Contents

Day02 Components and Templates

PART 01 – THE VIEWCHILD() VIEW QUERY FUNCTION

Angular components now come with signal-based template queries of which **`viewChild()`** is part of. The other APIs include **`contentChild`**(), **`viewChildren`**() and **`contentChildren`**(). These signal-based APIs and can be used in place of the traditional decorator-based template queries like @**`ViewChild`**, @**`ContentChild`**, @**`ViewChildren`**, and @**`ContentChildren`**.

We will use **`viewChild()`** template query signal, to query the span element on the child component in our small app. We will then use that reference to change the text being displayed between the span tags, once the button is clicked.

1. Begin by importing the **`viewChild`**() function from **`@angular/core`** and declare a property of the **`Child`** class using this new function:

```
export class Child {
  empNameC = model("");
  private messageField = viewChild("spanMessage");
  onInput(event : Event){
```

Remember that the **`span`** tag on the template already has a template reference variable called **`#spanMessage`**. Do this in the **`Child`** component. You will need to import viewChild from **`@angular/core`**.

2. We also need to be more specific with the type of element we are working with, so lets add the **`ElementRef`** as a generic type:

```
  empName = model.required<string>();
  private messageField = viewChild<ElementRef<HTMLSpanElement>>("spanMessage")
```

The referenced element is a **`span`** tag, so **`HTMLSpanElement`**. Notice that I am using a double generic structure here, the outer element indicates the wrapper to an inner element. The inner type is the specific HTML element we need. This gives us access to all the properties and events of that element. Refer to Appendix B on the Day01 document. Import the **`ElementRef`** class from **`@angular/core`**.

This approach gives you **compile-time type safety**, meaning TypeScript will know that your referenced element is a **`span`** tag and will provide appropriate autocomplete suggestions and catch type errors. For example, if you try to access properties that don't exist on **`span`** elements, TypeScript will warn you before runtime.

3. With the HTML element referenced and accessed, we just need to know when to work with it, so when exactly do we pass a value into the span tag. We could use a lifecycle hook here, but one method is to use a constructor with an **`effect`**:

```
export class Child {
  empName = model.required<string>();
  private messageField = viewChild<ElementRef<HTMLSpanElement>>("spanMessage")
  constructor() {};
```

In JS based apps, timing is important since elements appear on the browser at different times.

4. Now we can add the **`effect()`** function from Angular:

```
  constructor() {
    effect(() => {
      const messageElement = this.messageField();
    });
```

Remember to import the **`effect()`** function from @**`angular/core`**. The function **`viewChild`**() will return a signal. Now, **`messageElement`** is a reference to the span element, which it itself is a signal. This means that if the span tag changes, **`messageElement`** updates. It works the other way also, we can now update the span text content using this same mechanism.

5. After all of these changes we can now check one more time for the element. If it is available (not empty), pass some content into the tag:

```
  constructor() {
    effect(() => {
      const messageElement = this.messageField();
      if (messageElement) {
        messageElement.nativeElement.textContent = `Hello, ${this.empNameC()}`;
      }
    });
  };
```

After these changes, notice that the text content of the **`<span>`** tag takes whatever value appears in the input form control (text box). Now if you change the value inside of this input box, **both** the welcome message and the **`<span>`** tag message updates. Once again, the property **`empNameC`** is a signal also.

PART 02 – CONTENT PROJECTION

Content projection is more like a pattern, it is Angular's way of implementing *transclusion*. It is used to insert, or "project," external HTML content into a component's template at a specific location. Just tell the receiving component where exactly that *imported* content should be rendered within it's template.

Content projection is used to build reusable and configurable UI components. This means that the content being passed is changeable. Think of content projection like placeholders that gets filled with HTML as needed. Content project in Angular is accomplished with a special custom HTML tag; **<ng-content>**. Similar to <slot> in web components. (Transclusion is just showing one HTML document into another).

1. Lets create a new component called **Messages**. This component will be the receiving component. In the terminal window:

```
ng g c components/messages
```

   We will include the CSS file this time, so no flags. We took care of test files when we created the app on Day01. So now we should receive three files from the Angular CLI.

