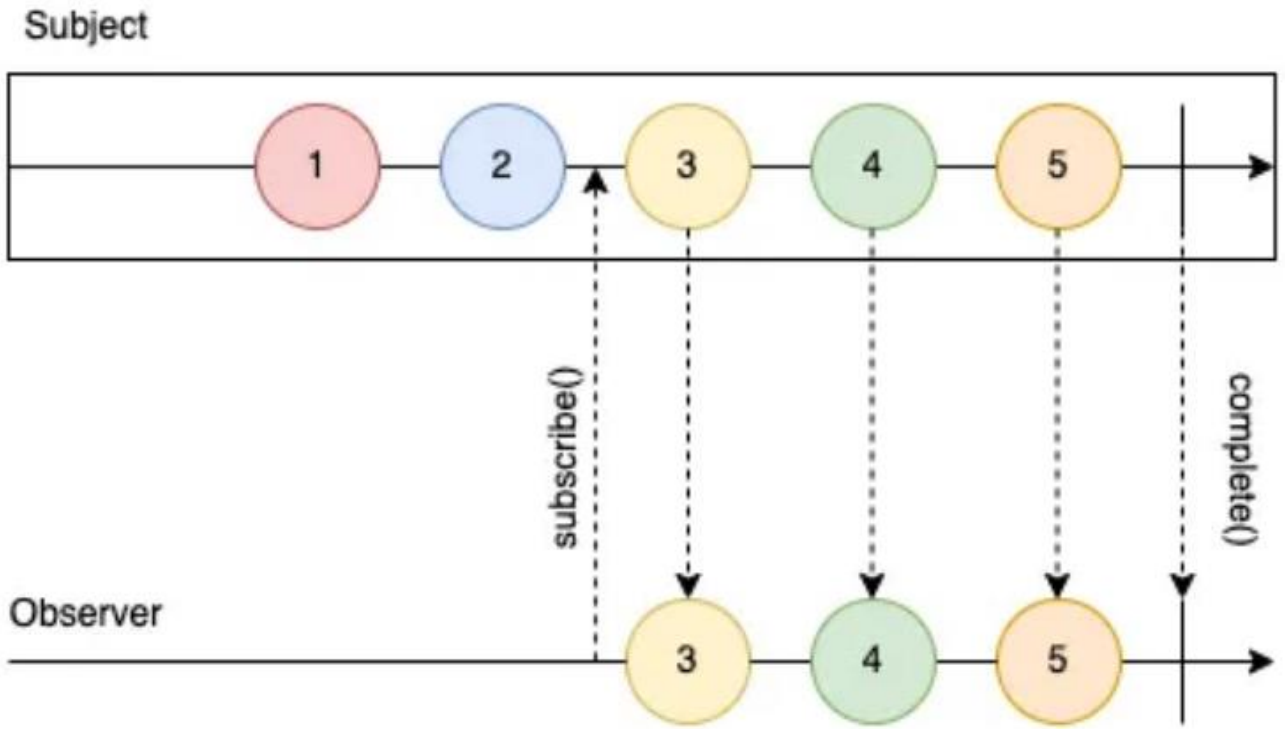


A **Subject** is a special type of **Observable** that allows values to be **multicasted to many observers.**

- Every **Subject** is an **Observable**. Given a Subject, you can subscribe to it and start receiving values normally.

- Every **Subject** is an **Observer**. To **send** a new value to the **Subject**, just call **next(value)**, and it will be multicasted to the **observers registered to listen to the Subject.**