2. Add some generic content to the **Messages** component:

```
<div class="msg">
  <h3>---Start of message----</h3>
  <h4>---End of message----</h4>
</div>
```

```
.msg{
  margin-top: 20px;
  border-top: 20px;
}
```

3. In order to allow HTML content to come through into this new component, we must provide a special tag and location inside of the Messages component:

```
<div class="msg">
  <h3>---Start of message----</h3>
  <ng-content></ng-content>
  <h4>---End of message----</h4>
</div>
```

   Notice we can decorate this component with any HTML we need.

4. We will use this component from the app's **Child** component, so import it there:

```
import { Component, effect, ElementRef, model, viewChild } from '@angular/core';
import { Messages } from '../../messages/messages';
@Component({
```

   Also add this component to the **imports[]** array (in the child.ts file)

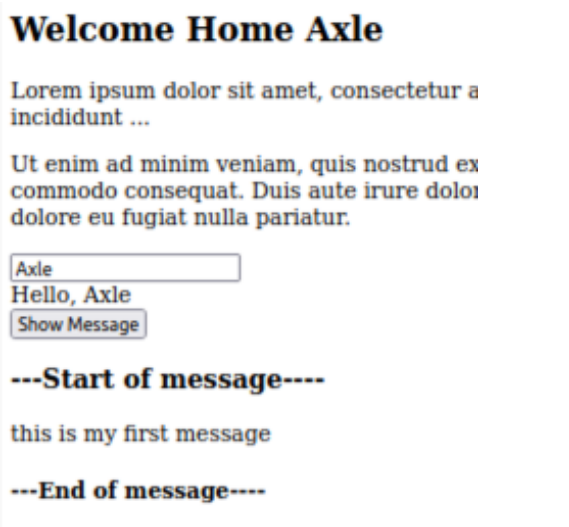
5. Once imported, use the selector of this imported component anywhere in the template of the consumer component, **`Child`** in this case:

```
<div>
  <button (click)="spanMessage.textContent='Hello'">Show Message</button>
</div>
<app-messages></app-messages>
```

Be careful of the selector that Angular chooses, it may not be what you are thinking it is.

6. Add some textual content to the <app-messages> tag on the Child component:

```
<app-messages>
  this is my first message
</app-messages>
```



6. Remember this feature is reusable, so we should be able to do something like this:

```
<app-messages>
  this is my first message
</app-messages>
<app-messages>
  this is my second message
</app-messages>
```

This works well:

7. And if we added one more, then:

```
<app-messages>
    this is my first message
</app-messages>
<app-messages>
    this is my second message
</app-messages>
<app-messages>
    this is my third message
</app-messages>
```

All three messages end up in the one container on the **`Messages`** component.

8. Change the Child component code to this instead:

```
<app-messages>
  <p>This is my first message</p>
  <p>This is my second message</p>
  <p>This is my third message</p>
</app-messages>
```

9. Here is something interesting, if we had just two **ng-content** tags, we still get the same result:

```
<div class="msg">
  <h3>---Start of messages----</h3>
    <ng-content></ng-content>
    <ng-content></ng-content>
  <h4>---End of messages----</h4>
</div>
```

10. But over in the parent, I added an attribute to one of the **<p>** tags, it can be anything, I am using **m1** here:

```
   <p m1>this is my first message</p>
   <p>this is my second message</p>
   <p>this is my third message</p>
```

There still should not be any changes in the output at this point

11. Then in the **Messages** component, I look for that specific attribute using another special attribute, **select**:

```
<div class="msg">
  <h3>---Start of messages----</h3>
    <ng-content select="[m1]"></ng-content>
    <ng-content></ng-content>
  <h4>---End of messages----</h4>
</div>
```

Again nothing changes, but see below.

12. What if I changed the order of the **p-tags** in the parent:

```
<app-messages>
  <p>This is my first message</p>
  <p>This is my second message</p>
  <p m1>This is my third message</p>
</app-messages>
```

13. The Messages component now adjusts it's output:

In this way, you can create some kind of order in the presentation. There are other ways of achieving order in your presentation. See the documentation for the other ways.

PART 03 – CUSTOM DIRECTIVE AND RENDERER

A directive is simply an instruction to the DOM. It tells the DOM to do something like add a new pair of `<div>` tags or change the look and feel of some HTML element.

Components are directives. We used a directive on Day01, `ngModel`. There are structural directives like `*ngIF` and `*ngFor`. However mostly you will come across attribute directives where the directive is used like an attribute.

With Angular we can develop our own custom directives. Available are attribute and structural directives. Attribute directives affect how HTML elements are rendered in the browser. Structural directives affect how the DOM builds itself by controlling the addition or removal of elements.

As a reminder, directives can be used just like an HTML tag, a class or as a directive on an existing element. This part focusses on the latter.

In this example we will create a directive that will change the text of an element when the element is hovered over by the mouse.

A note of caution. It is possible to manipulate the DOM directly using ElementRef and the nativeElement property of that element. With this method, a life cycle hook is completely bypassed. Although this works, it is not recommended. This bootcamp example shows the recommended method, at this point in time.

1. Use the CLI to generate a new directive

```
ng generate directive boldNBlue --skip-tests
```

This command will create the class in the app directory that will hold the logic for this hover. Remember the flag is optional, we did it on Day01.

2. Just to see the directive work, add a log statement in the constructor and we will be able to see the message in the console window:

```
import { Directive } from '@angular/core';
@Directive({
  selector: '[appBoldNBlue]'
})
export class BoldNBlue {
  constructor() {
    console.log("BoldNBlue works!!");
  };
};
```

If we see the console message, it means that the directive was wired properly, refer to #4 below for the actual message display. Notice the value in the selector key/value pair.

3. Lets now use this directive in the Home component, as you might have noticed the pattern by now, we import the directive then add it to the imports array:

```
import { Child } from "../child/child";
import { BoldNBlue } from '../../bold-nblue';
@Component({
  selector: 'app-home',
  imports: [Child, BoldNBlue],
  templateUrl: './home.html',
  styleUrl: './home.css'
})
```

You may see a yellow squiggly line under the **BoldNBlue** element in the imports array. Once we apply the directive, this warning will go away, see below.

4. Before continuing you should run the app and you should not see any messages. Now lets apply the directive to an HTML element on the template:

```
<h2>Welcome Home {{ empName() }}</h2>
<p appBoldNBlue>Lorem ipsum …
<p>Ut enim ad minim veniam, …
<app-child [(empName)]="empName"></app-child>
```

Add the directive to any of the HTML elements on the home.html template and run the application. You should now see the message in the **console** window.

5. We need three additional classes to help with this directive, so in the bold-nblue.ts file, add the following:

```
import { Directive, HostListener, ElementRef, Renderer2 } from '@angular/core';
```

6. Now instantiate the **Render** and **ElementRef** classes via the constructor:

```
export class BoldNBlue {
  constructor( private el : ElementRef, private render : Renderer2 ) {
    //console.log("BoldNBlue works!!");
     };
```

This gives us **el** as a reference to **ElementRef** and **render** as a reference to the **Render2** class.

7. Once #6 happens, those objects will be available to the entire class. Lets now implement the **@HostListener()** decorator function. Add this function below the constructor in the bold-nblue.ts file:

```
  @HostListener("mouseenter") onMouseEnter() {
    console.log("BoldNBlue works!!");
  }
```

Once again if you run the app, the console window will show the logged statement. Remember that the **BoldNBlue** directive was placed on the first paragraph on the **Home** template as an attribute. The **@HostListener** decorator allows a directive or component to listen for events on its host element and react to them by executing a specific method.

8. We can now add some logic using **Renderer2** and the **ElementRef** objects:

```
  @HostListener("mouseenter") onMouseEnter() {
    //console.log("BoldNBlue works!!");
    this.render.setStyle(this.el.nativeElement, 'color', 'blue');
    this.render.setStyle(this.el.nativeElement, 'font-weight', 'bold');
  }
```

Now when you run the app, the second paragraph will be bold and blue once the mouse hovers over it. To remove the style, see below.