Subject



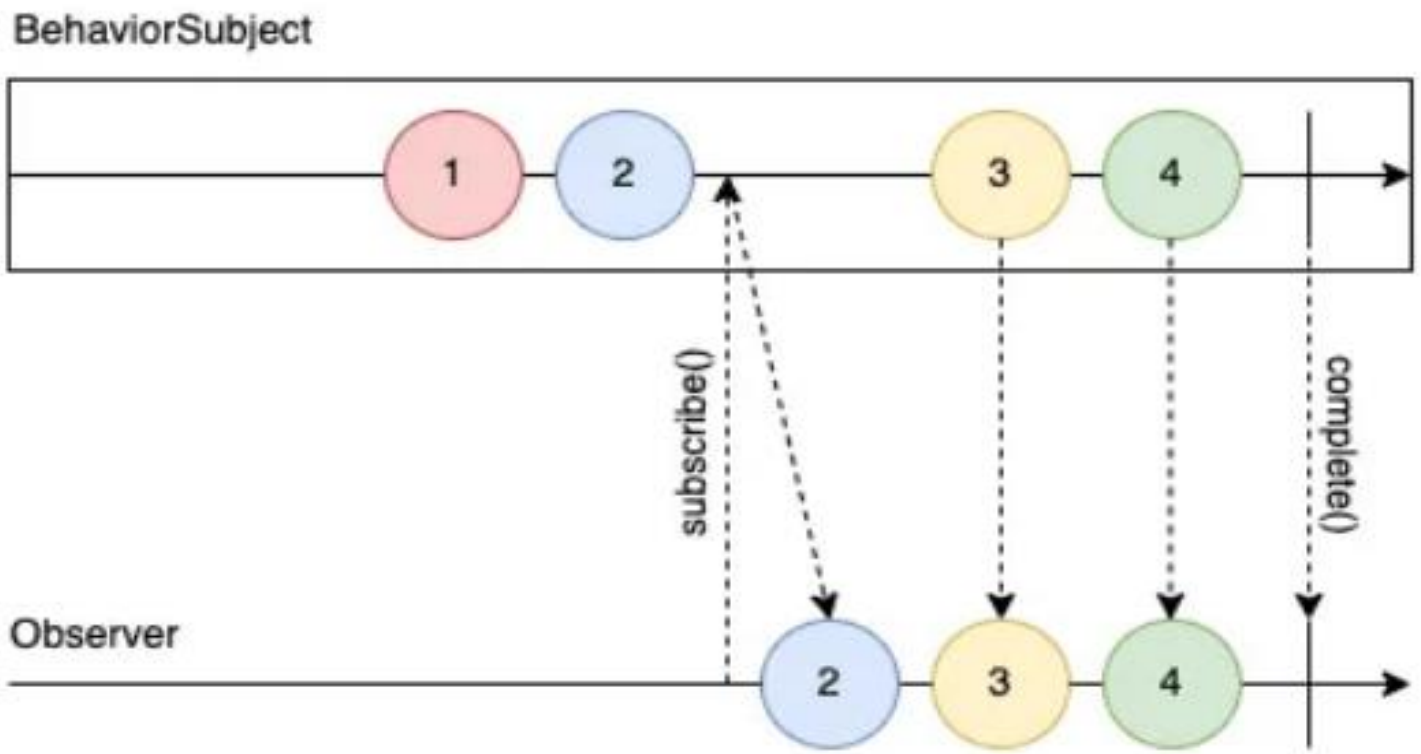
When you **subscribe** to a **Subject**, the **Observer** will **receive every value emitted after the subscription was made**. Any values emitted before the **subscription won't be received** by the **Observer**.

Subject

```
import { Subject } from 'rxjs';

const subject = new Subject<number>();

subject.next(1);
subject.next(2);

subject.subscribe({
  next: (v) => console.log(`Observer: ${v}`),
});

subject.next(3);
subject.next(4);
subject.next(5);

// Logs:
// Observer: 3
// Observer: 4
// Observer: 5
```

BehaviorSubject

BehaviorSubject behaves like a **Subject**, but the **Observer** also **receives** the last value **emitted before the subscription** was made.