9. Now just add a similar block of code for the **mouseleave** event, also available from **@HostListener()**:

```
    this.render.setStyle(this.el.nativeElement, 'font-weight', 'bold');
  };
  @HostListener("mouseleave") onMouseLeave() {
    this.render.setStyle(this.el.nativeElement, 'color', '');
    this.render.setStyle(this.el.nativeElement, 'font-weight', '');
  };
```

10. Note that it is the directive that is responsible for all the functionality involved here, not the component using it. The component consuming this directive just needs to place the directive where it is most effective. In our case I placed the custom directive **BoldNBlue** on the first paragraph of the consumer component (#4).

PART 04 – CONTROL FLOW

With Angular templates we can *conditionally* show, hide, and repeat HTML elements.

1. We will use a checkbox on the **`Child`** component to show or hide the **`Messages`** component (**`app-messages`**). First let's add a checkbox to the template:

```
</div>
 <input
  type="checkbox"
  value="Show Messages"
 >
<app-messages>
```

2. Connect this HTML element to a function in the TS file and give it a reference:

```
   </div>
    <input
     #showMsg
     type="checkbox"
     value="Show Messages"
     (change)="onCheckboxChange($event)"
    >
   <app-messages>
```

   Remember all HTML documents emit an event once interaction happens (**`$event`**). We must now create a function called **`onCheckboxChange()`** in the child.ts file.

3. Here is the function in the TS file:

```
  };
  onCheckboxChange(event: Event) {
    const checked = (event.target as HTMLInputElement).checked;
    if (checked) {
      console.log('Checkbox is checked');
    };
  };
```

   We can now call this method from the template using the checkbox. In the console window, the log message should be there and the checkbox will refresh and show a checkmark once checked. Don't worry about the uncheck event for now.

4. (Optional) You can use the checkbox-lg style from the home.css file to make the checkbox bigger. Just cut the style from home.css and put it in styles.css:

```
.checkbox-lg {
      transform: scale(3.5);
      margin: 20px;
}
```

   The styles.css file is global.

5. With the checkbox and logic in place we can apply that same logging logic to the entire component. Wrap the **<app-messages>** block into an **@if()** directive:

```
@if(showMsg.checked){
  <app-messages>
      <p>this is my second message</p>
      <p>this is my third message</p>
      <p m1>this is my first message</p>
  </app-messages>
 }
```

Now if you check the box we added #1, the entire block of three paragraphs appear. The uncheck works automatically, no further programming.

6. We can take it further with an **@else()** directive:

```
          <p>this is my third message</p>
          <p m1>this is my first message</p>
     </app-messages>
    } @else {
     <p>Messages are hidden</p>
 }
```

So, now, the first line of text you see is the paragraph in the **@else** clause. Then once you check the box, you see the three paragraphs in the **app-messages** section. Use this for placeholder type text content.

7. Moving on you might be wondering why we need communication between the template and the TS code for this functionality to work. Can we achieve the same functionality without the **onCheckboxChange()** function? Well, we already have a reference, **showMsg**, so lets remove the event binding from the checkbox:

```
   <input
     #showMsg
     type="checkbox"
     value="Show Messages"
     (change)="onCheckboxChange($event)"
    >
```

If you test now, you will see it does not work, we need some way to trigger the change event, see below.

8. Without the component class function, the event does not fire and show the messages. However Angular now has the ability to trigger change detection on the template itself. Add the following code to trigger change detection and a refresh:

```
<input
  #showMsg
  type="checkbox"
  value="Show Messages"
  (change)="0"
 >
 @if(showMsg.checked){
```

It should work as in #5. Note, *change detection* in Angular is a huge topic. We cover that in another bootcamp.

Angular also have the following control flow statements:
@else if
@for
@empty
@swithch
@case
@default

PART 05 – PIPES

A Pipe in Angular is a function that allows you to change the appearance of data, so it displays better to the user. They are typically used to format text, currency, dates, or other data. Pipes do not change the value of data, only the presentation of that data to the user.

Pipes are implemented with a vertical bar or "pipe" character (|) in the template. Angular has several built-in pipes that are ready to be used. We used the **`async`** pipe in the **Introduction to Angular bootcamp**.

Some pipes can take parameters. The **`DatePipe`** can take parameters such as "short" or "longData":

1. To see how pipes work, lets use one of Angular built-in pipes that work with calendar dates. The **`DatePipe`** Takes a JavaScript Date object, a Unix timestamp, or a date string and formats that value into a human-readable format. Before we use the pipe in the **`Home`** component, lets import the **`CommonModule`**. This module contains the **`DatePipe`** functionality:

```
import { CommonModule } from '@angular/common';
@Component({
  selector: 'app-home',
  imports: [Child, BoldNBlue, CommonModule],
  templateUrl: './home.html',
  styleUrl: './home.css'
})
```

The **`DatePipe`**, along with other fundamental Angular pipes like **`CurrencyPipe`**, **`JsonPipe`**, and **`UpperCasePipe`**, is part of the **`CommonModule`**

2. In the class, create a new date with the usual JS **`Date()`** function:

```
export class Home {
  empNameH = model("Axle");
  fName = "Jane";
  currentDate = new Date();
```

3. Now just add **`currentDate`** to the template and pass it through the pipe:

```
<h2>
  Welcome Home {{ empName() }}.
  Today is {{ currentDate | date }}
</h2>
```

4. Remember parameters, well lets modify the date with a parameter:

```
Today is {{ currentDate | date:' dd-MM-yyyy' }}
```

You can also use **`short`**, **`long`**, **`full`** and so on instead of formats like **`dd-mmm-yyyy`**

5. Lets create a custom pipe that will take a nine-digit telephone number and transform it into the format *(123) 456-7890*. First create the pipe like all other classes in Angular:

```
ng generate pipe naPhNo
```

I am just using **`naPhNo`** to represent North American Phone Number

6. We get the **`transform()`** function for free but it can accept anything, lets strengthen this a bit to accept only strings or numbers:

```
  transform(value: string | number): string {
    return "";
  }
```

Also, we will return an empty string to remove the error, so add that code as well.

7. We should now check for null or undefined before proceeding:

```
  transform(value: string | number): string {
    if (value === null || value === undefined) {
      return '';
    }
```

8. It is a good idea to take the value we received from use of the pipe and turn it into a string for processing. Also remove any non-digit characters:

```
      return '';
    };
    const strValue = value.toString();
    const cleanValue = strValue.replace(/\D/g, '');
    return "";
```

9. In North America all telephone numbers are written as nine digits, so lets check for this. If it is not 9 digits just return the original, do not transform it:

```
    const strValue = value.toString();
    const cleanValue = strValue.replace(/\D/g, '');
    if (cleanValue.length !== 10) {
      return strValue;
    };
    return "";
```

Of course you can add any logic to this part, I just return the original string that was passed in as a parameter. It did not meet our criteria for North America.

10. Finally add the logic using Regular Expressions to format the number according to North American standards:

```
      return strValue;
    };
    const formattedNumber = cleanValue.replace(
      /^(\d{3})(\d{3})(\d{4})$/,
      '($1) $2-$3'
    );
    return formattedNumber;
  };
```

Instead of returning an empty string, replace that empty string with the variable holding the formatted telephone number. At this point in the function we are guaranteed to have a string, so we satisfy the return type of this function.

11. Now for the application of this custom pipe, well that takes place on the template. But for now I will create a 9-digit number on the class then apply the pipe to that number on the template, so in the class add a member:

```
export class Home {
  empName = signal("Axle");
  currentDate =new Date();
  plainNumber : number = 1234567890;
}
```

I am using explicit typing here for **plainNumber**. This is added guarantee that **plainNumber** is a numeric value. Typically, a pipe is used in a dynamic situation for example modifying a value that a user typed in or retrieving data from a database. Since this is just a simple demonstration, I am using just a value declared as a property.

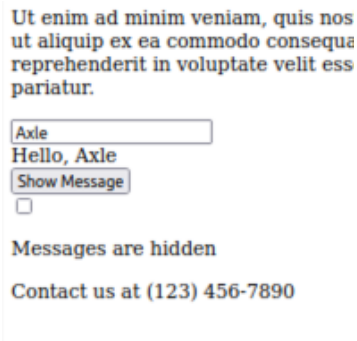
12. To make everything work, we must import the pipe, so in the TS file of Home component:

```
import { CommonModule } from '@angular/common';
import { NaPhNoPipe } from '../../na-ph-no-pipe';
import { Child } from "../child/child";
```

Remember to add this pipe to the imports array!