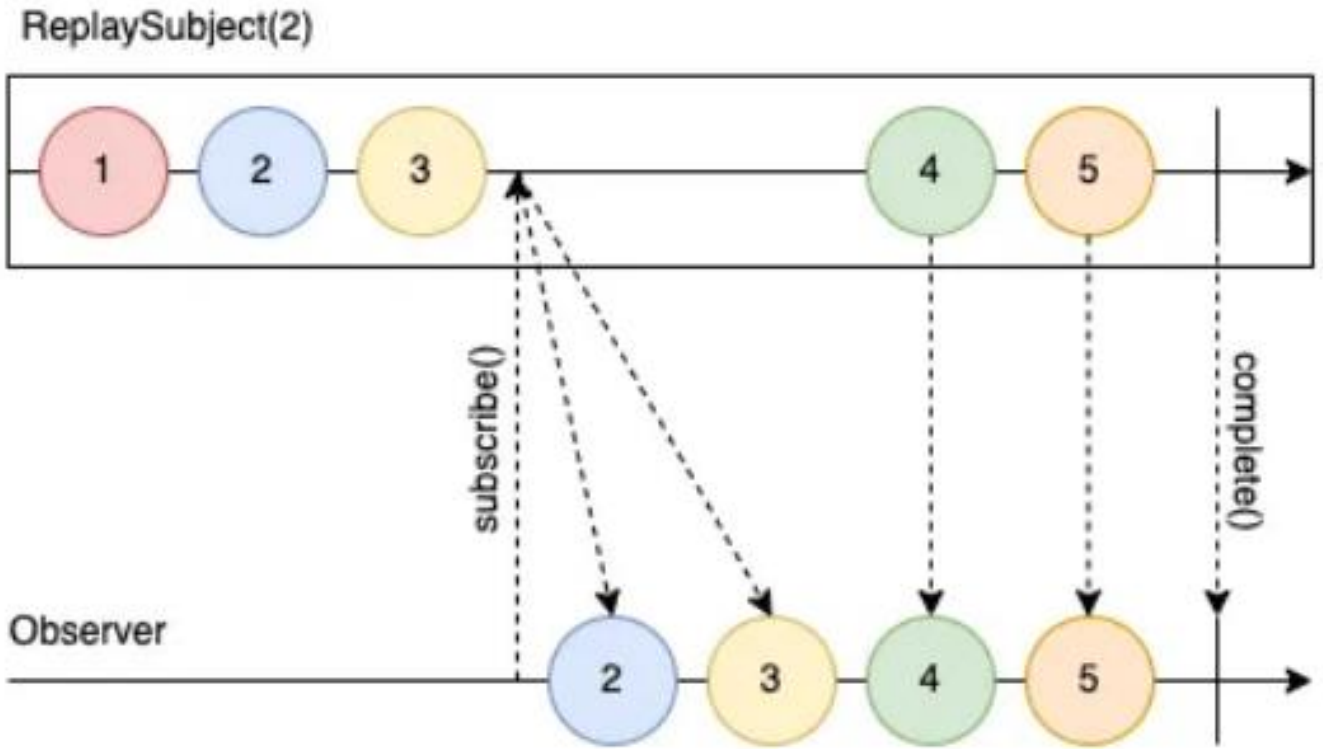


BehaviorSubject

```
import { BehaviorSubject } from 'rxjs';

const subject = new BehaviorSubject();

subject.next(1);
subject.next(2);

subject.subscribe({
  next: (v) => console.log(`Observer: ${v}`),
});

subject.next(3);
subject.next(4);

// Logs:
// Observer: 2
// Observer: 3
// Observer: 4
```

ReplaySubject

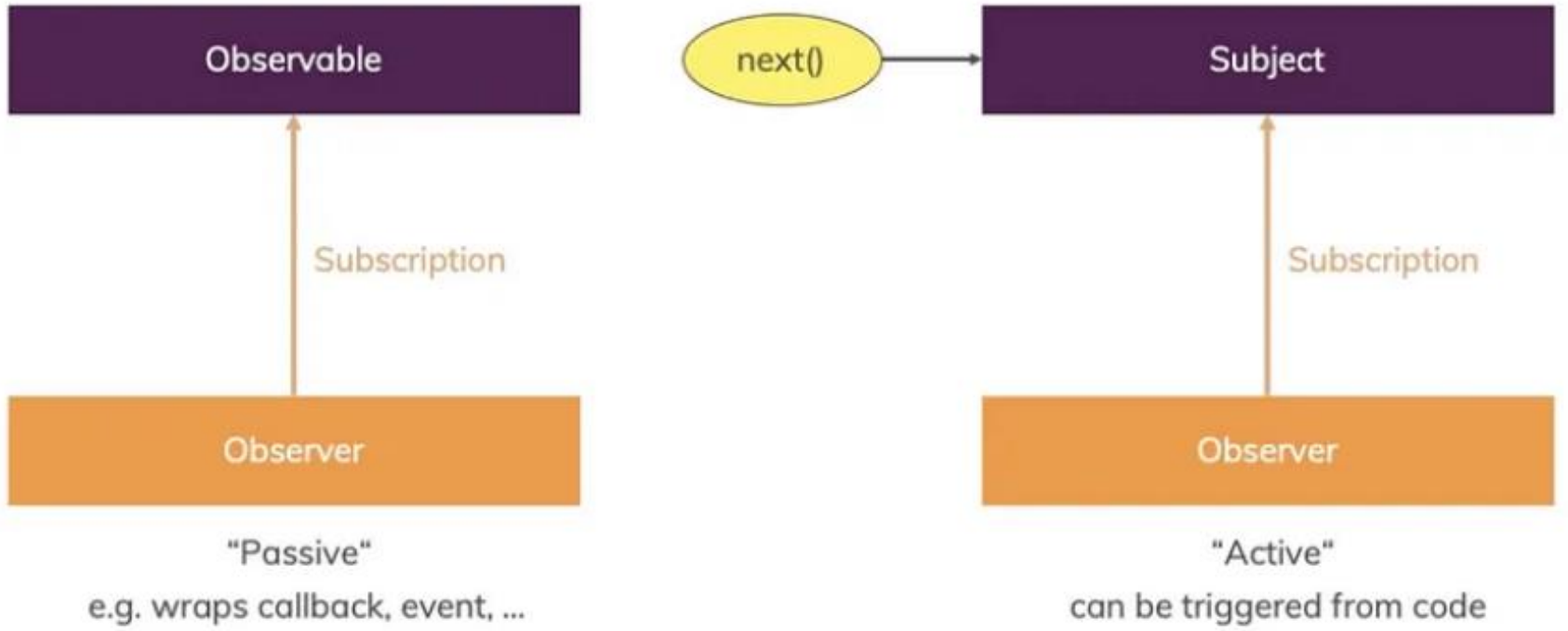
A **ReplaySubject** behaves like a **BehaviorSubject**, but instead of only emitting the last value, you can **record multiple values to replay them to new subscribers**. For example, you can buffer the last 2 values