13. Finally in the template:

```
<app-child [(empName)]="empName"></app-child>
<div>
  Contact us at {{ plainNumber | naPhNo }}
</div>
```



14. To accept a parameter it is as simple as adding it to the **`transform()`** method. Lets accept a parameter called 'S' for Spanish style telephone numbers:

```
  transform(value: string | number, country: string): string {
    if (value === null || value === undefined) {
      return '';
```

Remember to add this pipe to the imports array!

15. Then add your logic, here I am using regular expressions again. Add this logic above the **`formattedNumber`** logic:

```
    if (country === 'S') {
      return cleanValue.replace(
        /^(\d{3})(\d{2})(\d{2})(\d{3})$/,
        '$1 $2 $3 $4'
      );
    };
    const formattedNumber = cleanValue.replace(
```

Note: I am not exactly sure about how to format telephone numbers in Spain, but the important thing here is to accept a parameter. I did search and I saw two different version of Spanish telephone number styles.

16. To implement the parameter, do so in the template:

```
<div>
  Contact us at {{ plainNumber | naPhNo:'N' }}
</div>
```

So if N, it will print North American, but replace N with an S and it prints Spanish style.

PART 06 – (OPTIONAL) @DEFER

Deferrable views (@defer blocks,) reduce the initial bundle size of your application by deferring the loading of code in the browser, until some trigger that you program. This means a faster initial load and improvement in Core Web Vitals (CWV).

1. We will use the same telephone number, on the **Home** component, example as the piece of data we wish to defer. First add a pair of **** tags around the telephone number

```
<div>
  Contact us at <span> {{ plainNumber | naPhNo:'N' }} </span>
</div>
```

This isolates just the phone number. But you can do this with any block of template code.

2. Now we simply wrap this piece of code inside the **@defer** block using curly braces:

```
<div>
  Contact us at @defer() { <span> {{ plainNumber | naPhNo:'N' }} </span> }
</div>
```

We did not need the tags but good idea to use them, mainly for future configuration.

3. We just need some kind of trigger. Luckily when the *core template syntax* was introduced in Angular 17, a few triggers were part of that module, here I will use the **on hover** trigger:

```
<div>
  Contact us at @defer (on hover ) { <span> {{ plainNumber | naPhNo:'N' }} </span>
}
</div>
```

4. One last thing we can do here, we can add some text that will be a placeholder but replaced by the telephone number once the hover event occurs:

```
  Contact us at @defer (on hover) {
    <span>{{ plainNumber | naPhNo:'N' }}</span>
  } @placeholder {
    <span>Phone number will appear on hover</span>
  }
```

Note: **on hover** is part of the core framework along with **on idle, on timer** etc.

APPENDIX A – TYPESCRIPT INTRODUCTION

Most of the folks on this call already know that TypeScript is a superset of JavaScript. This means that any JS code you see, is also TS code. The purpose of TS is to make you a better JS programmer. The main idea behind TS is static typing. This means that we must declare the data type in advance and it cannot (should not) be changed. Static typing means that we get IDE support, so we can eliminate certain bugs early in development.

1. The primitive types in TS is considered to be number, string and Boolean. Add a new .ts file to your project and add the following lines:

```
let firstName : string;
let maxAllowed : number;
let isAuthorized : boolean;
```

   Notice the types are all lower case to specify the type and not the object. Two other types which are considered primitive are *null* and *undefined*.

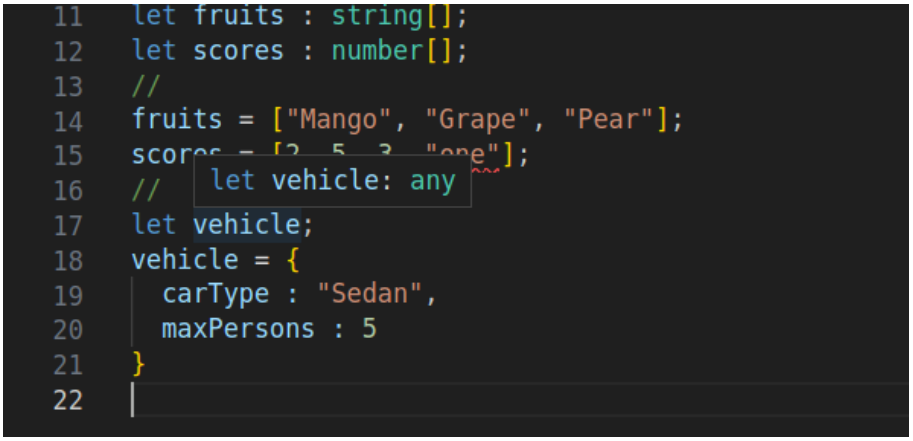
2. Complex types include arrays and objects.

```
let fruits : string[];
let scores : number[];
fruits = ["Mango", "Grape", "Pear"];
scores = [2, 5, 3, "one"];//NOT possible, strings NOT allowed
```

3. The object type can be a composite of other types. Below we declare a vehicle type and then attempt to define what the type should look like:

```
fruits = ["Mango", "Grape", "Pear"];
scores = [2, 5, 3, "one"];
//
let vehicle;
vehicle = {
  carType : "Sedan",
  maxPersons : 5
}
```

   The problem with this approach is that TS is left to define the type, and it will define this vehicle type as `any`, see image below:



4. The advantage in this situation of using TS is that we can now employ Object Type Definition, in other words we describe what our object should look like:

```
let vehicle : {
  carType : string,
  maxPersons : number
};
vehicle = {
  carType : "Sedan",
  maxPersons : 5
}
```

So now `vehicle` is valid.

5. The function below has two parameters that will be interpreted as type `any`:

```
function modifiedType(p1, p2) {
  return p1 + p2;
}
```

6. We can make it a bit clearer by declaring that the parameters are indeed of type `any`:

```
function modifiedType(p1:any, p2:any): any {
  return p1 + p2;
}
```

The return type is also `any`

7. The IDE does not complain if we call the function in #6 with any of these statements

```
  return p1 + p2;
}

modifiedType(10, "Hello");
modifiedType(true, "Hello");
modifiedType("Hello", {});
```

In some cases this is exactly what we need

8. With Generics, we can create our function and then decide at the time we call the function, exactly what type we want it to work with:

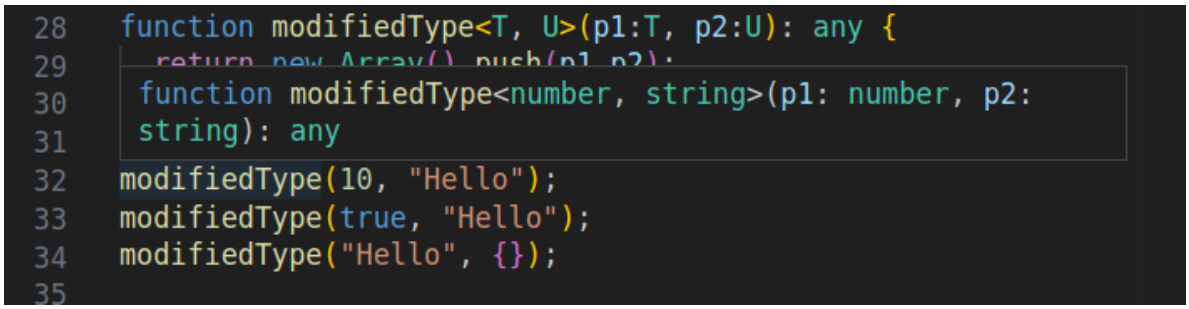
```
function modifiedType<T, U>(p1:T, p2:U): any {
  return p1 + p2;
}
```

Now, the return will not work. Typescript knows that it cannot add or concatenate certain types, for example object types.

9. Lets return an array instead, but one that contains both parameters:

```
function modifiedType<T, U>(p1:T, p2:U): any {
  return new Array().push(p1,p2);
}
```

Now, if you hover over the three calls to the function, you will see that the IDE has made a conclusion of what the types are based on when the function is called and what is being passed into the function. This is the beginnings of Generics.



In the image above notice that the interpretation of line 32 is that we are calling the `modifiedType` function with a number and a string. Not only that, we are guaranteed that the array being returned will have a number and a string.