ReplaySubject

```
import { ReplaySubject } from 'rxjs';

const subject = new ReplaySubject(2); // buffer 2 values for new subscribers

subject.next(1);
subject.next(2);
subject.next(3);

subject.subscribe({
  next: (v) => console.log(`Observer: ${v}`),
});

subject.next(4);
subject.next(5);

// Logs:
// Observer: 2 (Buffer value)
// Observer: 3 (Buffer value)
// Observer: 4
// Observer: 5
```

Observable vs Subject (difference)



Difference #1: **Observable** needs **observer interface** to see Observable behaviour while **Subject** acts as **both Observer** & **Observable**.

Subject can be used anywhere in the code to emit or subscribe a value but **Observable can be only be used freely to subscribe a value but you can't emit value outside from Observable scope**.

Observable vs Subject (difference #1)

```
export class TestComponent implements OnInit {
    public observableTxt: string | null = null;
    public subjectTxt: string | null = null;

    private subject: Subject<string> = new Subject<string>();
    private observable: Observable<string> = new Observable((observer) => {
        observer.next('Its observable');
    });

    public ngOnInit(): void {
        this.observable.subscribe((val: string) => {
            this.observableTxt = val;
        });

        this.subject.subscribe((val) => {
            this.subjectTxt = val;
        });

        this.subject.next('Its Subject');
    }
}
```

Subject can be used anywhere in the code to emit or subscribe a value but **Observable can be only be used freely to subscribe a value but you can't emit value outside from Observable scope**.

Observable vs Subject (difference #2)

Difference #2: **Observable** emits values **only when there is** a **subscriber**. If there are **no subscribers**, the Observable **does not emit** any values. The **Observable** "**delays**" its emissions **until** someone **subscribes** to it.

A **Subject can emit values** even if there are **no subscribers** yet. However, **if a subscriber subscribes later**, they will **miss any values** that were emitted **before** they subscribed. **The Subject emits values as soon as they are pushed**, and it does not "remember" or "cache" past emissions.

```
export class TestComponent implements OnInit {
    public observableTxt: string | null = null;
    public subjectTxt: string | null = null;

    private subject: Subject<string> = new Subject<string>();
    private observable: Observable<string> = new Observable((observer) => {
        // it is already subscribed before calling next()
        observer.next('Its observable');
    });

    public ngOnInit(): void {
        this.observable.subscribe((val: string) => {
            this.observableTxt = val;
        });

        // calling next() before subsribe to Subject
        this.subject.next('Its Subject');

        this.subject.subscribe((val) => {
            this.subjectTxt = val;
        });
    }
```

Observable vs Subject (difference #3)

Difference 3#: **Observables** are **Single Casting** while **Subjects are Multi Casting**. So this means when **Subject** emits a value then **all its subscribers** get **the same value** while in case of **Observable**, for each subscription Observable **emit a value separately**.

```
public ngOnInit(): void {
    this.observable.subscribe((val: string | number) => {
        this.observableTxt1 = val;
    });

    this.observable.subscribe((val: string | number) => {
        this.observableTxt2 = val;
    });

    this.subject.subscribe((val) => {
        this.subjectTxt1 = val;
    });

    this.subject.subscribe((val) => {
        this.subjectTxt2 = val;
    });
```

Observable vs Subject (difference)

```
private subject: Subject<string | number> = new Subject<string | number>();
private observable: Observable<string | number> = new Observable((observer) => {
    observer.next(Math.random());
});

public ngOnInit(): void {
    this.observable.subscribe((val: string | number) => {
        this.observableTxt1 = val;
    });

    this.observable.subscribe((val: string | number) => {
        this.observableTxt2 = val;
    });

    this.subject.subscribe((val) => {
        this.subjectTxt1 = val;
    });

    this.subject.subscribe((val) => {
        this.subjectTxt2 = val;
    });

    this.subject.next(Math.random());
}
```

Observable - Value1: 0.7821923780305577 Value2: 0.6228959705697719
Subject - Value1: 0.3728836813220071 Value2: 0.3728836813220071

Subjects are multicast

```
const subject = new Rx.Subject();

subject.subscribe({
  next: v => console.log("observerA: " + v)
});
subject.subscribe({
  next: v => console.log("observerB: " + v)
});

subject.next(1);
subject.next(2);

// output
// observerA: 1
// observerB: 1
// observerA: 2
// observerB: 2
```

Subject is a special type of Observable that allows values to be **multicasted to many Observers**. While plain Observables are unicast (each subscribed Observer owns an independent execution of the Observable), **Subjects are multicast**.